Web Application Developer Security Handbook

May 2014

Contents

Introduction

Prelude

In 2008, Fox Digital Security Services produced a Web Application Developer Security Handbook based on the OWASP Top 10 security issues from 2007. This handbook was used extensively within Fox to train developers on the most commonly seen security vulnerabilities in Web applications, and how to defend against them.

The security landscape has changed significantly in the last seven years. As a result, Fox Digital Security Services has updated the web application developer security handbook based on the most current version of the OWASP Top 10. This document is a result of that effort.

The aim of this document is to serve as a practical developer's reference book for the topics that are covered. Code samples are used as frequently as possible to illustrate difficult concepts, and demonstrate the proper way to mitigate vulnerabilities. This document includes sample code in the PHP, Java, and ASP.NET languages.

As not all topics lend themselves to code samples, some sections do not include specific examples. The brief section on Cloud Security is a notable example of this. Additionally, OWASP sections like Outdated Software also do not include code samples, as the underlying issue is not specific to any development language.

When possible, the authors have strived to include real-world examples of vulnerabilities covered in this handbook. The now infamous Heartbleed vulnerability, for example, is included in the Outdated Software section. Publicly disclosed vulnerabilities targeting some of the largest applications on the Internet are also included, as real-world examples are often the best way to understand both how a vulnerability works, and its impact.

Each section contains a list of Additional Resources that can be used to further explore the corresponding topic. Developers are strongly encouraged to study those resources, as it is difficult to cover every nuance and caveat of a single issue in such a targeted document as this one.

***NOTE**: The Web Application Developer Security Handbook is designed as a reference guide for the most prevalent vulnerabilities found in today's environment. As a web application developer, it is your responsibility to understand the context in which code is executing and devise a proper strategy for mitigating risks. The code samples presented should not be relied upon "as is" to build secure applications.*

Web Application Security in 2013

There have been significant changes in the OWASP Top 10 from 2007 to 2013. Of particular interest is the addition of cross-site request forgery (CSRF), a vulnerability that first appeared on the OWASP Top 10 soon after the 2008 Web Application Developer Security Handbook was released. Developers should pay close attention to CSRF, as it is an especially pernicious

Introduction

vulnerability, and can be quite difficult to mitigate.

This document attempts to cover OWASP Top 10 entries from 2013. A side-by-side of comparison of OWASP Top 10 2013 vs. 2007 is presented below, with the new vulnerability categories emboldened to distinguish them from the existing vulnerability categories:

OWASP 2013:	**OWASP 2007:**
Injection	Unvalidated User Input
Broken Authentication and Session Mgt	Broken Access Control
XSS	Broken Authentication and Session Mgt
Direct Object Reference	XSS
Security Misconfiguration	Buffer Overflow
Sensitive Data Exposure	Injection Flaws
Missing Function Level Access Control	Improper Error Handling
CSRF	Insecure Storage
Outdated Software	Application Denial of Service
Open Redirect	Insecure Configuration Management

While the emboldened subjects are reviewed in this document, new content has also been included regarding security advancements in cross-site scripting (XSS), particularly with regard to DOM-based XSS vulnerabilities, and SQL injection. Due to the prevalence of these vulnerabilities in Web applications, general remediation and mitigation techniques for both XSS and SQLi have been documented.

Input Validation

As input validation is a key first step to preventing security vulnerabilities, the core guidelines are presented below. When performing input validation, developers should validate input against the following four constraints:

- The data length (e.g., 10 bytes versus 1024 bytes) - important for preventing buffer overflow / underflow attacks.
- The range (e.g., 0 to 100) - important for preventing business logic issues, as well as integer overflows vulnerabilities.
- The data format (e.g., 123-45-6789 for a SSN, 12/3/45 for a date, and so on) - important for preventing all sorts of issues, including injection attacks.
- The data type (e.g., string versus integer) - also important for preventing all sorts of vulnerabilities.

Input validation strategies are primarily divided into either a whitelist approach, a blacklist approach, or sanitization. The order of preference for using these strategies is the same as the order provided. First, whitelist the input based on an approved set of valid inputs. This is the preferred security method. If a whitelist is impractical, blacklist the data that is considered invalid by constructing a list of known bad data. Keep in mind that a complete list all known bad data inputs can rarely be predetermined. Sanitization is not recommended in general, as it typically introduces more problems than it resolves. Bad data should result in an exception, not an attempt by the program to infer the intent of the user.

Insecure Direct Object Reference

Description

An insecure direct object reference vulnerability occurs when a developer exposes an internal object to the end user such as a database record, internal file, or key. Without the presence of an access control mechanism or other authentication scheme, an attacker could access the reference to obtain sensitive data.

Example Attack

As an example of an application vulnerable to an insecure direct object reference vulnerability, consider the following request:

```
http://example.com/app/accountInfo?acct=1
```

Figure 1 - An object reference

In the above request, the user that corresponds to account number one has requested information about their account through the `accountInfo` action. If user 1 were malicious, or simply very curious, she may attempt to access information belonging to another user by incrementing the integer value supplied to the `accountInfo` action, as seen below:

```
http://example.com/app/accountInfo?acct=2
```

Figure 2 - An insecure object reference

If the above request resulted in confidential information belonging to `user_2`'s account, then this would be a classic example of an insecure direct object reference vulnerability.

Vulnerability Identification

The following section will address common insecure direct object reference vulnerability patterns in ASP.NET, PHP, and Java applications.

SQL database querying can be vulnerable to insecure direct object reference attacks when a database key is displayed to the end user. If a malicious user were to modify this key value, access to otherwise privileged information could be granted.

```
SqlConnection con = new SqlConnection( ConnectionString );
con.Open();
SqlCommand query = new SqlCommand( "SELECT note_date, note_title, note_content FROM
user_data WHERE id = @note_id", con );
query.Parameters.Add( "@note_id", Request.QueryString["note_id"] );
SqlDataReader sqlReader = query.ExecuteReader();
if( sqlReader.Read() ) {
	// Display user data
}
```

Figure 3 - Vulnerable ASP.NET database query

The above code takes a user provided `note_id` value and looks up the corresponding data for that ID. If a user were to change their `note_id` from `100` to `101` the note data would be retrieved even if the user should not have access to it, as the request is never checked to ensure the user is allowed to access that data. In this example, the insecure reference is the database key corresponding to note objects.

Java

Scripts that allow users to specify a file path for reading file system content can be vulnerable to insecure direct object reference attacks. If a user were to manipulate the path, it may result in unintended files being accessed. The following code example takes the filename inputted by the user, and outputs the file contents in the response:

```
String filename = req.getParameter( "file" );

OutputStream output = resp.getOutputStream();
FileInputStream file = new FileInputStream( filename );
byte[] output_buffer = new byte[4096];
int len;

while ( ( len = file.read( output_buffer ) ) > 0 ) {
       output.write( output_buffer, 0, len );
}

file.close();
```

Figure 4 - File system insecure direct object reference in Java

As seen in the example above, no input validation is performed on the `filename` parameter. A malicious user may be able to abuse the lack of input validation and direct file reference to traverse directories and access other system files. For example, on a Linux operating system, an attacker may be able to access the system password file by submitting the following value:

```
../../../../../../etc/passwd
```

While this vulnerability could be remediated by performing stricter input validation on the `filename` parameter, it could be more cleanly resolved by preventing end users from directly referencing filesystem objects.

PHP

Due to the structure of SQL databases, many developers will directly expose database primary key values to end users via HTML forms. However, this can be dangerous if the database query is not protected with proper access controls, as a malicious user could simply modify the database primary key value to access privileged information. For example, consider the following code, and assume that `user_id` is a sequential integer representing the primary key for table `secret_user_data`:

```
 <?php
$con = $dbConnection->prepare('SELECT * FROM secret_user_data WHERE user_id = ? LIMIT
1');
$con->bind_param('i', $_GET['user_id']);
$con->execute();
```

Figure 5 - Database insecure direct object reference in PHP

While the database query above is protected from SQL injection attacks, it is still vulnerable to

Insecure Direct Object Reference

insecure direct object reference because an end user can modify the `user_id` value to access sensitive information belonging to other users.

To check for insecure direct object reference vulnerabilities, set up multiple accounts on the target application with varying levels of permissions. Identify objects that are shared between users with similar permissions (i.e. all admins should be able to access the administrative console), as well as objects that should be shared with no one (i.e. a user's personal messages, or internal server scripts). Attempt to access these objects in violation of the expected behavior of the application. If any unexpected access is possible, the application is likely susceptible to direct object reference vulnerabilities.

Particular attention should be paid to any request that submits a parameter value that appears sequential, or easily enumerated. Integers in particularly are a concern, as well as alphanumeric strings that appear to follow a pattern.

White Box Code Review

When reviewing an application's source for insecure direct object reference vulnerabilities, examine how the application services each request. Each request should be inspected to ensure that the user requesting the object has sufficient permissions to view the requested object. If the user requesting the object does not have permission to view the requested object, the request should be denied. Ideally, the request should be denied in such a manner that a user would not be able to determine an object that does not exist from an object to that the user does not have access.

Remediation and Mitigation Techniques

The root cause of insecure direct object reference vulnerabilities is the improper design or implementation of access controls. The assessment team recommends the following steps to address this vulnerability:

- Implement an authorization mechanism that requires strong authentication, enforces secure authorization based on strict access control lists, and properly logs all access to protected resources.
- Segregate the publicly accessible data from the data that requires authentication on the Web server directory structure.
- Properly implement session management and authorization controls on each resource to prevent unauthorized access.
- Redesign application components that rely on easily predictable file paths and avoid passing references to these resources directly to the client. Instead, proxy all requests for sensitive or protected information through the application using unique identifiers.

Access Reference Map

Mitigation of insecure direct object references can be accomplished by replacing easily predictable object references, such as database table primary keys, with securely generated random values. The following Java and PHP code examples illustrate how this can be accomplished through Access Reference Maps as implemented in the OWASP ESAPI library.

Insecure Direct Object Reference

Java

In the following Java example, the OWASP ESAPI library `AccessReferenceMap` interface is used to abstract direct references to filesystem objects from end users:

```
Set fileSet = new HashSet();
 fileSet.addAll(...); // add direct references (e.g. File objects)
 AccessReferenceMap map = new AccessReferenceMap( fileSet );
 // store the map somewhere safe - like the session!
 String indRef = map.getIndirectReference( file1 );
 String href = "http://www.aspectsecurity.com/esapi?file=" + indRef );
 ...
 // if the indirect reference doesn't exist, it's likely an attack
 // getDirectReference throws an AccessControlException
 // you should handle as appropriate
 String indref = request.getParameter( "file" );
 File file = (File)map.getDirectReference( indref );
```

Figure 6 - Access Reference Map implementation using OWASP API

`AccessReferenceMap` (the interface will need to be implemented by developers – above is illustrative only) accepts a `HashSet` object defining the list of direct objects to be abstracted. Indirect references to those objects are then returned with a call to the `getIndirectObjectReference` method. In this example, the map of indirect references is stored in the user session for subsequent use. Use of the `AccessReferenceMap` replaces direct references such as this:

```
http://aspectsecurity.com/esapi?file=/var/www/images/user_uploaded_image_0012.jpg
```

With difficult to guess values such as this:

```
http://aspectsecurity.com/esapi?file=2844b5dee656c00396c83ef4010bee326c0039
```

Even if an attacker found a way to guess the indirect object reference values (highly unlikely provided the access map implementation is secure), or somehow accessed a list of those values through another vulnerability, it could only result in access to files present in the map, and not in access to arbitrary files.

PHP

The following example illustrates a simplified access reference map in PHP that utilizes the `$_SESSION` array to store data:

```
<?php
session_start();

/*
 *  Turns a direct reference into an indirect reference by storing the data in the
$_SESSION array at index $random_value
 */
function direct2indirect( $value ) {
    // Append a __ARM string to front of random value so an attacker can't enumerate
other $_SESSION values
    $random_value = "__ARM".bin2hex( openssl_random_pseudo_bytes(15) );
```

Insecure Direct Object Reference

```
    $_SESSION[ $random_value ] = $value;

    return $random_value;
}

/*
 *  Turns an indirect reference back into a direct reference by retrieving it from
the $_SESSION array
 */
function indirect2direct( $indirect_string ) {
    // Check if reference exists in $_SESSION and that reference begins with __ARM
    if ( array_key_exists( $indirect_string, $_SESSION ) && substr( $indirect_string,
0, 5 ) === "__ARM" ) {
        return $_SESSION[ $indirect_string ];
    } else {
        return false;
    }
}

$direct_file_reference = "/var/www/files/userfile_10_15_13.txt";

$indirect_file_reference = direct2indirect( $direct_file_reference );
$retrieved_file_reference = indirect2direct( $indirect_file_reference );

echo "'".$direct_file_reference."' maps to '".$indirect_file_reference."'\n";
```

Figure 7 - Simple PHP access reference map implementation

The above example turns any direct references such as `/var/www/files/userfile_10_15_13.txt` into random token values such as `__ARMcf1a2bee48dd1a6601259ef5cca3d4`. The user is served this random token value instead of the direct reference. Any modifications to the ID will result in an invalid reference and a file will not be retrieved. This prevents the end user from tampering with the reference, thereby thwarting any attempts to read internal system files. It is also important to note that when implementing an access map one should always use securely generated random values such as the ones used in this example. If the token is predictable, an attacker may attempt to calculate the token and use to it access files they do not own. For a more persistent access reference map implementation, use of a database to store key value pairs is also a good option.

Additional Resources

Top 10 2010-A4-Insecure Direct Object References – OWASP
- https://www.owasp.org/index.php/Top_10_2010-A4-Insecure_Direct_Object_References

Troy Hunt: OWASP Top 10 for .NET developers part 4: Insecure direct object reference
- http://www.troyhunt.com/2010/09/owasp-top-10-for-net-developers-part-4.html

Zend_Acl - Zend Framework
- http://framework.zend.com/manual/1.5/en/zend.acl.html

Apache Shiro | Java Security Framework
- https://shiro.apache.org/authorization.html

Insecure Direct Object Reference

Role-Based Authorization – Microsoft Developer Network

- http://www.asp.net/web-forms/tutorials/security/roles/role-based-authorization-cs

Security Misconfiguration

Description

Security misconfiguration vulnerabilities occur when configuration settings in otherwise secure software cause unintended effects, which can range from information disclosure to remote code execution in the execution context of a privileged user account. The issues discussed here are specific to misconfigurations that arise in fully patched software. To learn about issues due to unpatched software, see the *Outdated Software* section.

Example Attacks

ASP.NET – Microsoft IIS

IIS contains many components including an HTTP server, ASP.NET interpreter, FTP server, terminal server, mail server, a web management console and more. In Windows Server 2008, for example, enable or disable IIS components by opening the "Control Panel," clicking "Programs and Features," and clicking "Turn Windows Features on or off." Examine the components enabled for IIS and its satellite services. A screenshot displaying common IIS components is given below:

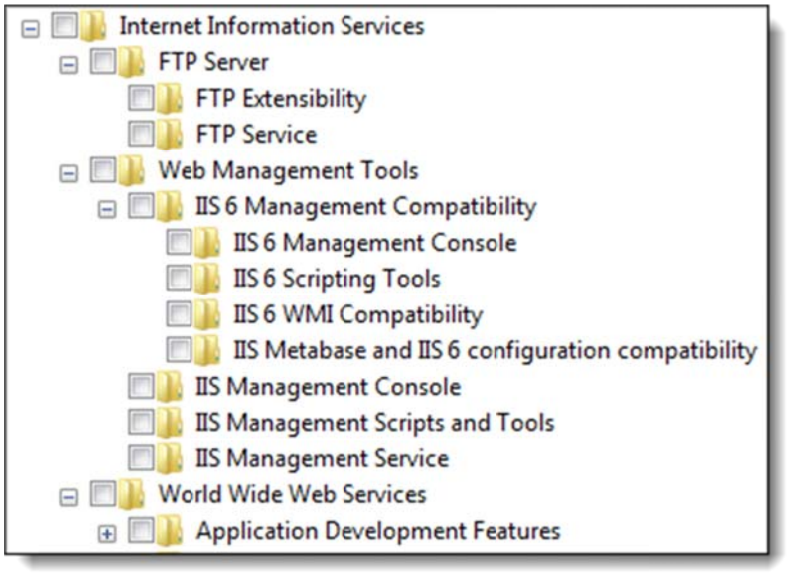


Figure 8 - Partial list of Microsoft IIS Components

Each enabled service should be investigated and if unnecessary, disabled, as each enabled component represents a unique attack surface with its own security concerns and configuration issues. Disabling unnecessary services reduces the overall attack surface of the corresponding system, and simplifies server management.

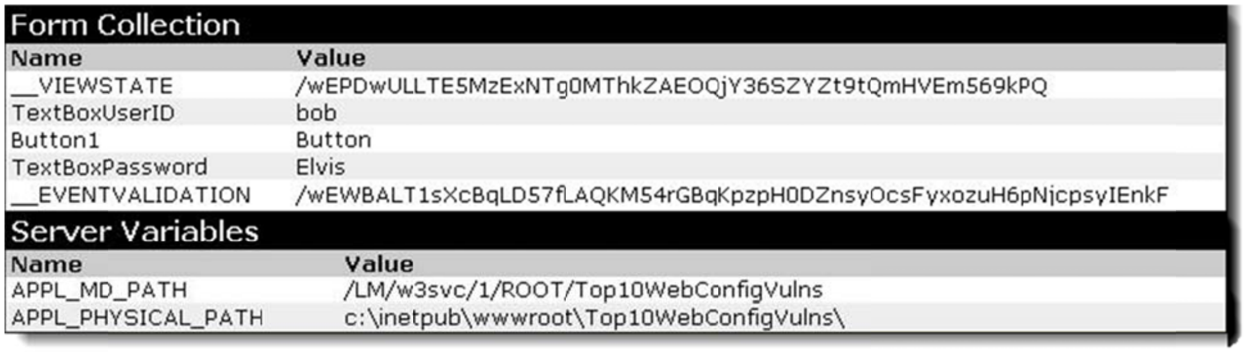
In the core ASP.NET component, there is a setting the deployment team should be aware of called `trace`, which enables a special page located in the web root named `trace.axd`. This page is used for debugging Web requests; however, if it remains enabled, it can leak highly sensitive information that may be useful to an attacker. The following `Web.Config` file snippet is an example of a server configuration enabling ASP.NET trace functionality:

Security Misconfiguration

```
<!—Web.Config Configuration File -->
<configuration>
    <system.web>
        <trace enabled="true"/>
    </system.web>
</configuration>
```

Figure 9 - Insecure `Web.Config` file example

Below is a screenshot showing the type of information typically found in `trace.axd`:



Form Collection

Name	Value
__VIEWSTATE	/wEPDwULLTE5MzExNTg0MThkZAEOQjY36SZYZt9tQmHVEm569kPQ
TextBoxUserID	bob
Button1	Button
TextBoxPassword	Elvis
__EVENTVALIDATION	/wEWBALT1sXcBqLD57fLAQKM54rGBqKpzpH0DZnsyOcsFyxozuH6pNjcpsyIEnkF

Server Variables

Name	Value
APPL_MD_PATH	/LM/w3svc/1/ROOT/Top10WebConfigVulns
APPL_PHYSICAL_PATH	c:\inetpub\wwwroot\Top10WebConfigVulns\

Figure 10 - `Trace.axd` output

`Trace.axd` leaks many details including absolute file paths, user passwords, user browser versions, and `viewstate` values. This information helps an attacker gain a greater understanding of the target application server, and can be used to directly compromise user accounts. Trace functionality should be disabled in production environments, as shown below:

```
<!—Web.Config Configuration File -->
<configuration>
    <system.web>
        <trace enabled="false"/>
    </system.web>
</configuration>
```

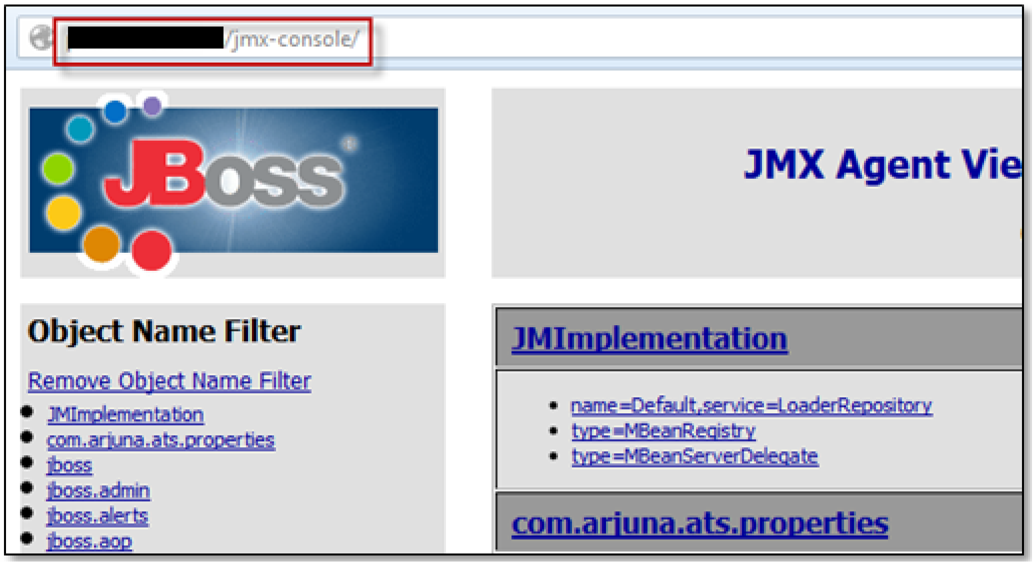
Figure 11 - Secured `Web.Config` file example

Securing Microsoft IIS is a large task with many different factors to consider. To learn more about securing IIS, refer to the *Additional Resources* section.

JBoss

JBoss is a popular choice for managing and deploying Java Web applications. One known issue with certain versions of JBoss is that it installs with a publicly available administration console. When insecurely configured, unauthorized users may be able to access the admin console either by supplying the default password, or no password at all. Access to the administration console gives users full access to the functionality of JBoss, including the ability to read database credentials, and execute arbitrary code in the context of the Web server. The JBoss Administration Console is shown below:

Security Misconfiguration



[redacted]/jmx-console/

JMX Agent Vie

Object Name Filter

Remove Object Name Filter

- JMImplementation
- com.arjuna.ats.properties
- jboss
- jboss.admin
- jboss.alerts
- jboss.aop

JMImplementation

- name=Default,service=LoaderRepository
- type=MBeanRegistry
- type=MBeanServerDelegate

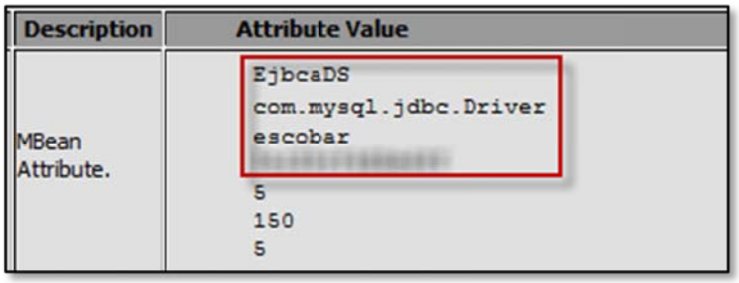
com.arjuna.ats.properties

Figure 12 - The exposed JBoss Administration console

Visiting the following JMX console URL reveals sensitive database credentials in plaintext:

```
/jmx-console/HtmlAdaptor?action=inspectMBean&name=jboss.management.local:J2EEServer=L
ocal,j2eeType=ServiceModule,name=ejbca-ds.xml
```

An example screenshot of the above page is shown below:



Description	Attribute Value
MBean Attribute.	EjbcaDS com.mysql.jdbc.Driver escobar [redacted] 5 150 5

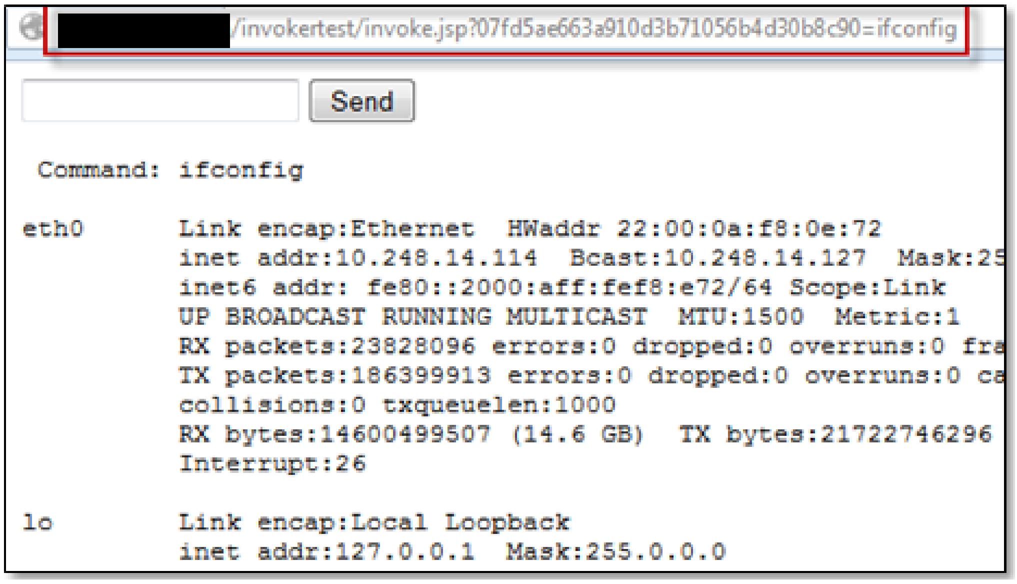
Figure 13 - Screenshot of the compromised database credentials

In addition to sensitive data disclosure, there is also a resource called `DeploymentFileRepository` that can be abused to upload and deploy arbitrary web application archive (WAR) files. That functionality is referred to as the `DeploymentFileRepository`, and is located at:

```
/jmx-console/HtmlAdaptor?action=inspectMBean&name=jboss.admin:service=
DeploymentFileRepository
```

The following screenshot illustrates command execution via a malicious WAR file that was uploaded through an exposed `DeploymentFileRepository` page:

Security Misconfiguration



[redacted]/invokertest/invoke.jsp?07fd5ae663a910d3b71056b4d30b8c90=ifconfig

```
 Command: ifconfig

eth0      Link encap:Ethernet  HWaddr 22:00:0a:f8:0e:72
          inet addr:10.248.14.114  Bcast:10.248.14.127  Mask:25
          inet6 addr: fe80::2000:aff:fef8:e72/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:23828096 errors:0 dropped:0 overruns:0 fra
          TX packets:186399913 errors:0 dropped:0 overruns:0 ca
          collisions:0 txqueuelen:1000
          RX bytes:14600499507 (14.6 GB)  TX bytes:21722746296
          Interrupt:26

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
```

Figure 14 - Java WAR shell executing the `ipconfig` command

In order to reduce the attack surface of JBoss application servers, the JMX Console should be disabled unless a valid business reason for enabling it exists. If it must be enabled, access to the console should be restricted to trusted networks, and require the use of strong passwords. For more information regarding how to secure JBoss application servers, please refer to the *How to Secure the JBoss Server* link found in the *Additional Resources* section below.

PHP

PHP is usually deployed as a dynamic module in a web application server, such as Apache or IIS. PHP itself has its own configuration file, which is typically called `php.ini` or `php.cfg`. One common PHP misconfiguration is to leave the `expose_php` option set to ‘`on`’, as shown in the example below:

```
expose_php = 'on'
```

Figure 15 - Insecure setting for `expose_php`

Leaving this default setting as is will result in a `X-Powered-By` HTTP header that discloses the exact version of PHP, including underlying server operating system, in all application server responses to PHP resources:

```
HTTP/1.1 200 OK
…
X-Powered-By: PHP/5.3.10-1ubuntu1
…
```

Figure 16 - Version information exposed

In this case, not only is the specific version number of PHP exposed, but since the version has been modified and tagged by the operating system vendor, an attacker can guess that this server is probably running a version of Ubuntu that has not been updated since March 2012. If this computer is running Samba, it is vulnerable to CVE-2012-1182, which is a remote code execution vulnerability that, if successfully exploited, will result in root access to the compromised system. This setting, therefore, not only reveals information about the PHP interpreter, but also provides information relating to the corresponding system update schedule. To be safe, the PHP interpreter should be configured to not disclose this information:

```
expose_php = 'off'
```

Figure 17 - Secure setting for `expose_php`

Another dangerous option in PHP is `register_globals`. Below is an excerpt from a PHP configuration file that improperly enables `register_globals`:

```
register_globals = 'on'
```

Figure 18 - Insecure setting for `register_globals`

When `register_globals` is set to on, variable values may be altered directly in a request, even if the developer did not intent do expose those variables. Consider the following PHP code excerpt:

```
//register_globals = on;
if (authenticated_user()) {
    $authorized = true;
}
if ($authorized) {
    include "/highly/sensitive/data.php";
}
```

Figure 19 - Dangerous use of a global variable

In the code excerpt above, the controller responds to a request for a sensitive page (`data.php`) only if a variable (`$authorized`) is true. Therefore, in a system where `register_globals` is enabled, the following request can circumvent the application's protection mechanisms:

```
GET /highly/sensitive/data.php?authorized=1 HTTP/1.1
…
```

Figure 20 - Bypassing authentication by abusing `register_globals`

The `register_globals` setting in PHP should not be enabled in production software. Attackers may be able to guess internal variables and manually set them in HTTP requests to circumvent server protections. By default recent versions of PHP have disabled `register_globals`. Developers should ensure this setting has not been altered in their environments:

```
register_globals = 'off'
```

Figure 21 - Secure setting for `register_globals`

A thorough discussion on PHP and its various security features can be found on the OWASP website. A link to the appropriate article is given in the *Additional Resources* section.

Security Misconfiguration

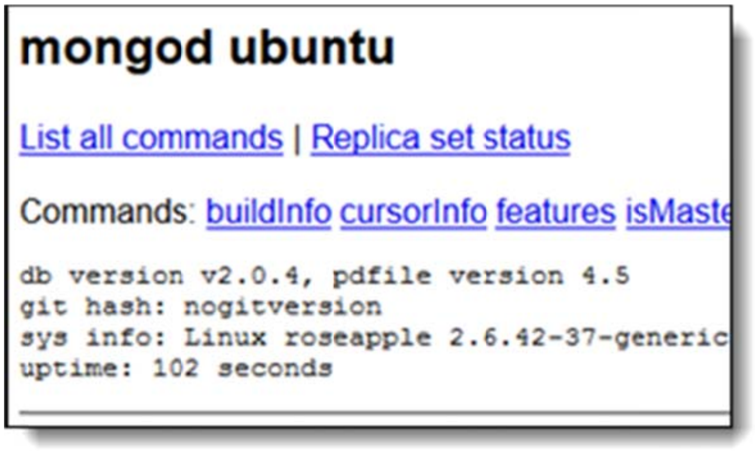
MongoDB

The popular "NoSQL" database, MongoDB, does not implement an authentication scheme by default. Attackers who discover exposed instances of MongoDB can simply connect to the database with administrative privledges. Exposed MongoDB databases can be easily identified using a port scanning utility such as NMap:

```
Starting Nmap 5.21 ( http://nmap.org ) at 2014-04-14 16:57 PDT
Nmap scan report for mandatorys-box (192.168.19.129)
Host is up (0.00022s latency).
Not shown: 54992 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
53/tcp    open  domain
80/tcp    open  http
631/tcp   open  ipp
3306/tcp  open  mysql
8080/tcp  open  http-proxy
27017/tcp open  unknown
28017/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 0.81 seconds
```

Figure 22 - Nmap discovering MongoDB

The above Nmap scan results show a service listening on port 28017, and connecting to that port with a normal Web browser revealed it to be an exposed MongoDB control panel:


mongod ubuntu

List all commands | Replica set status

Commands: buildInfo cursorInfo features isMaste

```
db version v2.0.4, pdfile version 4.5
git hash: nogitversion
sys info: Linux roseapple 2.6.42-37-generic
uptime: 102 seconds
```

Figure 23 - An exposed MongoDB control panel

The first step to secure the MongoDB server control console is to enable user authorization. This can be accomplished by editing the `mongodb.conf` file and changing the `security.authorization` attribute to `enabled`:

```
security.authorization = enabled
```

Figure 24 - Enabling MongoDB user authorization

Security Misconfiguration

Next, an administrator account must be created by connecting to the MongoDB server locally and issuing the following commands:

```
use admin
db.createUser(
  {
    user: "siteUserAdmin",
    pwd: "Use_A_V3rY!Str0nG_passWerD***",
    roles:
    [
      {
        role: "userAdminAnyDatabase",
        db: "admin"
      }
    ]
  }
```

Figure 25 - Command to add an admin user

By securing the MongoDB in the above manner, attackers will no longer be able to access the control panel without first supplying a valid username and password. Further controls to restrict access to the control panel are highly recommended, such as restricting access to the control panel to just trusted networking segments, and to actively monitor for failed authentication attempts. A complete reference on securing MongoDB is given in the *Additional Resources* section, below.

Additional Resources

Microsoft - IIS Hardening Checklist
- http://technet.microsoft.com/en-us/library/cc179961.aspx

DevX.com - Top 10 Security Vulnerabilities in .NET Configuration Files
- http://www.devx.com/dotnet/Article/32493

Microsoft - Enabling Only Essential IIS Components and Services
- http://technet.microsoft.com/en-us/library/cc783048%28v=ws.10%29.aspx

Center for Internet Security – Security Configuration Benchmark – IIS 7.0/7.5
- https://benchmarks.cisecurity.org/tools2/iis/CIS_Microsoft_IIS7_Benchmark_v1.2.0.pdf

RedTeam Pentesting - JBoss Remote Code Execution
- http://www.redteam-pentesting.de/en/publications/jboss/-bridging-the-gap-between-the-enterprise-and-you-or-whos-the-jboss-now

JBoss.org - Securing the JBoss JMX Console and Web Console
- https://community.jboss.org/wiki/securetheJmxConsole?_sscc=t

JBoss.org - How to Secure the JBoss Server
- https://docs.jboss.org/jbossas/docs/Server_Configuration_Guide/4/html/Security_on_JBoss-How_to_Secure_the_JBoss_Server.html

PHP.net - PHP Register Globals

Security Misconfiguration

- http://www.php.net/manual/en/security.globals.php

OWASP - PHP Security Cheatsheet
- https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet

MongoDB.org - MongoDB Security Guide
- http://docs.mongodb.org/manual/MongoDB-security-guide.pdf

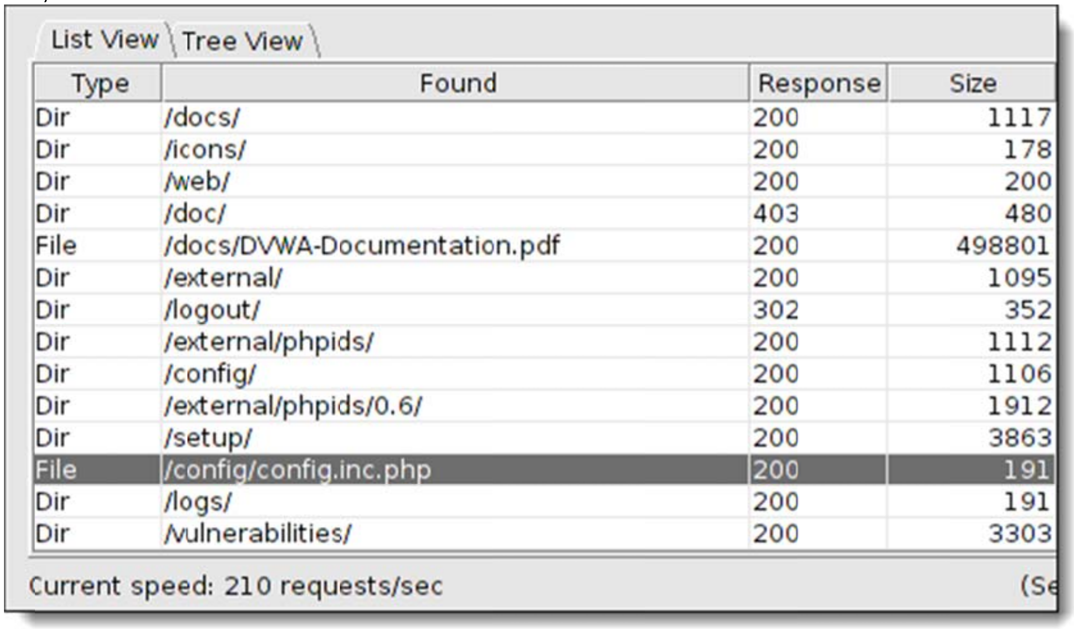
Sensitive Information Exposure

Description

Sensitive information exposure is a broad category of vulnerabilities, which can occur in a variety of ways. Publicly exposed debugging information (e.g. `phpinfo` or stack traces), and unprotected files placed within the Web server root directory are among the most common ways the vulnerability manifests within an application. The impact of this vulnerability can range anywhere from low to high risk depending on the nature of the information. However, even low risk information such as a version number or variable name can allow attackers to perform more targeted attacks.

Vulnerability Identification

The most direct approach for an attacker to identify sensitive information exposure vulnerabilities is to first spider the website, following every available link on the website. Next the attacker will use a file discovery tool such as DirBuster to look for hidden files are not linked to, but still accessible to users.



Type	Found	Response	Size
Dir	/docs/	200	1117
Dir	/icons/	200	178
Dir	/web/	200	200
Dir	/doc/	403	480
File	/docs/DVWA-Documentation.pdf	200	498801
Dir	/external/	200	1095
Dir	/logout/	302	352
Dir	/external/phpids/	200	1112
Dir	/config/	200	1106
Dir	/external/phpids/0.6/	200	1912
Dir	/setup/	200	3863
File	/config/config.inc.php	200	191
Dir	/logs/	200	191
Dir	/vulnerabilities/	200	3303

```
Current speed: 210 requests/sec                                    (Se
```

Figure 26 - DirBuster discovering hidden files

DirBuster uses common file names to discover files left on the server but not referenced anywhere in the application. These files will often contain sensitive information.

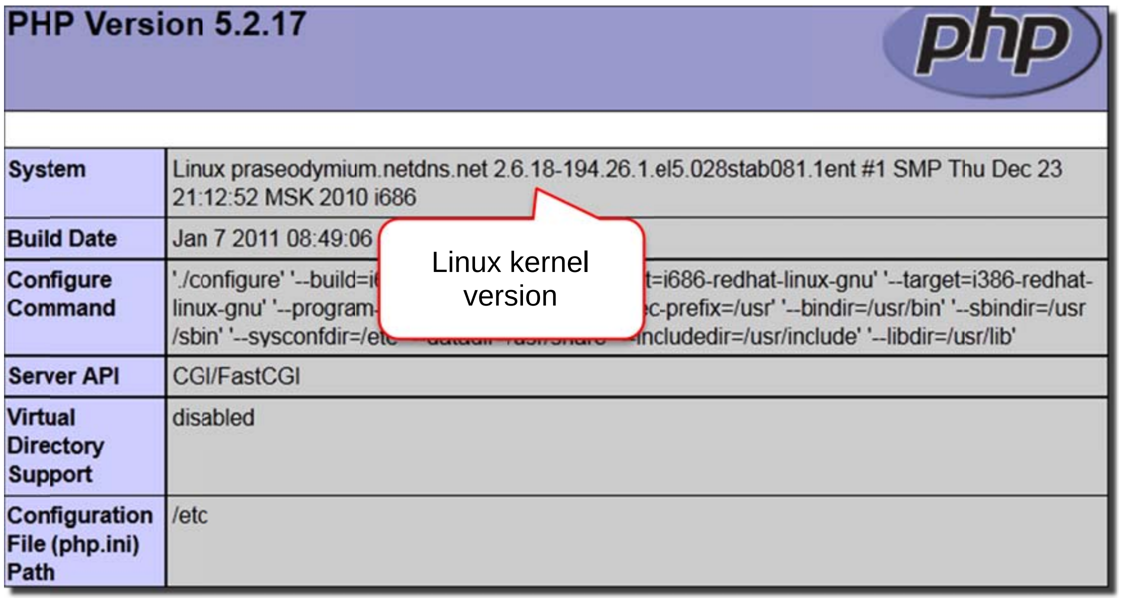
Every file that is accessible to users should be accessible for a purpose. Do not leave unnecessary files in the production environment. Developers should pass sensitive messages between each other via protected communication channels, rather than in an ad-hoc fashion in the source code itself. Ensure that all PII and other sensitive information are protected according to the applicable standards and guidelines. Data at rest (stored in a database, files, etc.) should be encrypted with strong passwords. Encrypt data in motion so that man-in-the-

Sensitive Information Exposure

middle attacks will not disclose sensitive information.

Example Attack

Using DirBuster an attacker discovers a `phpinfo` page inadvertently left on a production server:



PHP Version 5.2.17

System	Linux praseodymium.netdns.net 2.6.18-194.26.1.el5.028stab081.1ent #1 SMP Thu Dec 23 21:12:52 MSK 2010 i686
Build Date	Jan 7 2011 08:49:06
Configure Command	'./configure' '--build=i t=i686-redhat-linux-gnu' '--target=i386-redhat-linux-gnu' '--program- c-prefix=/usr' '--bindir=/usr/bin' '--sbindir=/usr/sbin' '--sysconfdir=/etc includedir=/usr/include' '--libdir=/usr/lib'
Server API	CGI/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc

Figure 27 - Publically exposed `phpinfo` page

In this example the page reveals to the attacker, the version of PHP used, the operating system, the kernel version, and other configuration information. However, the critical piece of information obtained by the attacker is the server's Linux kernel version. A quick search reveals several privilege escalation vulnerabilities existing within this build of the Linux kernel; thus if an attacker can gain code execution in the context of an unprivileged user (e.g. the Web server `www-data`), the kernel vulnerability can then be triggered as a second exploit to immediately gain root privileges on the system. The attacker can then leverage these super user privileges to pivot into the organization's internal network and compromise additional targets. Knowing this, it would behoove the attacker to spend additional time attacking this specific system, since the payoff will be greater.

Remediation and Mitigation Techniques

Unlike most vulnerabilities, sensitive information exposure does not have a standard remediation technique. Each instance of this vulnerability will have a different impact, a different scope, and a different remediation strategy. The first step in prevention is to audit all systems and examine how sensitive information is stored, transmitted, and used. Sensitive data should be used as seldom as possible, it should be encrypted at rest, and it should be deleted at the earliest opportunity. Restrict users that have access to sensitive information as much as possible. Disable caching for pages containing sensitive information.

Sensitive Information Exposure

Additional Resources

OWASP - Sensitive Data Exposure

- https://www.owasp.org/index.php/Top_10_2013-A6-Sensitive_Data_Exposure

OWASP - DirBuster

- https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project

Official PCI Security Standards

- https://www.pcisecuritystandards.org/

HIPPA Privacy Standards

- http://www.hhs.gov/ocr/privacy/hipaa/administrative/privacyrule/

Electronic Code of Federal Regulations

- http://www.ecfr.gov/cgi-bin/text-idx?c=ecfr&sid=11975031b82001bed902b3e73f33e604&rgn=div5&view=text&node=34:1.1.1.1.33&idno=34

Kaspersky - Best Practice Guide to Encryption

- https://shiro.apache.org/authorization.html

Outdated Software

Description

An outdated software vulnerability occurs when publically known security vulnerabilities are not patched in a timely fashion. Outdated software vulnerabilities are among the most dangerous because tools to exploit publicly known vulnerabilities are widely available, and require very little skill to use. Furthermore, sophisticated malware is often created to automatically search for and exploit vulnerable systems.

Example Attack

In April 2014, Neel Mehta of Google's security team discovered a critical security vulnerability (CVE-2014-0160) in the OpenSSL library that could be exploited to read privileged server memory from vulnerable SSL enabled servers. Because of this vulnerability, attackers could potentially gain access to SSL private keys, Web application passwords, and other sensitive information. The impact of the vulnerability was so severe that it was subsequently named “Heartbleed.” Several tools were released to exploit Heartbleed soon after the vulnerability was disclosed to the public. The following example shows the output received from one such utility:

```
# python heartbleed.py example.com
Connecting to example.com:443
Sending Client Hello...
Waiting for Server Hello...
 ... received message: type = 22, ver = 0302, length = 58
 ... received message: type = 22, ver = 0302, length = 2441
 ... received message: type = 22, ver = 0302, length = 525
 ... received message: type = 22, ver = 0302, length = 4
Sending heartbeat request...
 ... received message: type = 24, ver = 0302, length = 16384
Received heartbeat response:
…
  00e0: 2B 78 6D 6C 2C 61 70 70 6C 69 63 61 74 69 6F 6E  +xml,application
  00f0: 2F 78 6D 6C 3B 69 6D 61 67 65 2F 70 6E 67 2C 69  /xml;image/png,i
  0100: 6D 61 67 65 2F 6A 70 65 67 2C 69 6D 61 67 65 2F  mage/jpeg,image/
  0110: 2A 3B 71 3D 30 2E 39 2C 2A 2F 2A 3B 71 3D 30 2E  *;q=0.9,*/*;q=0.
  0120: 38 0D 0A 41 63 63 65 70 74 2D 4C 61 6E 67 75 61  8..Accept-Langua
  0130: 67 65 3A 20 65 6E 2D 75 73 3B 71 3D 30 2E 37 2C  ge: en-us;q=0.7,
  0140: 20 65 6E 3B 71 3D 30 2E 33 0D 0A 43 6F 6E 6E 65   en;q=0.3..Conne
  0150: 63 74 69 6F 6E 3A 20 63 6C 6F 73 65 0D 0A 43 6F  ction: close..Co
  0160: 6F 6B 69 65 3A 20 62 65 37 36 38 63 31 37 30 62  okie: be768c170b
  0170: 30 30 35 62 32 39 61 37 39 33 32 36 38 35 37 39  005b29a793268579
  0180: 37 31 33 37 33 64 3D 38 61 38 33 39 30 66 63 63  71373d=8a8390fcc
  0190: 34 61 30 62 38 31 31 30 31 34 36 34 62 64 39 33  4a0b81101464bd93
  01a0: 65 31 64 34 66 61 64 0D 0A 52 65 66 65 72 65 72  e1d4fad..Referer
  01d0: 6C 6F 67 73 2F 6D 79 5F 70 61 67 65 2F 61 64 64  logs/my_page/add
  01e0: 2F 0D 0A 0D 0A 7D 87 78 FD 1A A6 85 69 C5 92 6C  /....}.x....i..l
  01f0: CC AC 93 34 B6 E3 D5 93 DF 06 06 06 06 06 06 06  ...4............
  0200: 61 6D 65 5C 22 3B 4E 3B 73 3A 38 3A 5C 22 75 73  ame\";N;s:8:\"us
  0210: 65 72 6E 61 6D 65 5C 22 3B 4E 3B 73 3A 35 3A 5C  ername\";N;s:5:\
  0220: 22 65 6D 61 69 6C 5C 22 3B 4E 3B 73 3A 38 3A 5C  "email\";N;s:8:\
  0230: 22 70 61 73 73 77 6F 72 64 5C 22 3B 4E 3B 73 3A  "password\";N;s:
  0240: 31 34 3A 5C 22 70 61 73 73 77 6F 72 64 5F 63 6C  14:\"password_cl
  0250: 65 61 72 5C 22 3B 73 3A 30 3A 5C 22 5C 22 3B 73  ear\";s:0:\"\";s
  0260: 3A 35 3A 5C 22 62 6C 6F 63 6B 5C 22 3B 4E 3B 73  :5:\"block\";N;s
  0270: 3A 39 3A 5C 22 73 65 6E 64 45 6D 61 69 6C 5C 22  :9:\"sendEmail\"
```

Outdated Software

```
  0280: 3B 69 3A 30 3B 73 3A 31 32 3A 5C 22 72 65 67 69  ;i:0;s:12:\"regi
…
WARNING: server returned more data than it should - server is vulnerable!
```

Figure 28 - A server leaking a session cookie due to Heartbleed

In the above example, the Heartbleed vulnerability has been successfully exploited to extract 64KB of server-side memory in a single request, resulting in compromise of an authenticated user session ID. The session ID can then be used to obtain unauthorized access to the corresponding application.

The Heartbleed vulnerability was so obscure that it took professional researchers two years to discover it, and it resulted in the loss of billions of dollars due to the costs required to address the vulnerability. Worse yet, many embedded systems that were not designed with software updates in mind may never be patched against the Heartbleed vulnerability.

Remediation and Mitigation Techniques

The primary remediation strategy for resolving outdated software vulnerabilities is the timely patching of vulnerable software, and the creation of corporate policies and tools to ensure that patching is actually performed.

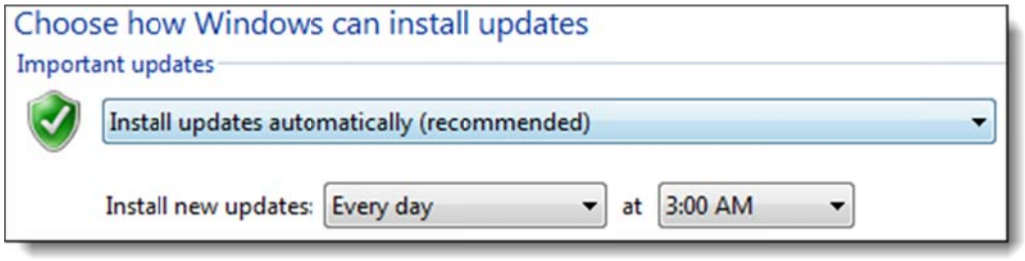
Proactive Defense and Automatic Patches

The most important step in remediating these types of vulnerabilities is to have vigilant IT staff that is able to react to emerging threats. This includes having staff ready to update and protect mission-critical systems at a moment's notice when critical vulnerabilities are announced.

Subscription to a vulnerability mailing list such as the one available through Mitre.org will also help staff stay informed of emerging threats that may include obscure software packages, which are overlooked by the normal maintenance cycle.

Automatic updates are used to keep machines up-to-date with on the most important software patches, and particularly software security patches. Ensuring that systems receive periodic software patches is critical to mitigating the risk of outdated software.

In Windows, software updates are configured in the Windows Update settings panel pictured below:


Choose how Windows can install updates

Important updates

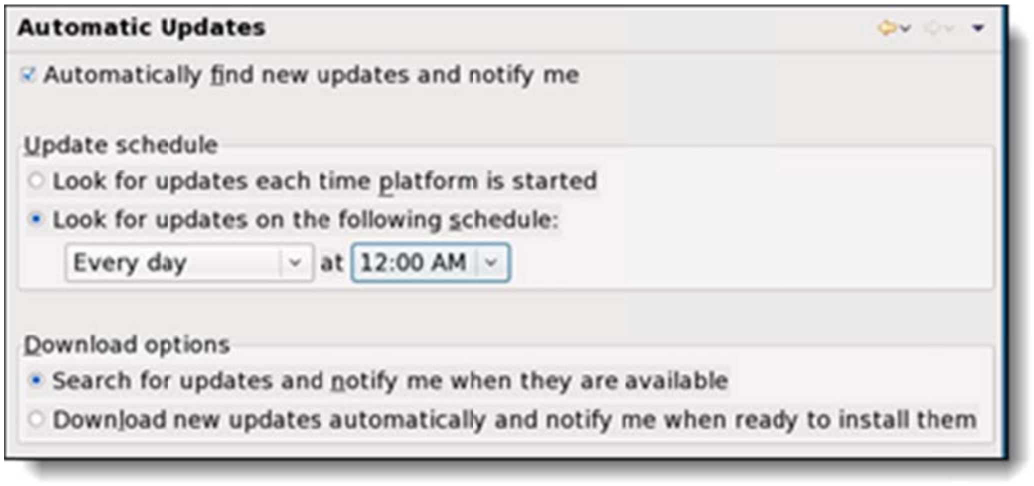
Install updates automatically (recommended)

Install new updates: Every day at 3:00 AM

Figure 29 - Windows Automatic Update set to install updates automatically every day

Outdated Software

RedHat Linux has a similar control panel that can be used to set an automatic security update schedule:



Automatic Updates

[✓] Automatically find new updates and notify me

Update schedule

- ○ Look for updates each time platform is started
- ● Look for updates on the following schedule:

Every day at 12:00 AM

Download options

- ● Search for updates and notify me when they are available
- ○ Download new updates automatically and notify me when ready to install them

Figure 30 - RedHat Automatic Update set to install security updates every day

Of course, enterprise IT groups will configure the above settings through domain policies, or on representative virtual images, and not on a machine-by-machine basis. In many cases, specialized vendor solutions will also be deployed to manage patch policies across disparate environments. These solutions often include system monitoring agents that report the current patch level of a system back to a centralized console in order to help organizations track corporate patch compliance, and identify risks.

In most cases, however, automatic updates will not update all software packages in your application's tool chain. It is recommended that every organization have their own software update plan to ensure that important applications are kept current with the latest patches.

Additional Resources

Heartbleed: Examining the impact

- http://www.darkreading.com/attacks-breaches/heartbleed-examining-the-impact-/d/d-id/1204330

Mitre.org: CVE Updates

- https://cve.mitre.org/cve/data_updates.html

Microsoft: How to Configure and Use Automatic Updates in Windows

- http://support.microsoft.com/kb/306525

RedHat: Security Updates

- https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/Deployment_Guide/ch-security-updates.html

Exploit Database

- http://www.exploit-db.com/

Missing Function Level Access Controls

Description

A missing function level access vulnerability occurs when an application fails to restrict access to a sensitive action. For example, a regular user should not be able to execute functions reserved for a privileged administrator. If a regular user could perform administrative functions such as deleting a user, the Web application is said to suffer from missing function level access controls. Privileged actions should not only be hidden from the user, but the user identity must be checked before sensitive actions are executed.

Example Attack

The following examples illustrate missing function level access control vulnerabilities in various languages:

ASP.NET

If functionality is just hidden instead of secured through an access control scheme, a user may be able to discover the hidden functionality, and then use it without authorization. In the following example, the code provides an option to message, add, or delete a user account:

```
Dim username = usernameInput.text;
switch( actionInput.text )
{
	case "message":
		message_user( username );
		break;
	case "add_as_friend":
		add_friend( username );
		break;
	case "delete_account":
		// Admin only, do not display to user on page!
		delete_account( username );
		break;
}
```

Figure 31 - Hidden functionality

Though the delete account option above is not displayed to non-admin users, it can be called by anyone, as no access control checks are performed. If an attacker discovered the `delete_user` action, they could perform the action despite lacking the required privileges. A positive check to ensure that the current user has permission to use the delete functionality is required. Security through obscurity is not a safe practice, and can often lead to scenarios such as this one.

Java

If an application fails to ensure that a user has proper privileges before accessing a function, a malicious attacker may enumerate a privileged action and abuse it. The following insecure code allows unauthorized users the ability to delete accounts, as well as escalate account privileges to that of an admin user:

```
if( user_logged_in() ) {
	String action = req.getParameter( "action" );
```

Outdated Software

```
        String username = req.getParameter( "username" );
        if( action == "delete" ) {
                delete_account( username );
        } else if ( action == "make_admin" ) {
                make_admin( username );
        } else {
                System.out.println( "Invalid action!" );
        }
} else {
        System.out.println( "Please log in!" );
}
```

Figure 32 - Insecure Privilege Validation in Java

The above code snippet only checks to ensure a user is logged in, but not if the user has privileges to perform actions such as `delete_account` or `make_admin`.

PHP

Access control checks must be performed at the server, and based on trusted information. If validation for accessing a function is performed using only user input, function level controls could potentially be bypassed. The following code snippet performs a database backup if the user is an administrator:

```
if( $_POST['is_admin'] == "true" ) {
        switch( $_POST['action'] ) {
                case "download_database":
                        $backup = New create_database_backup();
                        $backup->serve_download_link();
                        break;
        }
}
```

Figure 33 - Insecure Privilege Validation in PHP

The above snippet checks the value of the user submitted `is_admin` parameter, and if true, allows the database backup to occur. A malicious user could simply change the value to “true” in order to access the “protected” functionality of the application.

Vulnerability Identification

Black Box Code Review

When performing a black block code review for missing function level access it is important to test each application function to ensure it is properly validated. Each action should be tested while being unauthenticated, and authenticated with improper privileges. This will help expose any functions that are not employing proper privilege checking. If an action unexpectedly succeeds with improper privileges, then a missing function level access control has been located. For example, if the function being tested is the administrative ability to delete users, a regular user and an unauthenticated user should attempt this action. If the action succeeds and the user account is deleted, a missing function level access vulnerability has been enumerated.

White Box Code Review

Outdated Software

All privileged user actions should be properly validated to ensure that the user performing the action is both authenticated and is authorized. It is important to distinguish between authorization and authentication: authentication is the act of verifying who a user is and authorization is the process of ensuring that a user is allowed to perform some action.

For example, before a function such as `delete_user` is called, the current user permissions should be checked to ensure the user is authorized to perform that action. When implementing an access control scheme the developer should not have to manually implement access control for every function ad hoc but rather a general framework should be used. Ad hoc verification of permissions is error prone; a framework allows a much safer and organized approach to access control. For more information about implementing an access control framework, see *Remediation and Mitigation Techniques* below.

Remediation and Mitigation Techniques

The root cause of missing function level access vulnerabilities is the improper design or implementation of access controls. The assessment team recommends the following steps to address this vulnerability:

- Implement an authorization mechanism that requires strong authentication, enforces secure authorization based on strict access control lists, and properly logs all access to protected resources.
- Segregate the publicly accessible data from the data that requires authentication on the Web server directory structure. Ensure the implementation of proper session management and authorization on each resource to prevent unauthorized access.

PHP

The Zend framework contains the `Zend_Acl` module that can be used to implement an access control list (ACL) along with privilege management. An example of a Zend ACL is given below:

```php
<?php
require_once 'Zend/Acl.php';
$acl = new Zend_Acl();

require_once 'Zend/Acl/Role.php';
$acl->addRole(new Zend_Acl_Role('guest'))
    ->addRole(new Zend_Acl_Role('member'))
    ->addRole(new Zend_Acl_Role('admin'));

$parents = array('guest', 'member', 'admin');
$acl->addRole(new Zend_Acl_Role('someUser'), $parents);

require_once 'Zend/Acl/Resource.php';
$acl->add(new Zend_Acl_Resource('someResource'));

$acl->deny('guest', 'someResource');
$acl->allow('member', 'someResource');

echo $acl->isAllowed('someUser', 'someResource') ? 'allowed' : 'denied';
```

Figure 34 - Example Zend Access Control List

Outdated Software

In the above example, three roles are defined – `guest`, `member`, and `admin`. After that, a new role is created called `someUser` and the previously mentioned roles are assigned to that user. In this manner, a hierarchy of roles can be defined. When using an access control framework such as Zend, it is important to understand how conflicting permissions are resolved when inheriting from multiple roles. For example, in the above code, the `guest` role is denied permissions to `someResource`, while the `member` role is granted access. When `isAllowed` is called against `someUser` and `someResource`, the result will be `denied`, because inheritance conflicts resolve to assume the least amount of privileges. Using this system, a comprehensive permissions structure can be designed, thereby removing the need for constant ad hoc permission checks.

More information regarding Zend can be found at the *Zend_Acl - Zend Framework* link the *Additional Resources* section below.

Java

The Apache Shiro framework allows developers to implement access control schemes. An example access control in the Shiro framework is given below:

```
Subject currentUser = SecurityUtils.getSubject();

//guarantee that the current user is and administrator and
//therefore allowed to view the server logs:
currentUser.checkRole("admin");
openServerLogs();
```

Figure 35 - Example Shiro Access Control

The above snippet first gets the current user via the `SecurityUtils.getSubject()` method. It follows that by then checking if the current user has the role of `admin` and either allows the script to continue if the user has permission or blocks them if they do not. The framework allows for easy implementation of access verification and privilege checking. The following is an example from the Apache Shiro website showing the framework in action:

```
try {
    currentUser.login( token );
    //if no exception, that's it, we're done!
} catch ( UnknownAccountException uae ) {
    //username wasn't in the system, show them an error message?
} catch ( IncorrectCredentialsException ice ) {
    //password didn't match, try again?
} catch ( LockedAccountException lae ) {
    //account for that username is locked - can't login.  Show them a message?
}
    ... more types exceptions to check if you want ...
} catch ( AuthenticationException ae ) {
    //unexpected condition - error?
}
```

Figure 36 - Shiro framework for logging in a user

The framework allows the developer to implement an access control system that is manageable and not ad hoc. The framework can be used to display appropriate information to the user about access violations and prevents missing function level access issues from occurring. The snippet shows the action of authenticating the user and the different exceptions that can be thrown and handled.

For more information, see *Apache Shiro | Java Security Framework* found in the *Additional Resources* section below.

ASP.NET

Access control functionality can be implemented via forms authentication in the ASP.NET MVC framework. Before forms authentication can be implemented, it must first be enabled via the `web.config` file. The following is an example `web.config` file that redirects all anonymous requests to privileged resources to a standard login page:

```
<authentication mode="Forms">
  <forms loginUrl="~/Login" timeout="2880" />
</authentication>
```

Figure 37 - Example `web.config` configuration

Once this configuration has been implemented, a custom authentication controller should be created to handle login attempts. The following code snippet demonstrates such a controller:

```
[HttpPost]
public ActionResult Login(User model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        using (userDbEntities entities = new userDbEntities())
        {
            string username = model.username;
            string password = model.password;

            password =
FormsAuthentication.HashPasswordForStoringInConfigFile(password, "sha1");

            bool userValid = entities.Users.Any(user => user.username == username &&
user.password == password);

            if (userValid)
            {
                FormsAuthentication.SetAuthCookie(userid, false);
                if (Url.IsLocalUrl(returnUrl) && returnUrl.Length > 1 &&
returnUrl.StartsWith("/")
                    && !returnUrl.StartsWith("//") && !returnUrl.StartsWith("/\\"))
                {
                    return Redirect(returnUrl);
                }
                else
                {
                    return RedirectToAction("Index", "Home");
                }
            }
            else
            {
                ModelState.AddModelError("", "Incorrect username or password
provided, please try again.");
            }
        }
    }
    return View(model);
}

public ActionResult LogOff()
{
    FormsAuthentication.SignOut();
    return RedirectToAction("Index", "Home");
}
```

Figure 38 - Example authentication snippet

The above code checks the user-supplied username and password values to confirm they are valid. If the credentials are valid, an authentication cookie is set via the `FormsAuthenitcation.SetAuthCookie` method, and the user is redirected to the protected resource.

Custom roles can be set during authentication to allow users access only to specific resources in the Web application. To do this, a custom authentication method should be implemented by overriding the base `FormsAuthentication_OnAuthenticate` method. The following is an example custom authorization form that checks the current user roles:

```
protected void FormsAuthentication_OnAuthenticate(Object sender,
FormsAuthenticationEventArgs e)
{
    if (FormsAuthentication.CookiesSupported == true)
    {
        if (Request.Cookies[FormsAuthentication.FormsCookieName] != null)
        {
            try
            {
                string userid =
FormsAuthentication.Decrypt(Request.Cookies[FormsAuthentication.FormsCookieName].Valu
e).Name;
                string roles = string.Empty;

                using (userDbEntities entities = new userDbEntities())
                {
                    User user = entities.Users.SingleOrDefault(u => u.userid ==
userid);

                    roles = user.Roles;
                }

                e.User = new System.Security.Principal.GenericPrincipal( new
System.Security.Principal.GenericIdentity(userid, "Forms"), roles.Split(';'));
            }
            catch (Exception)
            {
                // Error occured
            }
        }
    }
}
```

Outdated Software

Figure 39 - Custom authorization snippet

The above code will check the current user roles to ensure that they are allowed to access the specific resource. Finally, decorators are used to indicate the roles that have access to specific code. The follow code snippet demonstrators the use of decorators to protect specific Web pages from being accessed:

```
public class MainController : Controller
{
    [Authorize]
    public ActionResult Index()
    {
        ViewBag.Message = "Welcome to the example web applications";

        return View();
    }

    [Authorize(Roles="Admin")]
    public ActionResult Private()
    {
        return View();
    }
}
```

Figure 40 - Authorization decorators in practice

The above snippet will prevent users that do not contain a role of `Admin` from accessing the `Private` method of the Web application controller. This can be extended to include any number of privileged roles, depending on the Web application purpose. For more information, *see* *Role-Based Authorization – Microsoft Developer Network* in the *Additional Resources* section below.

Additional Resources

Top 10 2013-A7-Missing Function Level Access Control – OWASP

- https://www.owasp.org/index.php/Top_10_2013-A7-Missing_Function_Level_Access_Control

CWE-285: Improper Authorization (2.6)

- http://cwe.mitre.org/data/definitions/285.html

Zend_Acl - Zend Framework

- http://framework.zend.com/manual/1.5/en/zend.acl.html

Apache Shiro | Java Security Framework

- https://shiro.apache.org/authorization.html

Role-Based Security – Microsoft Developer Network

- http://msdn.microsoft.com/en-us/library/52kd59t0%28v=vs.90%29.aspx

Cross-Site Request Forgery (CSRF)

Description

Web applications that are vulnerable to cross-site request forgery (CSRF) are unable to distinguish actions requested by a user's browser from actions the user intends to perform. When a user has an authenticated session with a site that stores authentication tokens in cookies, the user's browser automatically includes the user's authentication tokens to all HTTP requests it sends to that site. Since an attacker can cause a victim's browser to submit HTTP requests without the victim's consent, web applications that rely solely on authentication cookies to authorize actions will perform the unintended actions the attacker requests whenever the victim has an authenticated session.

Example Attack

Consider the following application administration form that is used to create a new user:

```
<!--http://www.example-host.com/create_user-->
<form name="input" action="create_user" method="post">
    Username: <input type="text" name="user"><br>
    Password: <input type="password" name="password"><br>
    Admin? <input type="checkbox" name="admin">
<input type="submit" value="Submit">
</form>
```

Figure 41 - Vulnerable user creation form

A screenshot of the administration form is given below:

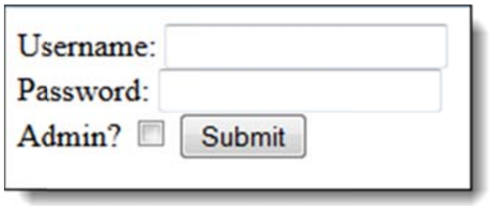


Figure 42 - The `create_user` form

In the above form, the only parameters required to form a valid request and create a new user are the user name, the admin flag, and the password. Suppose an attacker tricked an authenticated admin user into visiting the following webpage:

```
<html>
  <body onload='document.getElementById('csrf').submit()'>
    <form id="csrf" action="http://www.example-host.com/create_user"
    method="POST">
      <input type="hidden" name="user" value="hacker" />
      <input type="hidden" name="password" value="s3curem3" />
      <input type="hidden" name="admin" value="on" />
      <input type="submit" value="Submit request" />
    </form>
  </body>
</html>
```

Figure 43 - CSRF PoC form

Cross-Site Request Forgery (CSRF)

Should an authenticated admin user be lured into visiting the above page, the administrator's web browser will submit the following request to the administration form:

```
POST /create_user HTTP/1.1
Host: www.example-host.com
Cookie: SESSION_ID=aa562ceb8c6ab67fe80f945a35c0e717629c5b12

user=hacker&value=s3curem3&admin=on
```

Figure 44 - Forged user creation request

As is made clear above, the browser automatically appends the session cookies to the request, and from the perspective of the vulnerable application server, the request will be honored just as if the administrator had directly submitted it. The result is compromise of the application due to the creation of a new admin account with attacker-controlled values, namely the username "`hacker`" and the password "`s3curem3`."

Vulnerability Identification

Black Box Review

When reviewing applications for CSRF vulnerabilities, reviewers should first, familiarize themselves with the common CSRF protection schemes given in the *Remediation and Mitigation Techniques* section. All form actions should be examined to determine if CSRF protection exists on the page. If no CSRF protection is apparent, generate proof-of-concept code from a valid request using the *CSRF PoC form* above as a template. If the request is accepted by the application, then the form is vulnerable. If protection is present, then observe what happens when the CSRF protections are mangled or missing. Authenticate as different users and compare the CSRF protection values. Are CSRF tokens for double-submitted cookie values unique? Do they appear random? For the re-authentication method, can the re-authentication request be bypassed by omitting the credentials?

White Box Code Review

CSRF vulnerabilities occur due to the absence of a CSRF mitigation framework. Code reviews should ensure that a secure anti-CSRF framework is properly implemented, and is applied to the entire application. Typically, when an application is missing anti-CSRF controls on one form, it is missing them for all. Custom CSRF prevention mechanisms should be carefully reviewed to ensure that they cannot be bypassed. If a custom CSRF prevention framework is implemented, special care must be taken to ensure that form nonces are securely generated using secure random number generators. When in doubt, reviewers should create a *PoC form* to verify the secure operation of the application.

Remediation and Mitigation Techniques

Remediation of CSRF vulnerabilities requires that checks be implemented to verify the authenticity of the request. There are several ways to accomplish this, three of which are detailed below. Regardless of the adopted strategy, developers must ensure that anti-CSRF protections are included on all sensitive forms, and that the protections are included by default on all new forms.

Synchronizer Token

In the Synchronizer Token pattern, the application writes a securely generated cryptographic nonce to the page as a hidden form value, as demonstrated below:

```
<!--http://www.example-host.com/create_user-->

<form name="input" action="create_user" method="post">
    Username: <input type="text" name="user"><br>
    Password: <input type="password" name="password"><br>
    Admin? <input type="checkbox" name="admin">
<input type="hidden" name="csrf" value="3694cf84f194">
<input type="submit" value="Submit">
</form>
```

Figure 45 - User creation form protected with synchronizer token

Of these three methods of CSRF protection, synchronizer tokens are usually the preferred method. Their implementation can be language-agnostic, and requires no interaction from the user. An example POST request that includes a synchronizer token is shown below:

```
POST /create_user HTTP/1.1
Host: www.example-host.com
Cookie: SESSION_ID=aa562ceb8c6ab67fe80f945a35c0e717629c5b12; Secure; HttpOnly;

user=hacker&value=s3curem3&admin=on&csrf=3694cf84f194
```

Figure 46 - User creation request with synchronization token

Synchronizer tokens are an effective method of preventing CSRF attacks. However, their use does create certain usability concerns. For example, if the CSRF token is generated per request (as opposed to per session), it may generate invalid requests if the user clicks the browser back button and tries to resubmit the form.

JSP: Apache Struts

Apache Struts can automatically include synchronizer tokens in a Web form. An example login page is given below:

```
<%@ page language="java" %>
<%@ taglib  uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<html:html>
<head>
<title>
    <bean:message key="logon.title"/>
</title>
</head>

<html:errors/>

<html:form action="/create_user" focus="username">
```

Cross-Site Request Forgery (CSRF)

```
<table border="0" width="100%">
    <bean:message key="prompt.username"/>
    <html:text  property="username" size="16"/>
    <bean:message key="prompt.password"/>
    <html:password property="password" size="64"/>
    <html:submit>
        <bean:message key="button.submit"/>
    </html:submit>
</table>

</html:form>
</body>
</html:html>
```

Figure 47 - Example Apache Struts Form with synchronizer token

As you can see in the form above, there is no visible form parameter for the synchronizer token. This is because the token is added automatically as a feature of the `html:form` class. This page will be rendered like the example below:

```
<form name="create_user" method="post" action="submit">
    <input type="hidden" name="org.apache.struts.taglib.html.TOKEN"
                         value="41a390f7a1077ae74371625475760a7b">
    <input type="text" name="username" size="16">
    <input type="text" name="password" size="64">
    <input type="button" value="Submit">
</form>
```

Figure 48 - Rendered Apache Struts form with synchronizer token

As can be seen above, the Struts synchronizer token is present in the form, and will be sent to the application server when the form is submitted. The request will only be honored if the synchronizer token is present in the request, and is a token associated with the currently authenticated user session.

Double Submitted Cookie

The Double Submitted Cookie pattern validates requests by setting a cookie parameter when a user first requests a page. Client-side script then includes this value as a POST variable as well, which is demonstrated below:

```
POST /create_user HTTP/1.1
Host: www.example-host.com
Cookie: csrf=3694cf84f194; Secure;

user=hacker&value=s3curem3&admin=on&csrf=3694cf84f194
```

Figure 49 - Request protected by double submitted cookie

Provided the above `csrf` nonce is securely generated, an attacker will not be able to include a correct token with the forged request. This form of CSRF protection has the advantage of working on systems without an event handler. However, the `HttpOnly` cookie flag is not compatible with double submitted cookies.

Cross-Site Request Forgery (CSRF)

Java: Wicket

In this example a double submitted cookie form is built using the Wicket framework for Java. First, it requires the secure number generator and random number generator libraries:

```
import java.security.SecureRandom;
import java.util.Random;
```

Figure 50 - Java crypto libraries

The cookie library from the `javax.servlet` is imported next:

```
import javax.servlet.http.Cookie;
```

Figure 51 - Cookie handler library

The following Wicket libraries are also required:

```
import org.apache.wicket.markup.html.form.Form;
import org.apache.wicket.markup.html.form.HiddenField;
import org.apache.wicket.model.IModel;
import org.apache.wicket.model.Model;
import org.apache.wicket.protocol.http.WebRequest;
import org.apache.wicket.protocol.http.WebResponse;
import org.apache.wicket.validation.IValidatable;
import org.apache.wicket.validation.IValidator;
import org.apache.wicket.validation.ValidationError;
```

Figure 52 - Wicket libraries

Finally, these libraries are used to log attack attempts:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

Figure 53 - Logger libraries

The `DoubleSubmitCookieForm` will be extending the base `Form` object:

```
public class DoubleSubmitCookieForm extends Form {
```

Figure 54 - The class which creates a double submitted cookie form

What follows is the code used to generate a cryptographically secure random number. This is an important step, since the protection can be broken if an attacker can predict the token value:

```
  //Generate Cryptographically secure random number
  private static final Random random;
  static {
    random = SecureRandom.getInstance("SHA1PRNG");
  }
  //Generate new cookie value
  private static String token() {
    long r = random.nextLong();
      if (r < 0)
```

```
        {
       r = -r;
        }
    return Long.toString(r, 36);
  }
```

Figure 55 - Random token generator

The following code is responsible for setting the generated token nonce value in a HTTP cookie:

```
  //Set client's cookie value
  private String csrfProtection() {
    Cookie cookie = ((WebRequest) getRequest()).getCookie("csrf");
    if (cookie == null) {
      cookie = new Cookie("csrf", token());
      cookie.setPath("/");
      cookie.setSecure(true);
      ((WebResponse) getResponse()).addCookie(cookie);
    }
    return cookie.getValue();
  }
```

Figure 56 - Setting the double submitted cookie

This is the code that handles all form requests. Note that if the submitted value does not match the value stored in memory, the application throws an error and logs the request.

```
  //On form submit, check cookie value and compare with value stored in
  //server memory. Throw error if they don't match
  public DoubleSubmitCookieForm(String id, IModel model) {
    super(id, model);
    add(new HiddenField("csrf ", new Model
                 (csrfProtection())).setRequired(true).add(new IValidator()
{

    public void validate(IValidatable validatable) {
        if (!validatable.getValue().toString().equals(csrfProtection())) {
          log.warn("potential csrf attack, submitted value: " +
                   validatable.getValue());

          validatable.error(new ValidationError().setMessage
                              ("missing csrf protection cookie"));
        }
      }
    }));
  }
}
```

Figure 57 - Java Wicket form handler with Double Submit Cookie Value

To complete this example, client-side script is used to write the anti-CSRF cookie nonce value into a form:

Cross-Site Request Forgery (CSRF)

```
function ReadCookie(name)
{
  name += '=';
  var parts = document.cookie.split(/;\s*/);
  for (var i = 0; i < parts.length; i++)
  {
    var part = parts[i];
    if (part.indexOf(name) == 0)
      return part.substring(name.length)
  }
  return null;
}

$('#exampleForm').submit(function(){ //listen for submit event
    $.each(params, function(i,param){
      csrfValue = ReadCookie(csrfProtection)
        $('<input />').attr('type', 'hidden')
            .attr('name', csrfProtection)
            .attr('value', csrfValue)
            .appendTo('#commentForm');
    });

    return true;
});
```

If the value submitted by the client does not match the value held at the server, the handler throws an error and logs the request.

Re-authentication

The third CSRF prevention pattern is Re-authentication. The Re-authentication pattern is the simplest to understand: users are forced to resubmit their authentication credentials when performing a sensitive action. An example of this pattern is implemented in PHP below.

PHP

In the following code, an authenticated POST request is received that contains the end user's username and password, as well as the other form values that constitute the sensitive form action:

```
<?php
…

submit_request(sanitize_user_input($_POST[]))
{
    if(!auth_test($_POST["username"], $_POST["password"], $_SESSION[uid]))
    {
        invalid_request();
    }
    else
    {
        accept_request();
    }
}
```

Cross-Site Request Forgery (CSRF)

```
?>
```

Figure 58 - PHP form which forces re-authentication

In the above code, `auth_test` is used to verify that the supplied username and password are correct and associated with the currently authenticated session. The request is only honored if the authentication check succeeds.

The re-authentication method is one of the most secure ways to protect against CSRF; however, it is disruptive to the user experience, and an overreliance on it can encourage users to choose weak passwords. Re-entering passwords as a protection against CSRF should be used only on the most sensitive forms, if at all.

Additional Resources

OWASP CSRF Prevention Cheat Sheet
- https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

Microsoft - Preventing CSRF
- http://www.asp.net/web-api/overview/security/preventing-cross-site-request-forgery-%28csrf%29-attacks

OWASP – CSRFGuard 3 Installation Instructions
- https://www.owasp.org/index.php/CSRFGuard_3_Installation

NoCSRF Homepage
- http://bkcore.com/blog/code/nocsrf-php-class.html

Prevent CSRF using ASP.NET’s Anti Forgery Token
- http://blog.stevensanderson.com/2008/09/01/prevent-cross-site-request-forgery-csrf-using-aspnet-mvcs-antiforgerytoken-helper/

CSRF-Filter Project Website
- https://code.google.com/p/csrf-filter/

Open Redirect

Description

An open redirect vulnerability allows an attacker to hijack application redirection functionality. When an application relies upon user-controllable input to build a redirection URL, attackers can manipulate the input to send users to a dangerous website.

Example Attack

The following URL redirects a user to the `control_panel.php` page upon successful authentication:

```
http://www.example.com/login.php?redirect_url=/control_panel.php
```

Figure 59 - Redirect link to “`control_panel.php`”

However, the above parameter can be modified to redirect the user to a different page upon a successful login, as illustrated here:

```
http://www.example.com/login.php?redirect=http://attacker.com/
```

Figure 60 - Malicious redirect to the attacker’s site upon successful login

When a user clicks on the above link, the login form will be presented, as normal:

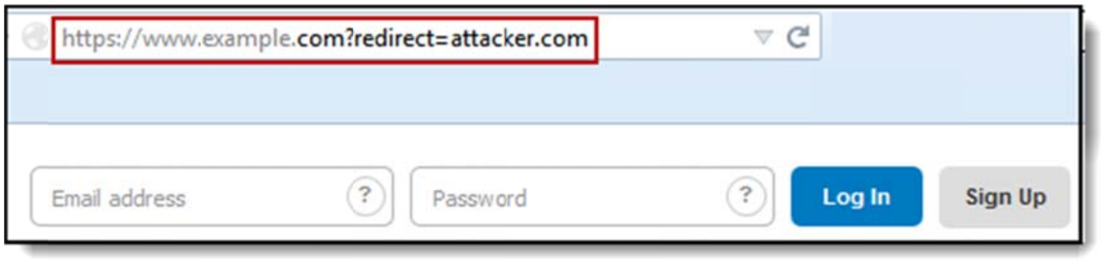
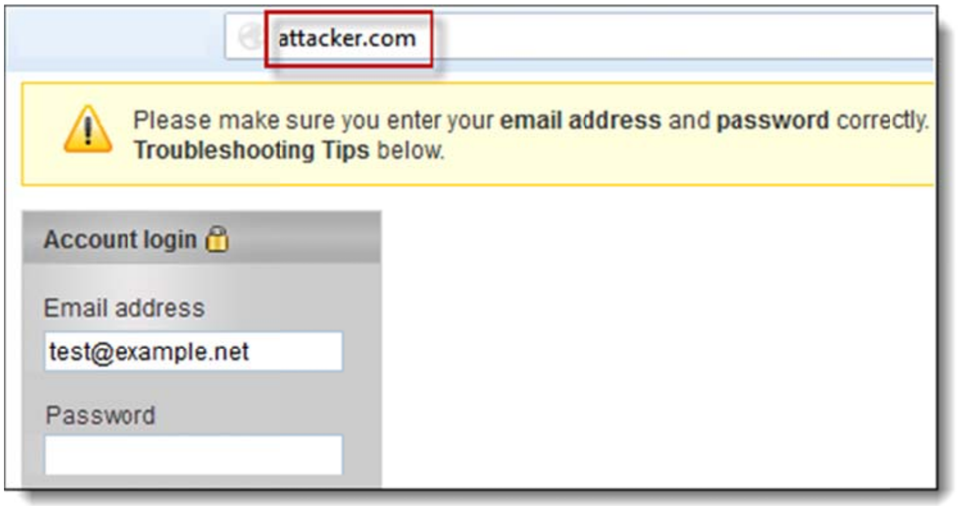


Figure 61 - User clicks on malicious link

However, upon successful authentication, the victim will be redirected to the URL specified by the `redirect` parameter in the query string. In this case, that URL is `attacker.com`. Upon reaching `attacker.com`, the user is presented with an authentication credential phishing page that attempts to lure the victim into resubmitting their credentials to the attacker-controlled site:

Open Redirect



Please make sure you enter your **email address** and **password** correctly. **Troubleshooting Tips** below.

Account login

Email address: test@example.net

Password:

Figure 62 - Malicious website prompts user to re-enter credentials

When the user re-enters their credentials, the malicious website redirects the user back to the real site and the victim user's session resumes as normal. The attacker retains the credentials and uses them for future access.

Vulnerability Identification

The most common reason that open redirect vulnerabilities exist is because of improper URL validation. When examining code, check to ensure that the URL is being properly scrutinized before redirecting the user. Parsing URLs for validity can be tricky, for that reason it is important to err on the side of caution when implementing a validation scheme.

The following code excerpts illustrate this vulnerability in Java, PHP, and ASP.NET.

Java

```
String queryString = request.getQueryString();
if( queryString.contains("redirect_url") ) {
        response.sendRedirect( request.getParameter("redirect_url" ) );
}
```

Figure 63 - Java unvalidated open redirect

The above snippet checks to see if the URL contains a `redirect_url` parameter and, if found, redirects the user to it.

PHP

```
if( isset( $_GET['url'] ) ) {
        header( 'Location: '.trim( $_GET['url'] ) );
}
```

Figure 64 - PHP unvalidated open redirect

This snippet checks for the existence of the `$_GET['url']` variable and redirects the user to the user specified URL using the `Location` header.

ASP.NET

```
Dim url As String = Request.QueryString["url"];
if( url != null ) {
      Response.Redirect( url );
}
```

Figure 65 - ASP.NET unvalidated open redirect

The above snippet sets the `url` variable to the user specified value and redirects the user if the value is not null. All of the above snippets fail to validate the user-supplied URL parameter.

Black Box Review

In most cases, open redirect vulnerabilities can be discovered simply by observing application behavior, and noting any instances of dynamic redirection. Anytime a redirect occurs, check to see if the page redirect location is user-controllable. If it is, determine if the redirection can be forced to a user-crafted page. If it can, then the site is vulnerable to open redirect. It is important to also validate that a user is not being directed to a Web application page they do not intend to visit, even if it is on a trusted domain. Examples include pages that perform an action such as sending a message, depositing money, etc.

White Box Code Review

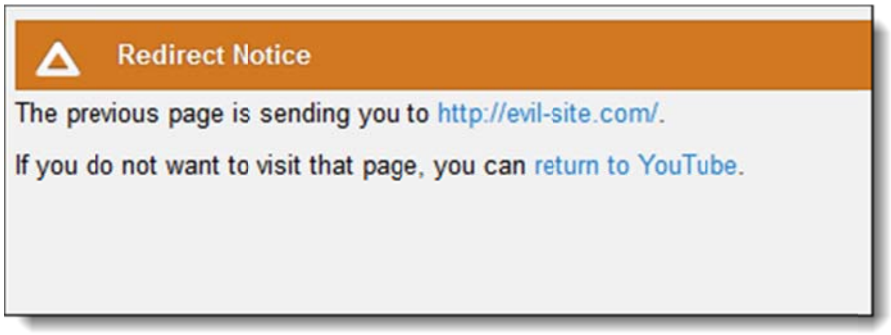
When performing a white box code review, all redirects in the application that handle user input should be enumerated. Examine the redirect and determine the purpose for it. If the application blindly redirects based on user-supplied input, then the code exposes an open redirect vulnerability. Depending on the purpose of the redirect, different mitigation steps should be taken. For more information on determining if a redirect is being properly sanitized, see the *Remediation and Mitigation Techniques* section below.

Remediation and Mitigation Techniques

In general, when redirecting users, a whitelist approach is the best option to mitigate open redirect vulnerabilities. If the user is redirected after authentication, the redirect should not send the user offsite. In this situation, a whitelist approach or a simple check of the redirect URL domain is enough to protect against open redirect attacks. Many situations do not warrant the need to take user input for a redirect at all, and the safest redirect is one not influenced by end users.

However, in some cases a whitelist approach is not practical. For example, when redirects are used to direct users to a third party site. In this situation, it is best to notify the user of the transition to a third party domain via an appropriate page, such as the following:

Open Redirect


Redirect Notice

The previous page is sending you to http://evil-site.com/.

If you do not want to visit that page, you can return to YouTube.

Figure 66 - YouTube redirect warning page

A notification such as this one makes alerts the user to the fact they are being sent to a different domain, thereby decreasing the likelihood of a successful phishing attack.

ASP.NET

The Microsoft MVC framework offers functionality for validating redirect actions to prevent open redirect vulnerabilities from occurring. The `IsLocalUrl` method in the MVC `UrlHelper` class performs a simple check to ensure that the redirect is local to the main application domain. The following code snippet accepts a user entered `redirectURL` parameter, and redirects the user only if the `redirectURL` value is local to the current domain:

```
if ( IsLocalUrl( redirectURL ) ) {
        return Redirect( redirectURL ); // Redirect to validated URL
} else {
        return RedirectToAction("Index", "Homepage"); // Redirect user back to
homepage
}
```

Figure 67 - ASP.NET validated URL

As can be seen above, if the user-supplied URL is not local to the current domain, the user is safely directed to the application homepage. For more information on prevent open redirects in ASP.NET, see the *Preventing Open Direction Attacks* article listed in the *Additional Resources* section.

Java

The `URL` class in Java can be used to validate input URLs before performing application redirects. The `getHost` method in the `URL` class can be used to validate that the redirect target domain is a trusted value. The following snippet takes a user-supplied `url` parameter and uses the `getHost` method to validate that it points to the correct domain:

```
URL inputURL = new URL( request.getParameter("url") );
if( inputURL.getHost() == "mydomain.com" ) {
        response.sendRedirect( inputURL );
} else {
        response.sendRedirect( "https://mydomain.com/” ); // Invalid host, send user
to homepage
}
```

Figure 68 - Java example URL host validation

If the user-supplied redirect value does not point to a trusted domain, the user is simply redirected to the homepage. For more information on parsing URL's in Java, see *Parsing a URL* listed in the *Additional Resources* section.

PHP

The PHP library contains a `parse_url` function that can be used to split a URL into an array containing sections such as scheme, host, query, etc. The URL host value can then be validated to ensure it is the same domain as the Web application. The following PHP snippet safely redirects a user to a provided `redirectURL` parameter only if the target domain is in a whitelist of trusted domains:

```
<?php
$url = $_GET['redirectURL'];
$whitelist = array( 'yourdomain.com', 'trusteddomain.com' );

if( filter_var( $url, FILTER_VALIDATE_URL ) ) {
    $host = parse_url( $url, PHP_URL_HOST);
       if( !$host ) {
              // Invalid URL provided, redirect back to homepage
         header( 'Location: /' ) ;
       } else if ( in_array( $host, $whitelist ) ) {
              // URL points to correct host, redirect URL
         header( 'Location: '.trim( $url ) );
       } else {
              // Bad URL provided, redirect back to homepage
              header( 'Location: /' ) ;
       }
} else {
    // Bad URL provided, redirect back to homepage
    header( 'Location: /' ) ;
}
```

Figure 69 - PHP example URL host validation

The above snippet validates the URL using the `filter_var` function along with the `FILTER_VALIDATE_URL` constant in PHP. This strict validation method is used to ensure that a URL is both valid and absolute. Once validated, `parse_url` is used to validate that the URL host is in the list of trusted domains before the redirect is performed.

Additional Resources

Official Google Webmaster Central Blog: Open redirect URLs: Is your site being abused?

- http://googlewebmastercentral.blogspot.com/2009/01/open-redirect-urls-is-your-site-being.html

CWE - CWE-601: URL Redirection to Untrusted Site ('Open Redirect')

- http://cwe.mitre.org/data/definitions/601.html

Open Redirect

Unvalidated Redirects and Forwards Cheat Sheet – OWASP

- https://www.owasp.org/index.php/Unvalidated_Redirects_and_Forwards_Cheat_Sheet

Preventing Open Redirection Attacks (C#) : The Official Microsoft ASP.NET Site

- http://www.asp.net/mvc/tutorials/security/preventing-open-redirection-attacks

Parsing a URL (The Java™ Tutorials > Custom Networking > Working with URLs)

- http://docs.oracle.com/javase/tutorial/networking/urls/urlInfo.html

PHP: parse_url - Manual

- http://us3.php.net/parse_url

Cross-site Scripting (XSS): DOM-Based

Description

The document object model (DOM) is the language-independent object model used to visually represent and manipulate markup languages such as HTML and XML. DOM-based XSS vulnerabilities occur in Web browsers when client-side script insecurely modifies the DOM resulting in the injection of arbitrary JavaScript instructions. DOM-based XSS vulnerabilities can also result from the use of unsafe JavaScript functions such as `eval()`. DOM-based XSS is distinct from reflective and stored XSS in that it is a bug in client-side code, and rarely involves server interaction after the user-agent has retrieved the page.

Example Attack

An application maintains dynamic links using URL fragments:

```
var fragment = document.location.hash.substring(1);
document.write("<a href='" + fragment + "'>Link #1</a>");
```

A specially crafted URL can inject a JavaScript URL resulting in the execution of attacker-controlled script into the page, an example payload is shown below:

```
http://website.com/index.php#javascript:alert(1)
```

Resulting in the follow HTML being added to the DOM:

```
<a href="javascript:alert(1)">Link #1</a>
```

When the user clicks the link, the JavaScript is executed in the context of the current domain.

Vulnerability Identification

DOM-based XSS vulnerabilities can affect any JavaScript function that modifies the DOM, and is not limited to the `document.write` function used in the example above. The following function should be carefully inspected to discover DOM-based XSS vulnerabilities:

```
document.write()    //Writes HTML expressions or JavaScript code to a document
document.writeln()  //Writes HTML expressions or JavaScript code to a document and \n
eval()              //Evaluates arbitrary JavaScript statements
setInterval()       //Calls JavaScript function after specified time delay
setTimeout()        //Same as setInterval()
navigate()          //Loads document from specified URL
element.innerHTML() //Sets or returns inner HTML of element
element.innerHTML() //Sets or returns outer HTML of element
.onClick()          //Executes JavaScript when an element is clicked
.onLoad()           //Executes JavaScript when the element loads
.onError()           //Executes JavaScript when an error is thrown
```

Figure 70 - Potentially unsafe JavaScript methods/attributes

Because there are a tremendous number of potential payloads and permutations, manual testing can be inefficient against all but the smallest applications. As a result, commercial Web application scanners are available to automate testing and achieve better coverage. Burp

Suite by PortSwigger, WebInspect by HP, or the open-source Zed Attack Proxy are popular choices for automating XSS injection attempts.

Remediation and Mitigation Techniques

Remediation of DOM-based XSS vulnerabilities can be challenging, and may require large parts of client-side code to be re-written. Ideally JavaScript template libraries such as Underscore.js or Mustaches should be used to control DOM content. These libraries will automatically perform much of the encoding required to prevent XSS vulnerabilities. However, depending on the context within the DOM additional encoding may be required. Should business requirements prohibit the use of a template library, the following guidelines should be enforced for all client side code:

- Never use string concatenation to build HTML tags.
- User controlled content should always be inserted as text within HTML tags (e.g. `<pre>user data</pre>`), and HTML entity encoding must be applied to convert script code into harmless output.
- Avoid placing user controlled content within HTML tag attributes.
- Implement modern browser headers such as `Content-Security-Policy` to prevent unsafe inline JavaScript from executing.

Additional Resources

OWASP Cross-Site Scripting Page
- http://www.owasp.org/index.php/Cross_Site_Scripting

Understanding Malicious Content Mitigation for Web Developers
- http://www.cert.org/tech_tips/malicious_code_mitigation.html

OWASP - DOM Based XSS Prevention Cheat Sheet
- https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet

OWASP - DOM Based XSS
- https://www.owasp.org/index.php/DOM_Based_XSS

WebAppSec.org - DOM Based Cross-Site Scripting
- http://www.webappsec.org/projects/articles/071105.shtml

Risks of Unsafe JavaScript Functions
- http://blog.csnc.ch/2013/01/dom-based-xss-unsafe-javascript-functions/

HP - WebInspect
- http://www8.hp.com/us/en/software-solutions/software.html?compURI=1341991/

PortSwigger: Burp Suite
- http://www.portswigger.net/burp/

SQL Injection

Description

SQL injection (SQLi) is an application vulnerability that allows an attacker to hijack improperly constructed SQL statements. By injecting specially crafted inputs through a vulnerable application, an attacker can change the intended behavior of a SQL statement and execute malicious commands on the back-end database.

A blind SQL injection vulnerability occurs when a SQL injectable statement does not directly return the results of injected commands to the user. Though the results themselves are not directly accessible by an attacker, the attacker can still retrieve information from the database by indirect means.

Example Attack

The following sections will illustrate two ways by which blind SQL injection issues can be identified and exploited: content-based attacks, and time-based attacks.

Content-Based Attack

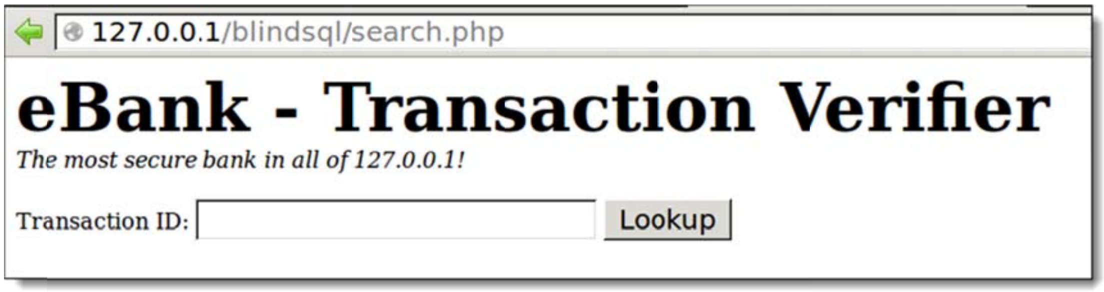
Content-based blind SQL injection attacks are characterized by the injection of true or false boolean SQL logic into application parameter values. This technique is useful as boolean logic that equates to true in a SQL query will always result in a successful query, and depending on how the application is implement, this might result in some content in the application response. In some cases, the presence of a SQL injection vulnerability can be inferred by observing the application take different logic paths in response to the SQL injection attempts. Regardless, query logic that always equates to false should result in a response that is significantly different from the true case. This difference in application behavior strongly indicates the presence of a SQL injection vulnerability in the corresponding application. To better understand this, consider the following HTML form that is constructed via dynamic PHP and is vulnerable to content-based blind SQL injection (the yellow highlighted line):

```
    $con = mysqli_connect("localhost", "root", "firewire", "user_database");

    try {
        $result = mysqli_query( $con, "SELECT * FROM transactions WHERE id =
".$_GET['id']);
    } catch ( Exception $e ) {
        echo 'ERROR';
    }

    $output = mysqli_fetch_array( $result );

    if( !$result ) {
        echo '<h1>Transaction does not exist!</h1><br />';
    } else {
        echo '<h1>Transaction exists!</h1><br />';
    }
```

Figure 71 - Content-based blind SQL injectable form

In a web browser, the above form is displayed as the following:

SQL Injection



127.0.0.1/blindsql/search.php

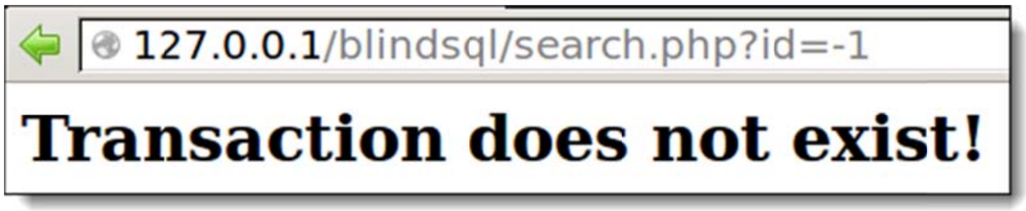
eBank - Transaction Verifier

The most secure bank in all of 127.0.0.1!

Transaction ID:

Figure 72 - The vulnerable transaction form

The preceding code snippet takes user input and constructs a database query to check for the transaction ID submitted by the user. The query to check for the transaction ID is unsafely performed via string concatenation, and is thus vulnerable to SQL injection. However, due to a secure application configuration, malformed SQL queries do not result in verbose error messages that return sensitive information regarding the SQL query or structure of the underlying database, as commonly seen in classical SQL injection attacks. By using the content-based blind SQL injection technique, an attacker can still determine if the above form is vulnerable to SQL injection, despite the absence of verbose error messages. To do this, the attacker first uses a value that will generate a known query result. In this example, a transaction ID of "-1" returns false and is used as the base query:

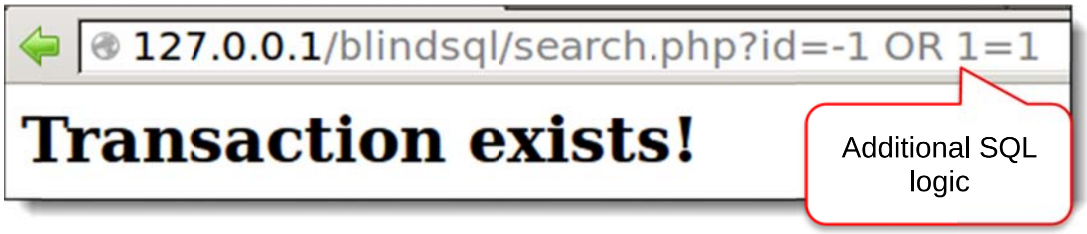


127.0.0.1/blindsql/search.php?id=-1

Transaction does not exist!

Figure 73 - Application returning a false value for the base query

Now that the attacker has a point of reference, SQL logic that equates to true is injected to test if the application is vulnerable to SQL injection:

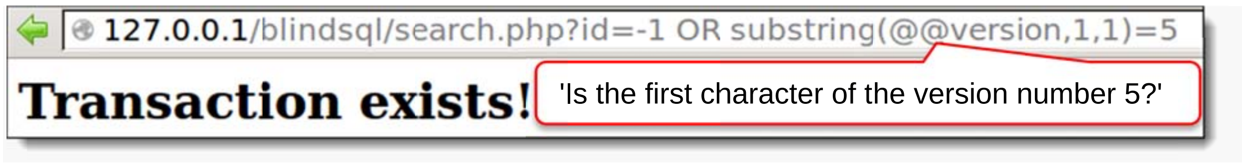


127.0.0.1/blindsql/search.php?id=-1 OR 1=1

Transaction exists!

Figure 74 - Additional SQL logic forcing the application to return true

Since the application has returned a true response ("Transaction exists!") in response to a transaction ID that is clearly invalid, an attacker could reasonably suspect that the application is vulnerable to SQL injection. To obtain final confirmation, the attacker can craft queries to pull information out of the database one character at a time, as shown below:



127.0.0.1/blindsql/search.php?id=-1 OR substring(@@version,1,1)=5

Transaction exists!

Figure 75 - Enumeration of MySQL database version through blind SQL injection

As can be seen in the above screenshot, the web application has responded with a "Transaction Exists" message (i.e. a "true" response), meaning the attacker has retrieved the first character of the MySQL database version ("5"). As manually enumerating data in this manner would be a time-consuming and error-prone task, tools such as the popular `sqlmap` application are commonly used to quickly extract database content using blind SQL injection techniques:

```
mandatory@ubuntu:~/Pentest/sql/sqlmap$ ./sqlmap.py -u
'http://127.0.0.1/blindsql/search.php?id=4' --tables
...
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)?
[y/N] y
sqlmap identified the following injection points with a total of 118 HTTP(s)
requests:
---
Place: GET
Parameter: id
    Type: AND/OR time-based blind
    Title: MySQL > 5.0.11 AND time-based blind
    Payload: id=4 AND SLEEP(5)
---
[12:17:50] [INFO] the back-e
web server operating system:                           4 or 12.10 (Raring Ringtail or
Precise Pangolin or Quantal
web application technology: Ap    e 2.2.22, PHP 5.3.10
back-end DBMS: MySQL 5.0.11
```

sqlmap enumerating MySQL version number

Figure 76 - sqlmap exploiting content-based blind SQL injection

As can be seen in the above output, the `sqlmap` tool has identified the MySQL version using blind SQL injection techniques, as described above.

Time-Based Attack

Even when an application does not give an explicit true/false response to an injected request, an attacker can still infer the results of these requests using time-based blind SQL injection methods. Consider the following HTML form that is vulnerable to time-based blind SQL injection at the highlighted line:

```
if( isset( $_GET['id'] ) ) {
    $con = mysqli_connect("localhost", "root", "firewire", "user_database");

    try {
        $result = mysqli_query( $con, "SELECT * FROM transactions WHERE id =
".$_GET['id']);
    } catch ( Exception $e ) {
        echo 'ERROR';
    }

    $output = mysqli_fetch_array( $result );

    echo '<h1>Transaction complete</h1><br />';

    mysqli_close( $con );
} else {
    echo '
        <font size="7"><b>eBank - Transaction Verifier</b></font><br />
        <i>The most secure bank in all of 127.0.0.1!</i><br /><br />

        <form action="" method="get">
        Transaction ID: <input type="text" name="id"></input>
        <input type="submit" value="Lookup"></input>
        </form>
   ';
}
```

Figure 77 - Time-Based Blind SQL Injectable form

An attacker can use queries that are designed to cause detectable delays in the response from the database in order to identify whether SQL injection vulnerabilities exist. For example, if the payload contains a `BENCHMARK()` or `SLEEP()` statement with some delay value – say 5 seconds – the result of the query can be inferred by timing responses from the server and checking whether they took roughly 5 seconds to complete. Using this strategy, an attacker can slowly enumerate the entire database contents with a request such as this one:

```
http://127.0.0.1/blindsql/search.php?id=4 AND 6463=IF((ORD(MID((SELECT
DISTINCT(IFNULL(CAST(schema_name AS CHAR),0x20)) FROM INFORMATION_SCHEMA.SCHEMATA
LIMIT 0,1),8,1))=116),SLEEP(5),6463)
```

The following screenshot shows the response after injecting the preceding URL (assume a 5 second delay was encountered):

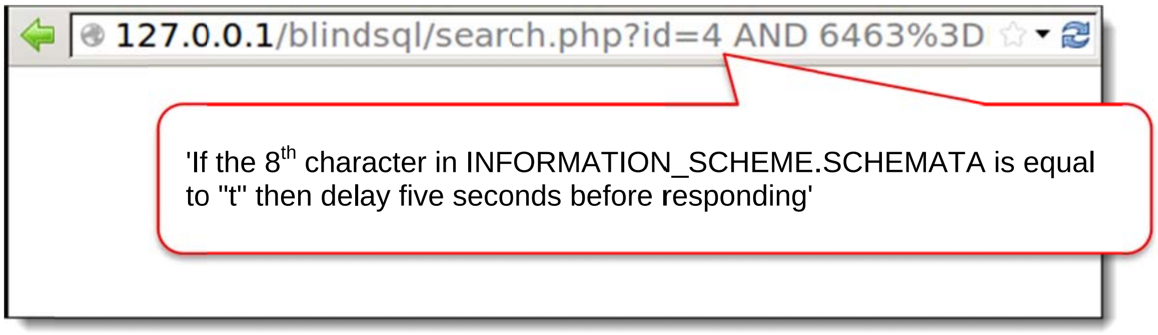

127.0.0.1/blindsql/search.php?id=4 AND 6463%3D
'If the 8th character in INFORMATION_SCHEME.SCHEMATA is equal to "t" then delay five seconds before responding'

Figure 78 - Lengthy time-based SQL injection query

Though the response is empty, the time to receive the response took just over five seconds, thereby strongly indicating that the query returned true. Per the injection test, a true response means that the 8th character in the `SCHEMATA` table is the ASCII letter "`t`." Continuing in this manner, an attacker can enumerate the entire database systematically by performing a series of probing SQL injections that return either true or false. Like content-based blind SQL injection attacks, time-based attacks are rarely performed manually; instead, tools such as `sqlmap` can be used to automate the process:

```
[12:22:45] [INFO] fetching database names
[12:22:45] [INFO] fetching number of databases
[12:22:45] [WARNING] time-based comparison requires larger statistical model, please
wait...............................
do you want sqlmap to try to optimize value(s) for DBMS delay responses (option '--
time-sec')? [Y/n] y
[12:22:52] [WARNING] it is very important not to stress the network adapter during
usage of time-based payloads to prevent potential errors
5
[12:22:57] [INFO] retrieved:
[12:23:02] [INFO] adjusting time delay to 1 second due to good response times
information_schema
[12:24:13] [INFO] retrieved: mysql
[12:24:33] [INFO] retrieved: perfor
```

Figure 79 - `sqlmap` exploiting Time-Based Blind MySQL injection

As can be seen in the above output, `sqlmap` injects a delay command into the SQL query and measures the response time. One character at a time is enumerated until the entire database has been completely extracted. While this attack is time-consuming, it is effective and has the same impact as a regular non-blind SQL injection attack.

Stored Procedure

Use of stored procedures is often recommended to remediate SQL injection attempts. However if done insecurely, stored procedures can also be vulnerable if implemented insecurely. The following code snippet takes a `userid` parameter and returns all results from the `users` table:

```
CREATE PROCEDURE GetUser
       @userid varchar(256)
AS
BEGIN
       DECLARE @query nvarchar(1024);
       SET @query = 'SELECT * FROM users WHERE id = ''' + @userid + ''' ';
       EXEC(@query);
END
GO
```

Figure 80 - Vulnerable stored procedure

Due to the use of string concatenation in the stored procedure shown above, it is vulnerable to SQL injection via the `userid` parameter. The process for exploiting SQL injection in stored procedures is the same as a regular SQL injection vulnerability.

Vulnerability Identification

The follow sections will cover how to identify blind SQL injection vulnerabilities in Web applications through the use of injection test cases and static code analysis.

SQL Injection

Black Box Review

When performing black box testing, it is important to test that all user input is being properly validated, as SQL injection is not limited to just URL parameters. For example, HTTP header names and header values are often improperly trusted by developers and used to construct dynamic SQL queries. In addition, SQL injection can also be performed in any application that uses untrusted input to construct a SQL statement, and is not just limited to HTTP based applications. Inserting a single quote or SQL comment delimiter `--` into a parameter can often reveal that an application is performing SQL querying insecurely, as these will typically break dynamically constructed queries. The following are common black box SQL injection test cases:

```
MySQL
' or '1'='1
' and '1'='0
' or 1=1 --
or 1=1 #
UNION SELECT IF(1,SLEEP(5),null) –-
UNION SELECT IF(1,BENCHMARK(10000000,MD5('A')),null) --
```

```
MSSQL
' or '1'='1 /*
' and '1'='0 --
' or 1=1 --
or 1=1 --
' waitfor delay '0:0:5' –-
```

Tools such as `sqlmap` can be used to automate SQL injection tests against URL parameters, cookies, form values, and other input fields. It should be noted that due to network latency and server performance issues, false positives are common when attempting to identify SQL injection vulnerabilities via time-based techniques alone. For more information on using the `sqlmap` tool, see *SQLmap tutorial for beginners – hacking with SQL injection* in the *Additional Resources* section below.

White Box Code Review

When reviewing source code for SQL injection vulnerabilities, use of string concatenation to build SQL queries should be located and replaced with parameterized queries that safely bind user input to SQL query statements. It is important to not only audit the source code for string concatenation vulnerabilities, but to also check that all stored SQL procedures do not make use of dynamic string concatenation to dynamically construct queries from unvalidated user input.

ASP Source Code Analysis

The "Microsoft Source Code Analyzer for SQL Injection" tool can be used to locate SQL injection vulnerabilities in ASP code. The following syntax demonstrates use of this tool to audit ASP source code:

```
msscasi_asp.exe /input="c:\web\index.asp"
```

SQL Injection

The above command will display warnings for areas that may be vulnerable to SQL injection. For more information on this tool, see *The Microsoft Source Code Analyzer for SQL Injection* listed in the *Additional Resources* section.

Remediation and Mitigation Techniques

The root cause of SQL injection is the dynamic construction of SQL statements using string concatenation, which allows the mixing of control information and data. The assessment team recommends the following to address SQL injection:

- Use parameterized queries (also referred to as prepared statements) when constructing SQL statements to ensure that input cannot be misinterpreted as SQL syntax. The use of parameterized queries within stored procedures is a secure alternative.
- Avoid the use of string concatenation methods when constructing dynamic SQL statements within stored procedures. It is a common misconception among developers to assume that SQL injection vulnerabilities only occur in higher-level programming languages, and not in SQL programming constructs; strict input validation and parameterized queries also apply to stored procedure development.
- Perform thorough server-side validation of all input prior to use within SQL statements.
- Associate a data type to each SQL input parameter where possible.
- Apply the principle of least privilege to all accounts connecting to the database.

Parameterized Queries

Parameterized queries (also known as prepared statements) are precompiled SQL statements that take values as inputs rather than dynamically constructing a query with the parameters built in. They offer not only a more secure querying method, but also a significant performance increase due to the database being able to cache more generalized statements instead of specific query strings.

PHP

Use of the PHP Data Objects (PDO) library provides a safe way to perform SQL queries without being vulnerable to SQL injection. The following is a PDO version of the vulnerable transaction lookup snippet previously shown:

```
$sql = "SELECT * FROM transaction WHERE `id` = :id";
$ps = $pdo->prepare($sql);
$ps->bindValue("id", $_GET['id'], PDO::PARAM_INT);
$ps->execute();
```

Figure 81 - Secure PDO querying

The above snippet first declares the SQL statement using the `prepare()` method. All variables are then set using the `bindValue()` method which binds user input to the SQL query logic in a manner that cannot be abused to execute SQL injection attacks. For more information on using the PDO library, see *PHP: Prepared Statements with PDO Tutorial* in the *Additional Resources* section below.

SQL Injection

Java

The JDBC driver in Java offers developers a secure way to perform parameterized SQL queries that are safe from SQL injection. The following code excerpt demonstrates how to safely construct a SQL query using the JDBC drivers:

```
String selectStatement = "SELECT * FROM actions WHERE action_id = ?";
PreparedStatement prepStmt = con.prepareStatement(selectStatement);
prepStmt.setString(1, actionid);
ResultSet rs = prepStmt.executeQuery();
```

Figure 82 - JDBC parameterized querying

The above code declares a SQL statement and then binds the user specified value stored in variable `actionid` to the `action_id` parameter. With this snippet, SQL injection is not possible and the database is safe from injection because the user-supplied input is not concatenated directly to the SQL query control statement, and is instead just treated as data. For more information about safe querying in Java, see *Java - Using Prepared Statements* in the *Additional Resources* section below.

ASP.NET

ASP.NET provides built-in support for parameterized SQL querying that can be used to safely query backend databases. The following code excerpt uses the `SqlDataAdapter` method to construct a SQL query for querying a musician's first and last name from a database:

```
using System.Data;
using System.Data.SqlClient;

using (SqlConnection connection = new SqlConnection(connectionString))
{
       DataSet exampleDataset = new DataSet();
       SqlDataAdapter dataAdapter = new SqlDataAdapter( "SELECT first_name, last_name
FROM musicians WHERE id = @id", connection);
       dataAdapter.SelectCommand.Parameters.Add("@id", SqlDbType.VarChar, 11);
       dataAdapter.SelectCommand.Parameters["@id"].Value = musician_id.Text;
       dataAdapter.Fill(exampleDataset);
}
```

Figure 83 - ASP.NET parameterized querying using `SqlDataAdapter`

As can be seen above, a new `SqlDataAdapter` object with the SQL query to be performed. Parameters are specified with a preceding @ symbol and are later declared using `Parameters.Add` and set using `Parameters['example'].Value`. In this example, the `id` parameter is declared to be of type `VarChar` with a length of 11 characters. `id` is then set to the value stored in the `musician_id` variable, which was directly supplied by the end user. Finally, the `dataAdapter.Fill()` method is used to complete the SQL query. For more information about safe querying in ASP.NET see *How To: Protect From SQL Injection in ASP.NET* in the *Additional Resources* section below.

SQL Injection

Securing Stored Procedures

The following stored procedure is not vulnerable to SQL Injection:

```
CREATE PROCEDURE SP_ProductSearch @prodname varchar(400) = NULL AS
DECLARE @sql nvarchar(4000)
SELECT @sql = ' SELECT ProductID, ProductName, Category, Price ' +
              ' FROM Product Where '
IF @prodname IS NOT NULL
  SELECT @sql = @sql + ' ProductName LIKE @prodname'
EXEC sp_executesql @sql, N'@prodname varchar(400)',@prodname
```

Figure 84 - Secure stored SQL procedure

The above stored procedure does not place user input inside single quotes. Also, instead of calling `EXEC` on the SQL statement itself, it calls `EXEC` on the `sp_executesql` call, and passes the user input in as a second parameter to that call. If an attacker attempts to inject a payload such as `' OR '1'='1'--` into this procedure, it will result in a query for a product with that ID, and not result in the altering of the SQL query logic itself.

Additional Resources

SQL Command.Prepare Method (System.Data.SqlClient)

- http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.prepare%28v=vs.110%29.aspx

Java - Using Prepared Statements

- http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html

PHP: Prepared statements and stored procedures - Manual

- http://ca2.php.net/pdo.prepared-statements

Are stored procedures safe against SQL injection? : Palisade

- http://palizine.plynt.com/issues/2006Jun/injection-stored-procedures/

SQL Injection - OWASP

- https://www.owasp.org/index.php/SQL_Injection

Query Parameterization Cheat Sheet - OWASP

- https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet

How To: Protect From SQL Injection in ASP.NET

- http://msdn.microsoft.com/en-us/library/ff648339.aspx

PHP: Prepared Statements with PDO Tutorial

- http://prash.me/php-pdo-and-prepared-statements/

Sqlmap tutorial for beginners – hacking with sql injection

- https://ctrlaltnarwhal.wordpress.com/2012/10/27/hacking-with-sqlmap-a-beginners-guide/

The Microsoft Source Code Analyzer for SQL Injection

- https://support.microsoft.com/kb/954476

XML External Entity Injection

Description

An XML External Entity is an object that can be used to access local or remote content. XML External Entity Injection vulnerabilities occur when an entity declaration contains user tainted data, the XML processor may disclose confidential information, or suffer a denial-of-service attack.

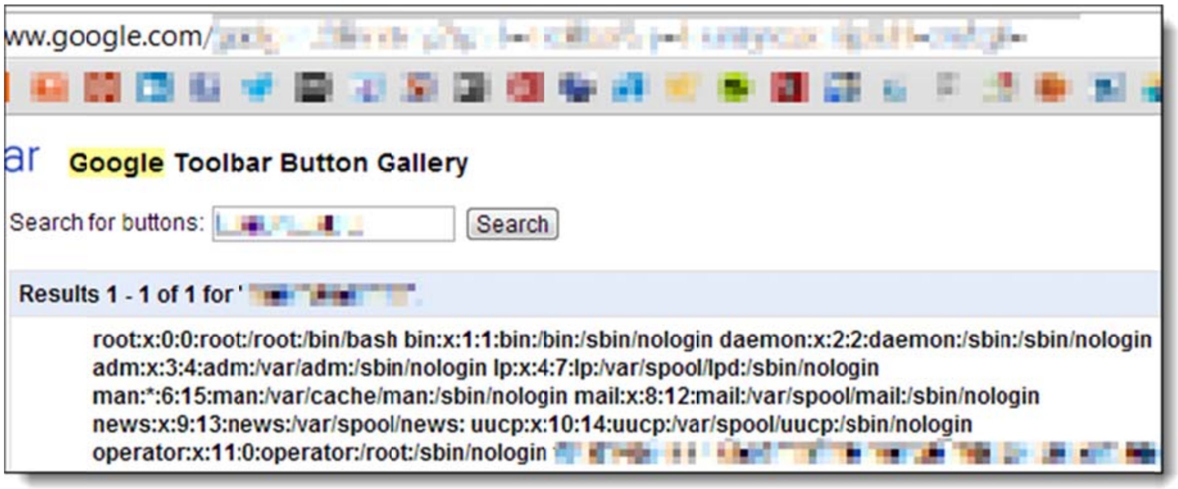
Example Attack

In 2014, the security research group Detectify disclosed a remote file read vulnerability in Google's servers using an app known as the "Google Toolbar Button Gallery." This app allowed a remote attacker to read any file Google's server process could read. The researcher uploaded a file to Google's servers, which contained code similar to the example below:

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

Figure 85 - XXE remote file read exploit

Since Google's XML parser did not allow locally sourced URI's passed to the `SYSTEM` identifier, the researchers successfully posted this file to their account and when they opened the file, this is a screenshot of what they found:



```
root:x:0:0:root:/root:/bin/bash bin:x:1:1:bin:/bin:/sbin/nologin daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
man:*:6:15:man:/var/cache/man:/sbin/nologin mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/var/spool/news: uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
```

Figure 86 - Google remote file read PoC

As you can see above, the `/etc/passwd` file stored locally on Google's server was readable by the researchers. An alternate attack is given below:

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

Figure 87 - XXE denial of service exploit

Since the file resource `/dev/random` never terminates, the server enters a denial of service condition, where the URI request will never complete. The vulnerability was responsibly disclosed to Google and no longer exists in production.

Vulnerable Code Example: Java Document Builder Factory

`DocumentBuilderFactory` is a Java Class that handles XML processor features. An example of a vulnerable `DocumentBuilderFactory` object is given below:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
try {
dbf.setFeature("http://xml.org/sax/features/external-general-entities", TRUE);

dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", FALSE);

// remaining parser logic
...
```

Figure 88 - Vulnerable `DocumentBuilderFactory` object

Vulnerability Identification

Black Box Review

Black box testing for XXE injection is relatively straightforward, first create a valid XML document that makes an external reference to a file or URL, and then attempt to load the malicious document. The following is a maliciously constructed document to load the “`/etc/passwd`” file on Linux machines upon successful XXE exploitation:

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

Figure 89 - Example `SYSTEM` XXE payload

The `PUBLIC` identifier is similarly vulnerable:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE test [
      <!ELEMENT test ANY >
      <!ENTITY injectedentity PUBLIC "-//test//testtype//EN" "file:///etc/passwd"
>]><test>&injectedentity;</test>
```

Figure 90 - Example `PUBLIC` XXE payload

This XML document will attempt to load the “`/etc/passwd`” file into the “`&injectedenitity`” entity. Depending on the application being tested, the file or entity name should be changed to fit with the situation. If, for example, the attack was being attempted on a Windows machine the file should be changed to “`file:///c:/boot.ini`” as Windows does not have a “`passwd`” file.

XML External Entity Injection

Even if external file references are not allowed, an attacker could still cause a denial of service condition with the following entity declaration:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE lolz [

    <!ENTITY lol "lol">
    <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
    <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
    <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
    <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
    <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
    <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
    <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
    <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Figure 91 - Example `PUBLIC` XXE payload

This recursively defined XML Entity will attempt to write billions of "lol" strings into the response, causing the server to become unresponsive.

White Box Code Review

When reviewing source code for XXE injection vulnerabilities, the assessment team recommends the following:

- Locate functions that process XML data in the web application
- Check if user input is being processed either directly or indirectly by any of the enumerated functions.
- If user input is being used, the XML external entity declaration functionality should be disabled as shown below.

Remediation and Mitigation Techniques

PHP

To prevent XML external entity injection attacks in PHP it is recommended to remove the libxml external entity loaded if it is not being used for XML parsing. If the web application uses the SimpleXML library this is the only way to prevent external entities from being injected. Other PHP libraries such as XMLReader provide more refined functionality to prevent remote entity inclusion and other unwanted actions. The only real solution however, is to sanitize user supplies XML from containing injected content. The following snippet demonstrators disabling external XML entities along for both SimpleXML and XMLReader:

```
// Sample secured SimpleXML loader script
libxml_disable_entity_loader(true); // Disable remote entity inclusion
$xml_doc = simplexml_load_string( $userXML ); // Load XML document through SimpleXML

// Sample secured XMLReader script
libxml_disable_entity_loader(true); // Disable remote entity inclusion
$xml_doc = XMLReader::xml( $userXML, 'UTF-8', LIBXML_DTDLOAD|LIBXML_DTDATTR );
```

Figure 92 - Disabling entity inclusion for XMLReader & SimpleXML in PHP

The function “`libxml_disable_entity_loader()`” disables XML entity inclusion for SimpleXML, DOM, and XMLReader in PHP. This effectively mitigates any XXE attacks as external entities are simple not loaded. The XMLReader object offers finer tuned control with flags such as “`LIBXML_DTDLOAD`” preventing entity inclusion and “`LIBXML_DTDATTR`” specify the XML must be valid DTD.

ASP.NET

When parsing XML with ASP.NET it is important to disable entity inclusion to prevent XXE injection. The following code snippet demonstrates how to properly disable external entity loading from the `XmlDocument` loader object:

```
XmlDocument xml_doc = new XmlDocument();
xml_doc.XmlResolver = null; // Prevent external entities from being included
xml_doc.LoadXml( xmlInput );
```

Figure 93 - Disabling entity inclusion for XmlResolver in ASP.NET

Setting the `xml_doc`’s “`XmlResolver`” property to `false` will prevent any XXE attacks from being preforming via user input. This is the most secure method to ensure that the application is not vulnerable to injection.

Java Document Builder Factory

To configure Document Builder correctly, a developer should set the following options in the object prototype:

```
dbf.setFeature("http://xml.org/sax/features/external-general-entities", FALSE);
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", TRUE);
```

Figure 94 - Disabling vulnerable `DocumentBuilderFactory` features

This disables the vulnerable features in Document Builder.

Additional Resources

OWASP XML External Entity Processing

- https://www.owasp.org/index.php/XML_External_Entity_%28XXE%29_Processing

Detectify - How we got read access on Google’s production servers

- http://blog.detectify.com/post/82370846588/how-we-got-read-access-on-googles-production-servers

websec.io - Preventing XXE in PHP

- http://websec.io/2012/08/27/Preventing-XEE-in-PHP.html

How to prevent XXE attack (XmlDocument in .net)

- https://stackoverflow.com/questions/14230988/how-to-prevent-xxe-attack-xmldocument-in-net

XML External Entity Injection

Testing for XML Injection (OWASP-DV-008) - OWASP

- https://www.owasp.org/index.php/Testing_for_XML_Injection_%28OWASP-DV-008%29

Apache - XML Entity and URI Resolvers

- https://xerces.apache.org/xml-commons/components/resolver/resolver-article.html

SecPod Research Blog - XML External Entity Attack

- http://secpod.org/blog/?p=1337

JSON Hijacking

Description

JSON hijacking vulnerabilities occur when applications exposing a JavaScript Object Notation (JSON) API return sensitive information without implementing the proper authorization. This vulnerability is similar to CSRF because a successful attack depends upon tricking a vulnerable JSON service into responding to an illegitimate request sent from the user's browser. JSON hijacking attacks can result in the disclosure of sensitive data, and the execution of arbitrary JavaScript.

Example Attack

To exploit a JSON hijacking vulnerability, an attacker first creates a page that overrides the `Object.prototype.__define__Setter` method for the targeted object. The following is an example of an insecure JSON response:

```
GET http://vulnerable-site.com/api/json/getAuthToken/ HTTP/1.1
Host: www.vulnerable-site.com

[{“csrf_token”: "3417209384712348927118"}]
```

Figure 95 - Vulnerable JSON request

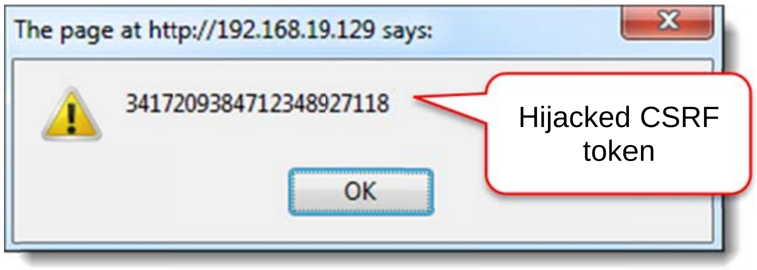
Because the JSON object is being returned through a GET request, this web application is vulnerable to a JSON hijacking attack. The attacker will include this page inside of a script “`src`” tag on a third party domain to hijack the JSON object. The following is a proof of concept for this scenario:

```
<!DOCTYPE html>
<html>
       <head>
           <script type="text/javascript">
           Object.prototype.__defineSetter__('csrf_token',function(obj)
          {
              alert(obj);
          });
           </script>
           <script src="http://vulnerable-site.com/api/json/getAuthToken/"></script>
       </head>
</html>
```

Figure 96 - Snippet demonstrating a JSON hijacking attack

The attacker overrides the `Object.prototype.__define__Setter` method for the `‘csrf_token’` object. Once the page is included, the object is loaded and the maliciously overridden object setter is executed:

JSON Hijacking


The page at http://192.168.19.129 says:

3417209384712348927118

Figure 97 - Successful JSON hijacking attempt in Firefox 3.6.28

In this example, the attacker has stolen a CSRF token and displayed it as a popup alert. In a real world situation, the data being served could be much more sensitive and the user would not be alerted of the attack.

Vulnerability Identification

Any server response to an HTTP GET request that contains a valid JSON object request is vulnerable to JSON hijacking. While many modern browsers implement protections against JSON hijacking affected applications should still be considered vulnerable, since the server cannot dictate which user-agents issue requests to the API.

Remediation and Mitigation Techniques

The root cause behind a JSON hijacking attack is a vulnerability to Cross-site Request Forgery (CSRF). CSRF attacks occur when a malicious web application forces a user's browser to create a targeted HTTP request to a third party web application, often with the goal of executing a sensitive transaction. Hence, the countermeasure is to apply CSRF protection as well as:

- Disable the use of the GET HTTP method in JSON API calls.
- Alternatively, insert JavaScript at the beginning of all JSON responses that will cause a hijack script to halt in an infinite loop. For example inserting `for(;;)` into the response data before the JSON array object will prevent hijacking code from being interpreted by a web browser.
    - This requires a client-side mechanism to remove the aforementioned modifications as needed for utilization in client-side features.

Additional Resources

JavaScript - Why does Google prepend while(1); to their JSON responses? - Stack Overflow
- https://stackoverflow.com/questions/2669690/why-does-google-prepend-while1-to-their-json-responses?lq=1

JSON Hijacking - You've Been Haacked
- http://haacked.com/archive/2009/06/25/json-hijacking.aspx/

Is JSON Hijacking still an issue in modern browsers? - Stack Overflow
- https://stackoverflow.com/questions/16289894/is-json-hijacking-still-an-issue-in-modern-browsers

JSON Hijacking

Chapter 12: Security – Microsoft Developer Network

- http://msdn.microsoft.com/en-us/library/hh404095.aspx

Full Disclosure: Gmail JSON Hijacking Attack Technique

- http://seclists.org/fulldisclosure/2010/Oct/199

JSONP Hijacking

Description

JSONP or "JSON with padding" is an antiquated technique used to communicate JSON messages across domains. By design JSONP is used to subvert same-origin policy, as a result any domain can read data served via JSONP. For this reason, JSONP should be considered a security vulnerability, and never used in modern web applications. Existing applications that require cross-domain content should be updated to use HTML5 cross-origin resource sharing (CORS).

JSONP hijacking is an attack similar to CSRF in that an attacker controlled page can force a user-agent to perform authenticated actions against another domain, unlike CSRF, upon successful exploitation a JSONP hijacking vulnerability the attacker can ascertain data within the server response.

Example Attack

A router manufacturer wants users to be able to log into their website and access information about their router configuration. This requires the router manufacturer owned domain to be able to request data from the user's local router located on a separate domain. However, do to the browser's enforcement of the same-origin policy a simple cannot be used, instead a JSONP API is added to the local router, when the local router receives an HTTP request to `/cfg` it produces the following response:

```
HTTP/1.1 200 OK
Content-Type: text/javascript; charset=utf-8

display_cfg({'username': 'admin', 'password': 'hunter2'})
```

The manufacturer's website includes the following code:

```
<script>
function display_cfg(config) {
    …
}
</script>
<script src="router.local/cfg"></script>
```

Because the `<script/>` tag can be used to retrieve content from other domains, this will result in an HTTP request to the router's local webserver and the JSONP response will be sent to the user's browser. Because the manufacture's website has already defined a `display_cfg` function prior to the inclusion tag that issued the JSONP request, upon receiving the response the browser will execute the manufacture defined `display_cfg` function, with the JSONP object as an argument. This allows script executing within the context of the manufacture's website to access data returned from the local domain.

However, if the user browses to an attacker-controlled domain that contains a `display_cfg` function and identical `<script/>` tag, the same JSONP response will be evaluated in the context of the attacker's domain.

JSONP Hijacking

Vulnerability Identification

All JSONP content is vulnerable to JSONP hijacking, however depending on the nature of the data returned by the application the risk of such a vulnerability varies greatly.

Remediation and Mitigation Techniques

JSONP is not a secure method of communicating data between origins; replace all JSONP functionality with HTML5 CORS technologies, and follow any applicable CORS security best practices.

Additional Resources

Full Disclosure: CVE-2012-5649 Apache CouchDB JSONP arbitrary code execution with Adobe Flash

- http://seclists.org/fulldisclosure/2013/Jan/82

Cross-Origin Resource Sharing (CORS)

Description

Cross-Origin Resource Sharing (CORS) is a new feature of HTML5. It allows resources from one domain to be requested from a different domain. It is a way to extend the trust relationship for one domain to other domains. If the CORS policy for an application is too permissive, an attacker can act as a man in the middle for that website, attacking its users.

Example Attack

This example involves a bank that has an overly permissive CORS policy. An example request to test CORS policy of the bank is given below:

```
GET / HTTP/1.1
Host: vulnerablebank.com
Origin: http://attacker.net
```

Figure 98 - `CORS policy` test request

The response from the server indicates that the CORS policy is too permissive, allowing CORS request to come from any domain, as shown below:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET, OPTIONS
```

Figure 99 - Insecure CORS policy

After authenticating with the bank, the victim visits other pages, and requests an attacker-controlled webpage. The attacker has crafted a CORS request using JavaScript the victim loads it into their browser when the user visits the malicious site:

```
var xmlhttp;
if (window.XMLHttpRequest)
{
  xmlhttp=new XMLHttpRequest();
}
else
{
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
}
xmlhttp.open("GET","http://vulnerablebank.com/transfer.jsp",false);
xmlhttp.send();
if(xmlhttp.status==200)
{
var str=xmlhttp.responseText;
var n=str.search("csrfToken");
var token=str.substring(n+18,n+28);
var url = "http://vulnerablebank.comp/transfer?txID%2F=05-13-
2011&amp;Account=48752&amp;csrfToken=" + escape(token);
xmlhttp.open("GET", url, true);
xmlhttp.send();
```

Figure 100 - CORS request downloaded from the malicious site

Cross-Origin Resource Sharing (CORS)

Because the bank's CORS policy is too permissive, the server accepts the request and processes it with the permissions of the authenticated user. The user's browser returns the results of the request to the attacker's server:

```
HTTP/1.1 200 OK
Date: Mon, 05 May 2014 01:06:43 GMT
Access-Control-Allow-Origin: http://attacker.net
Access-Control-Allow-Credentials: true

<html>
From Acct: 48752
To Acct: 38729
Amount: $45.33
</html>
```

Figure 101 - Sensitive information passed to attacker using CORS

Vulnerability Identification

Black Box Review

Check the CORS policy by sending an example request to an external website and set the `Origin` parameter pointing to the target domain:

```
GET / HTTP/1.1
Host: target-domain.com
Origin: http://external-domain.net
```

Figure 102 - `CORS policy` test request

If the site is vulnerable to CORS attacks, the `Access-Control-Allow-Origin` parameter in the server's response will be a site you can control:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
```

Figure 103 - Insecure CORS policy

In this case, the wildcard character indicates that requests can be submitted from any domain.

White Box Code Review

Inspect the CORS policy in your server's configuration files. Detailed examples are given below in the Remediation and Mitigation Techniques section.

Remediation and Mitigation Techniques

Your server's CORS policy must be set to allow CORS request from the minimum set of domains that are needed for your application to function properly. Keep in mind that every allowed CORS domain essentially adds that domain to your application's CORS policy. Security vulnerabilities on those domains have the potential to affect your domain as well. Examples on how to enable CORS on a variety of server technologies is given below:

Cross-Origin Resource Sharing (CORS)

Apache

In `Apache.conf`:

```
Header set Access-Control-Allow-Origin "secure.trusted-domain.net"
Header set Access-Control-Allow-Methods "GET, POST, etc..."
```

Figure 104 - Setting Apache CORS policy

ASP.NET

In any resources which require CORS:

```
Response.AppendHeader("Access-Control-Allow-Origin", "secure.trusted-domain.net");
Response.AppendHeader("Access-Control-Allow-Methods", "GET, POST, etc...");
```

Figure 105 - Setting ASP.NET CORS policy

This setting can be used to allow CORS to ASP.NET resources only, rather than to all of IIS.

IIS 7

Using these `web.config` settings will set the CORS policy for all IIS web technologies:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
 <system.webServer>
   <httpProtocol>
     <customHeaders>
       <add name="Access-Control-Allow-Origin" value="secure.trusted-domain.net" />
       <add name="Access-Control-Allow-Methods" value="GET, POST, etc..." />
     </customHeaders>
   </httpProtocol>
 </system.webServer>
</configuration>
```

Figure 106 - Setting IIS 7 CORS policy

PHP

Add these lines to the top of any PHP CORS-enabled resources:

```
<?php
 header("Access-Control-Allow-Origin: secure.trusted-domain.net");
 header("Access-Control-Allow-Methods: GET, POST, etc...");
```

Figure 107 - Setting PHP CORS policy

Additional Resources

enable-cors.org - Examples of how NOT to enable CORS

- http://enable-cors.org/server.html

Mozilla Developer Network - HTTP Access Control (CORS)

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

Cross-Origin Resource Sharing (CORS)

OWASP - Test Cross Origin Resource Sharing

- https://www.owasp.org/index.php/Test_Cross_Origin_Resource_Sharing_%28OTG-CLIENT-002%29

Attack and Defense Labs - Shell of the Future

- http://www.andlabs.org/tools.html#sotf

Demystifying HTML5 Attacks

- http://resources.infosecinstitute.com/demystifying-html-5-attacks/

WebSocket Security

Description

The WebSocket protocol is a feature released in HTML5 that defines how servers and browsers can negotiate a bidirectional TCP connection. This new protocol does not obey the same-origin policy or Cross-Origin Resource Sharing (CORS), and if not secured, can be vulnerable to rogue WebSocket connections from malicious domains.

Example Attack

WebSockets can be spawned from rogue domains if the origin of the connection is not validated. Origin is a header sent by the browser generally when performing cross-site requests, which specifies the location the request originated from. The following code excerpt illustrates a chat room implemented with the Jetty WebSocket framework which fails to validate the origin of the remote connection:

```
public class WebSocketExample implements WebSocketListener {

    private Session outbound;

    @Override
    public void onWebSocketBinary(byte[] payload, int offset, int len) {
    }

    @Override
    public void onWebSocketClose(int statusCode, String reason) {
        this.outbound = null;
    }

    @Override
    public void onWebSocketConnect(Session session) {
        this.outbound = session;
    }

    @Override
    public void onWebSocketError(Throwable cause) {
        cause.printStackTrace(System.err);
    }

    @Override
    public void onWebSocketText(String message) {
        if ((outbound != null) && (outbound.isOpen())) {
            try {
                JSONObject json = (JSONObject) JSONSerializer.toJSON( message );
                receiver = json.getString( "receiver_id" );
                message = json.getString( "message" );
                send_message( receiver, message );
                outbound.sendMessage( "MESSAGE_SENT_OK" );
            } catch (IOException e) {
                // Error out
            }
        }
    }
}
```

Figure 108 - Vulnerable WebSocket server code

WebSocket Security

The above code receives a JSON object over a WebSocket connection through the `Messenger` method in the Jetty framework. Upon receiving the JSON object, the `receiver_id` is parsed along with the `message` and is sent to users in the chat room. Due to the lack of `Origin` validation, a rogue WebSocket could be spawned from an attacker in order to send unauthorized chat messages as an authenticated user in a manner that is similar to CSRF attacks.

Remediation and Mitigation Techniques

The following section covers how to mitigate common WebSocket security risks, including `Origin` verification, token-based authentication schemes, and transport encryption.

Origin Verification

When implementing a WebSocket endpoint, the `Origin` value should be checked to ensure it is from a trusted domain. While this will not protect the server from clients that directly spoof the header value, it will prevent browser-based script from interacting with the service. Since the `Origin` header is set by the browser and not modifiable by JavaScript it can be safely used to validate where a request has originated from. The following code snippet demonstrates verification of the `Origin` header in PHP:

```php
<?php
if( isset( $_SERVER['HTTP_ORIGIN'] ) && $_SERVER['HTTP_ORIGIN'] ==
"http://example.com" ) {
    // Origin is valid
} else {
    // Origin is not set or is not valid
}
```

Figure 109 - `Origin` validation in PHP

The above snippet will prevent rogue requests from being performed from untrusted origins. However, validation by just checking client cookies or other header data is problematic, as that data is untrusted, and can be spoofed easily outside of the browser environment. To address this problem, developers often implement a token-based authentication scheme to protect the WebSocket endpoint.

Token-Based Authentication

A token-based system is often implemented to compensate for the absence of authentication controls in the WebSocket protocol. While there are many approaches to take, such a system might possess the following characteristics:

- The client requests an authentication token over HTTP to later send when creating the WebSocket connection.
- The server securely generates a random token, associates it with the authenticated client session, and records the token value in a server-side persistent store.
- The generated token is returned to the client.
- The client spawns a WebSocket connection to the server and sends the token.
- The server checks the token against the persistent store and associates the client session with the WebSocket connection if the tokens match.

WebSocket Security

- The WebSocket session has been properly authenticated and no further verification is required, the token only needs to be sent once due to the TCP connection being persistent.

The following is an example of WebSocket token-based authentication implemented in the PHP-WebSockets framework:

```
require_once('./websockets.php');

class AuthenticationServer extends WebSocketServer {
  $this->authenticator = new Authentication();

  protected function process ($user, $message) {
      if( $this->authenticator.is_authenticated === true ) {
        // Process user data
      }
  }

  protected function connected ($user) {
    $this->authenticator->checkWebSocketCode( $message );

    if( $this->authenticator->isValid() ) {
        $this->send( $user, 'Authentication completed!' );
    } else {
        $this->send( $user, 'Authentication failed!' );
    }
  }

  protected function closed ($user) {
      $this-authenticator->logout();
  }
}

$echo = new AuthenticationServer("0.0.0.0","9000");

try {
  $echo->run();
}
catch (Exception $e) {
    $echo->stdout($e->getMessage());
}
```

Figure 110 - WebSocket authentication code using the PHP-WebSockets framework

The `AuthenticationServer` method is used to read and write data to the spawned WebSocket connection. Once a WebSocket connection is received, the `connected` method is called, the sent `$message` variable then is checked for a valid authentication token. If the token is valid, the user is authenticated and no further verification is required.

The following proof-of-concept client code could be used to communicate with the server code shown above:

```
<!DOCTYPE html>
<html>
   <head>
      <script type="text/javascript">
```

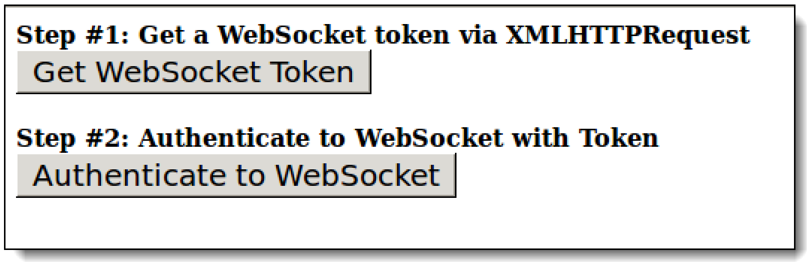
WebSocket Security

```
         var socket = null;
         var isopen = false;
         var token = "";

         window.onload = function() {

            socket = new WebSocket("ws://127.0.0.1:9000");
            socket.binaryType = "arraybuffer";

            socket.onopen = function() {
               console.log("Connected!");
               isopen = true;
            }

            socket.onmessage = function(e) {
               console.log("Text message received: " + e.data);
            }

            socket.onclose = function(e) {
               console.log("Connection closed.");
               socket = null;
               isopen = false;
            }
         };

         function GetWebSocketToken() {
            var request = new XMLHttpRequest();
            request.open("GET", "/getWebSocketToken/", false);
            request.send();
            token = request.responseText;
         }

         function WebSocketAuth() {
            if ( isopen && token !== "" ) {
               socket.send( token );
            } else {
                alert( "We need to get a valid token first!" );
            }
         };
      </script>
   </head>
   <body>
    <b>Step #1: Get a WebSocket token via XMLHTTPRequest</b><br />
    <button onclick='GetWebSocketToken();'>Get WebSocket Token</button><br /><br />
    <b>Step #2: Authenticate to WebSocket with Token</b><br />
    <button onclick='WebSocketAuth();'>Authenticate to WebSocket</button>
   </body>
</html>
```

Figure 111 - Client side portion of WebSocket authentication example

Viewed in a browser, the above HTML form would appear as follows:

WebSocket Security


Step #1: Get a WebSocket token via XMLHTTPRequest

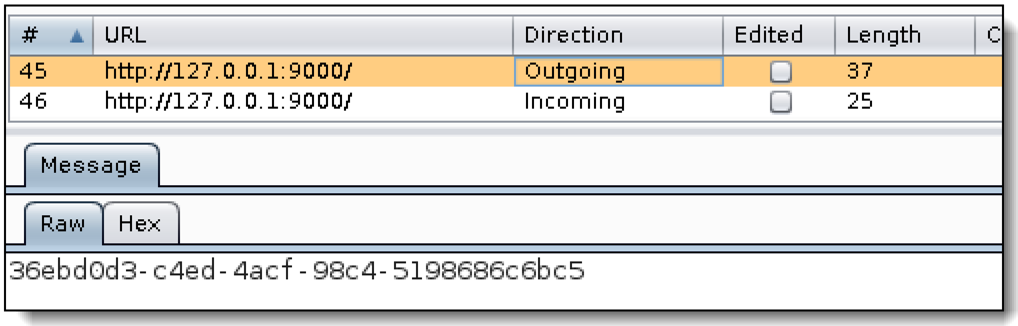
Get WebSocket Token

Step #2: Authenticate to WebSocket with Token

Authenticate to WebSocket

Figure 112 - Screenshot of the example WebSocket snippet

In our example WebSocket authentication scheme, the `GetWebSocketToken` function makes an XMLHTTPRequest to `/getWebSocketToken/` in order to obtain a valid token for the WebSocket session. The server associates this token with the authenticated client session, and stores it in the backend database. The following screenshot taken in the Burp Web proxy software shows the token that is sent from the client to the server in order to authenticate the WebSocket connection:



#	URL	Direction	Edited	Length	C
45	http://127.0.0.1:9000/	Outgoing	☐	37	
46	http://127.0.0.1:9000/	Incoming	☐	25	

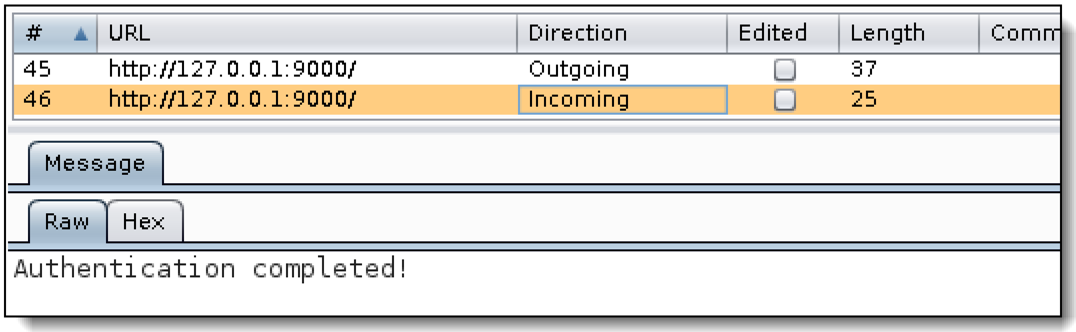
Message

Raw Hex

```
36ebd0d3-c4ed-4acf-98c4-5198686c6bc5
```

Figure 113 - The authentication token being sent over the established WebSocket TCP connection

Once the token has been received, the server checks for it in the database. If the token is valid, the server will associate the authenticated client session with the WebSocket connection.



#	URL	Direction	Edited	Length	Comm
45	http://127.0.0.1:9000/	Outgoing	☐	37	
46	http://127.0.0.1:9000/	Incoming	☐	25	

Message

Raw Hex

```
Authentication completed!
```

Figure 114 - Successful authentication message being sent to client via WebSockets

As the token matched, the WebSocket is successfully established and further communications can commence. In the event of a failure to find a match, the server would be responsible for

WebSocket Security

properly disconnection and disposing of the WebSocket connection.

WSS vs WS

WSS, or "WebSockets over SSL/TLS," is the equivalent of HTTPS for WebSocket connections. In order to prevent Man-in-the-Middle attacks, use of WSS is strongly encouraged for any application that transmits sensitive information. As with HTTPS, if TLS/SSL is already configured on the corresponding application server, a simple connection to `wss://yourdomain.com/` is usually enough to establish a secure WSS connection. When developing technologies that use WebSockets, developers should carefully consider the implications of WSS vs WS, and decide upon the appropriate level of transport security.

Additional Resources

Cross-Site WebSocket Hijacking (CSWSH)

- http://www.christian-schneider.net/CrossSiteWebSocketHijacking.html

WebSocket Security | Heroku Dev Center

- https://devcenter.heroku.com/articles/websocket-security

Autobahn Framework

- http://autobahn.ws/

Web Messaging

Description

Web Messaging creates a simple protocol for securely transferring data between separate domains, commonly referred to as cross-origin communication. For example, Web Messaging can be used by a parent page hosted on `example.com` to communicate directly with an iFrame that is sourced to domain `example-framed.com`, despite the fact that these two domains normally would not be allowed to exchange messages directly due to the same-origin policy. Web Messaging allows for a more simplified data transfer mechanism between distinct domains, but can also cause potential security issues if incorrectly implemented.

Example Attack

The follow section describes three security concerns related to the failure to properly restrict Web Messaging origin values. They are the failure to specify a sender origin, the use of a wildcard target origin when sending messages, and the improper handling of received messages.

The following code snippet receives user input from a text box and sends it via Web Messaging to a child iFrame that sources an external domain:

```
<html>
<head>
   <title>Web Messaging Example</title>
</head>
<body>
   <b>Web Message: </b>:
   <input type="text" style="width: 300px" id="input">
   <button onclick="clickHandler()">Send</button><br /><br />
   <iframe src="http://iframe-domain.com/receiver" id="frame" style="width:
500px; height: 500px;" >
   </iframe>
   <script>
      var iframe = document.getElementById( "frame" );
      var clickHandler = function() {
         iframe.contentWindow.postMessage(
                                    document.getElementById("input").value, "*");
         }
    </script>
</body>
</html>
```

Figure 115 - Vulnerable Web Messaging Sender script

The above snippet uses the Web Messaging specific `postMessage` function to send a message (some data) to the `iframe-domain.com/receiver` page. The end receiving this message on domain `iframe-domain.com` might be implemented as follows:

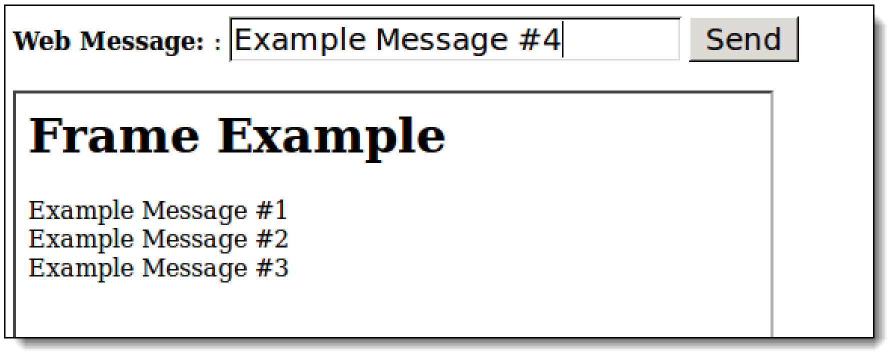
```
<html>
   <h1>Frame Example</h1>
   <script>
   var messageEventHandler = function( event ) {
```

Web Messaging

```
        var messages_div = document.getElementById( "messages" );

        messages_div.innerHTML += event.data;
    }

    window.addEventListener('message', messageEventHandler, false);
    </script>
    <div id="messages"></div>
</html>
```

Figure 116 - Vulnerable Web Messaging receiver script

The above snippet adds an event listener for the `messageEventHandler` to receive any sent Web Messages, regardless of the domain that issued those messages. The following is the combined Web page loaded in a normal browser:



Example Message #1
Example Message #2
Example Message #3

Figure 117 - Example cross-document messaging page

As the example script does not validate the origin of the received message, any domain can embed the `iframe-domain.com/receiver` page on domain `iframe-domain.com` into an iFrame, and send arbitrary messages to it. Additionally, because the receiver frame adds the message to the `div` element `innerHtml` attribute, the example script is also vulnerable to a XSS attack. For example, if a rogue domain issued a Web Message with the content of:

```
<iframe src="javascript:alert('XSS')"></iframe>
```

The receiving document would execute the JavaScript leading to DOM-based XSS as shown in the following screenshot:

Web Messaging

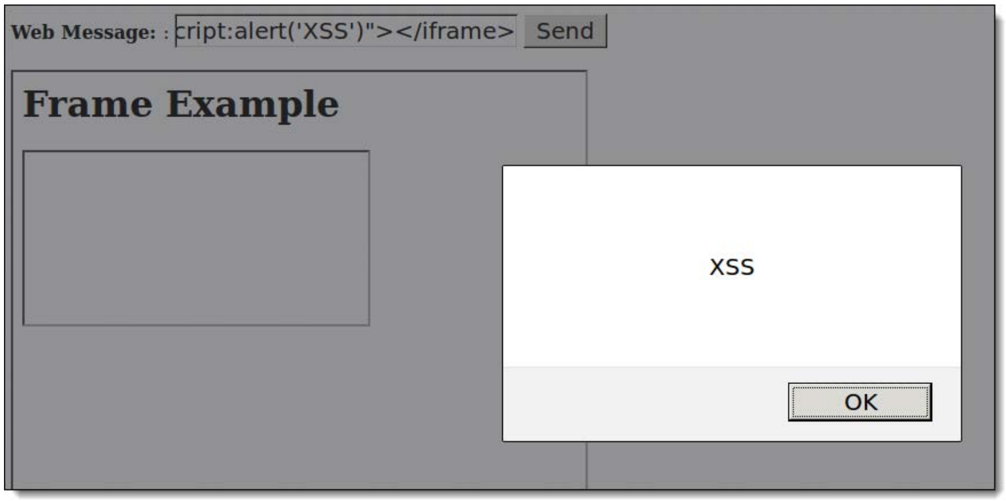


Figure 118 - XSS via Web Messaging

The XSS vulnerability has occurred because the receiving document has directly evaluated JavaScript from the parent page. It should be noted that even if the sending document is trusted, no direct evaluation should be performed due to the risk of a XSS vulnerabilities transferring from the sender document to the receiver page.

In the event of a real-world vulnerability such as the one shown above, an attacker would have the ability to execute arbitrary client-side script in the context of the receiving domain. This could lead to the compromise of active user sessions, the execution of sensitive application functionality, unauthorized disclosure of sensitive data, and numerous other negative consequences.

Remediation and Mitigation Techniques

The following section covers remediation techniques for preventing Web Messaging related vulnerabilities.

Verify Sending Document Origin

When implementing applications that use Web Messaging, whitelists of trusted domains should be used to restrict the domains that are allowed to send messages. The following example illustrates how code that receives Web Messages can be implemented to restrict messages to a single trusted domain:

```
var messageEventHandler = function( event ) {

   if( event.origin == 'http://example.com' ) {
            var messages_div = document.getElementById( "messages" );

      messages_div.innerHTML += event.data;

      window.addEventListener('message', messageEventHandler, false);

    }
}
```

Figure 119 - Properly validated Web Message origin

As can be seen above, the `event.origin` attribute is restricted to a single origin (highlighted in yellow), preventing unauthorized domains from interacting with the Web Messaging code.

Strict Sending Without Wildcards

The side receiving Web Messages is not the only side that must consider the implications of a failure to specify the target origin. The sender too must be concerned, as messages that are not sent to a specific origin, can be intercepted by malicious script executing in other domains. For example, the vulnerable code above used a wildcard to send the Web Message to all child iFrames in the document. Sending a cross-domain message in this way is dangerous because malicious script operating in other domains can intercept the message, and any sensitive data it contains. To mitigate this risk, the sender should specify the target origin that is to receive the message when calling the `postMessage` function:

```
var iframe = document.getElementById( "frame" );
var clickHandler = function() {

    iframe.contentWindow.postMessage(
        document.getElementById("input").value,
        'http://example.com' );

}
```

Figure 120 - Properly validated Web Message target

The `postMessage` function above is called with the target origin set to `http://example.com/`, thereby protecting the corresponding message contents from access by unauthorized script executing in other domains.

Do Not Directly Evaluate Web Messages

Even if a Web Message is received from a trusted domain, the message should not be directly evaluated as it could result in a XSS vulnerability in the receiving domain. As seen in the above attack example, XSS vulnerabilities can spread across domains through Web Messages if the message content is directly evaluated or otherwise used incorrectly. The following is an example of a safely handled Web Message:

```
var messageEventHandler = function( event ) {

   if( event.origin == 'http://example.com/sender' ) {
            var messages_div = document.getElementById( "messages" );

      messages_div.innerText += event.data;

      window.addEventListener('message', messageEventHandler, false);

    }
}
```

Figure 121 - Safe handling of web message content

The above example appends the received Web Message to the `message_div` via the

Web Messaging

`innerText` attribute, which is not interpreted as HTML when displayed in client browsers. This is a safe way to handle incoming web messages and is not vulnerable to a XSS attack.

Additional Resources

Securing Frame Communication in Browsers

- http://seclab.stanford.edu/websec/frames/post-message.pdf

HTML5 Web Messaging - Security

- http://www.w3.org/TR/webmessaging/#security-postmsg

HTML5 Security Cheat Sheet - OWASP

- https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#Web_Messaging

Server-Sent Events

Description

Server-Sent Events (SSE) allow browsers to stream updates from a Web source continuously. SSE differs from WebSockets in that they are not bidirectional but rather allow servers to push one-way updates to the browser. Use cases include stock tickers, live news feeds, and other one-way live update services.

Example Attack

Server sent events can be vulnerable if the streamed content is evaluated directly at the client side. This is attack pattern is similar to the early XSS attacks noted against popular RSS feed clients. Additionally, if user input is not validated additional events could potentially be injected.

The following code snippet opens up an SSE session and streams content from the server. The data received from the server is then dynamically appended to the current document:

```
<html>
    <head>
        <title>Server Side Event Example</title>
    </head>
    <body>
        <div id="content">
        </div>
        <script>
            if( !!window.EventSource ) {
                var source = new EventSource( 'stream.php' );
            } else {
                alert('Browser does not support SSE' );
            }

            source.addEventListener('message', function(e) {
                var new_data = document.createElement('span');
                new_data.innerHTML = e.data;
                document.getElementById( "content" ).appendChild( new_data );
            }, false);

            source.addEventListener('open', function(e) {
                console.log( 'Connection has been established' );
            }, false);

            source.addEventListener('error', function(e) {
                if (e.readyState == EventSource.CLOSED) {
                    console.log( 'Connect has been closed' );
                }
            }, false);

            source.addEventListener('logout_users', function(e) {
                alert('The system is currently down for maintenance, you will now be
logged out.');
                clear_user_cookies();
                logout();
            }, false);
        </script>
    </body>
</html>
```

Server-Sent Events

Figure 122 - Vulnerable Server-Sent events implementation

The above code adds event listeners for the `open`, `error`, `message`, and `logout_users` events. These events will be fired depending on the state of the SSE session. Listeners are declared for each event that will be streamed by the server. For example, if the server streamed the following message:

```
event: logout_users
```

The pre-declared event `logout_users` will be fired and the user will be logged out. This allows the server to perform actions on a multitude of clients by streaming the users commands. The following code sample shows the server side code for streaming content to the end users:

```
<?php
date_default_timezone_set("America/New_York");
header("Content-Type: text/event-stream\n\n");

$user_stream = new UserStream( $_SESSION['id'] );
while (1) {
    $user_stream->update();
    foreach( $user_stream->updates as $msg ) {
        echo "data: ".$msg."\n\n";
    }
    ob_flush();
    flush();
    sleep(1);
}
```

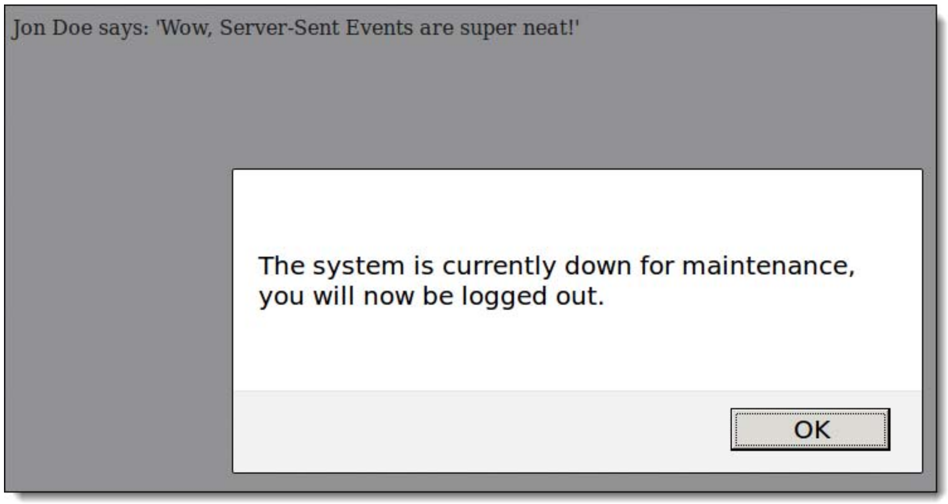
Figure 123 - Server side code snippet

The above code declares the appropriate headers and creates a `UserStream` object, which takes the user ID that is to receive a list of social updates. These messages are then updated and pushed continuously to the target user. The above code is vulnerable to Server-Sent Event injection due to a lack of validation of user submitted messages. If a malicious user were to submitted a crafted message to the stream such as the following:

```
Example Message\n\nevent: logout_users
```

The attacker could trigger the `logout_users` event, thereby forcefully logging out all users who streamed the attacker message:

Server-Sent Events



Jon Doe says: 'Wow, Server-Sent Events are super neat!'

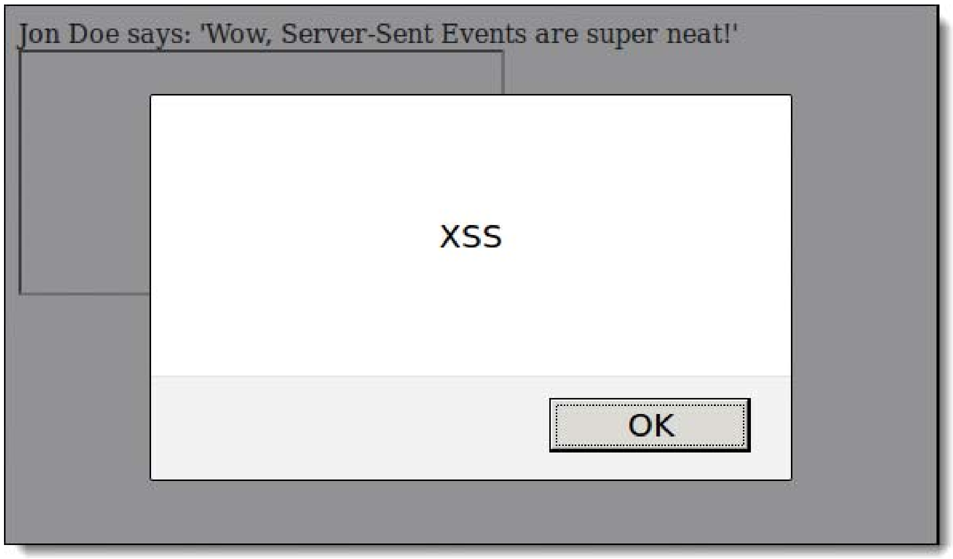
The system is currently down for maintenance, you will now be logged out.

Figure 124 - Server-Sent Event Injection

For this reason it is important to escape all newlines and sanitize all user input in order to prevent the injection of rogue `event:`, `data:`, and other event strings. The same should be done for any input that contains JavaScript in order to mitigate the risk of XSS via SSE. For example, if a malicious user input:

```
Test<iframe src="javascript:alert('XSS')"></iframe>
```

The rogue JavaScript would be streamed and executed on all browsers that received the malicious message:



Jon Doe says: 'Wow, Server-Sent Events are super neat!'

XSS

Figure 125 - XSS via Server-Sent Events

The above example demonstrates how directly evaluating content from an SSE stream can result in an XSS vulnerability.

Server-Sent Events

Remediation and Mitigation Techniques

Content streamed via Server-Sent Events should never be evaluated directly on the client side. User-supplied content that is incorporated into a server stream should be validated, and properly encoded to prevent injection of malicious client-side script, and arbitrary SSE events. In particular, it is important that users not be allowed to inject newlines into the stream. If a malicious user injects a rogue `event:` line into the stream, they could potentially fire sensitive events. Finally, the origin of all events should be checked against a whitelist of allowed domains to prevent third party domains from sending malicious messages.

Additional Resources

HTML5 Security Cheat Sheet - OWASP

- https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#Server-Sent_Events

Web Storage

Description

Web Storage allows for a more versatile client side storage solution. Unlike cookies, Web Storage data is not transmitted on every HTTP request but rather is retrieved via a JavaScript `localStorage.getItem` or `sessionStorage.getItem` function call. Conversely, data can be set using the `localStorage.setItem` or `sessionStorage.setItem` function. The `localStorage` object is persistent throughout browser sessions while the `sessionStorage` object is not. Most modern browsers support a storage size of 5MB per domain, allowing a much larger amount of information to be stored on the browser than previously possible with cookies.

Example Attack

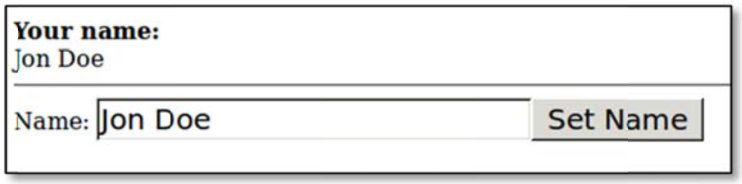
The following section covers how unsafe evaluation of Web Storage content can result in DOM-based XSS vulnerabilities via browser-stored JavaScript. The following snippet takes a user supplied name and stores it, via Web Storage, for later use in the application:

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Web Storage Example</title>
    </head>
    <body>
        <b>Your name: </b>
        <div id="name">
        </div>
        <hr />
        Name: <input type="text" id="input_name" style="width:
300px;"></input><button onclick="set_name()">Set Name</button>

        <script>
            document.getElementById( "name" ).innerHTML = localStorage.getItem(
"name" );

            function set_name() {
                var new_name = document.getElementById( "input_name" );
                localStorage.setItem( "name", new_name.value );
                document.getElementById( "name" ).innerHTML = new_name.value;
            }
        </script>
    </body>
</html>
```

Figure 126 - Vulnerable web storage code

This snippet uses the `localStorage.getItem` function to retrieve stored content and append it to the `name div` element via the `innerHTML` method. The following is an image of this script as rendered in a Web browser:

Web Storage


Your name:
Jon Doe

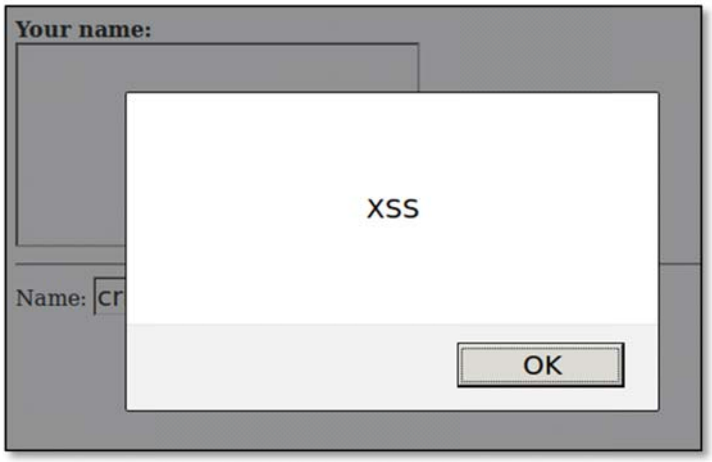
Name: Jon Doe

Figure 127 - Browser view of web storage script

The above document is vulnerable to a XSS vulnerability due to the insecure use of the `innerHTML` attribute, which directly evaluates content. Consider what would happen if a user entered the following input:

```
<iframe src="javascript:alert('XSS')"></iframe>
```

Submitting the above input will result in the user-supplied JavaScript being appended to the `name div` and subsequent execution of said script:


Your name:

Name: cr

XSS

Figure 128 - XSS upon injecting the name input field

However, this is not where the exploitation ends. Due to the page using Web Storage to dynamically create page content, the malicious JavaScript is also executed again upon loading the page:

Web Storage

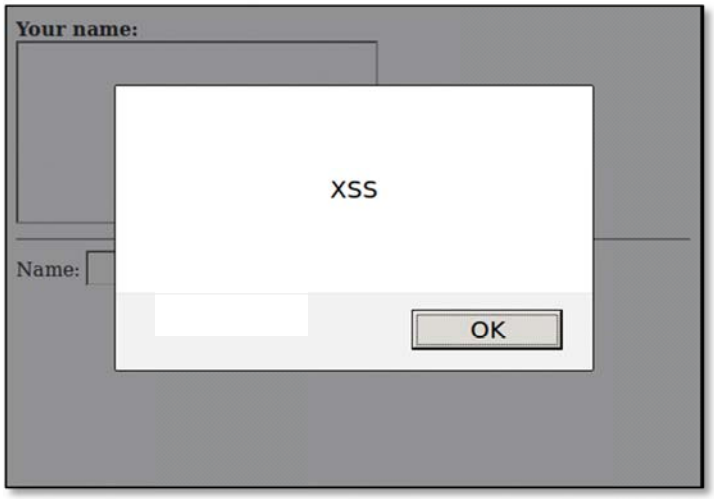


Figure 129 - XSS upon loading the page

This is due to the unsafe use of `innerHTML` on page load in the following code:

```
document.getElementById( "name" ).innerHTML = localStorage.getItem( "name" );
```

This directly evaluates content stored in web storage when the user loads the page. If the initial XSS vulnerability in the name input field was patched the user would still be actively exploited via XSS upon loading the page. As long as the `name` data is stored in the user’s browser, every page visit will execute the malicious JavaScript. For this reason, it is important not to directly evaluate any Web Storage data to prevent stored XSS on the user's web browser.

Remediation and Mitigation Techniques

The following remediation steps cover how to safely use Web Storage, as well as best practice when storing information in the browser.

Web Storage vs Cookies

Web Storage differs from cookies in many respects. When data is stored as a cookie it is sent on every HTTP request to the originating domain. However, Web storage is accessed through JavaScript and does not having a path restriction or any concept equivalent to cookie flags. This means that JavaScript on `example-domain.com` can access all Web Storage data for that domain. Of course, this means that XSS vulnerabilities in a web application can be exploited to read all Web Storage data for the corresponding domain, and write arbitrary values.

Do Not Store Sensitive Data in Web Storage

Sensitive data should not be stored using Web Storage, as it is not meant for secure data storage. Credit card numbers, personally identifiable information (PII) including social security numbers, health information governed by regulations such as HIPAA, are all examples of data

Web Storage

that are not appropriate to store via Web Storage. There are several reasons for this: first, cross-site scripting vulnerabilities in a Web application can be used to access all data in Web Storage. Second, locally stored data will persist even after the browser is closed, increasing the likelihood that the data will eventually be compromised over time. Third, data in Web Storage is not encrypted. Finally, many users access applications from public kiosks, and other shared systems, and data stored via Web Storage may be accessed by malicious actors using or managing those consoles.

Do Not Evaluate Data Stored in Web Storage

Care should be taken when manipulating data stored in Web Storage in the browser. Data that are influenced by users could contain malicious script that, if unsafely evaluated in the browser, could result in script injection vulnerabilities (i.e. XSS). For example, consider an application that stores short messages from various users in Web Storage. If the application displays those messages in an insecure fashion, a malicious user could inject arbitrary script into other users' stores, and have that script exfiltrate all messages from all users, or perform other malicious actions. All untrusted data must be securely validated, and if handled by dynamic script, done so in way that is safe, and will not result in insecure evaluation of that script.

Additional Resources

Web Storage – W3

- http://dev.w3.org/html5/webstorage/#security-storage

Web Storage Security | WhiteHat Security Blog

- http://blog.whitehatsec.com/web-storage-security/

HTML5 Security Cheat Sheet - OWASP

- https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#Local_Storage

Geolocation

Description

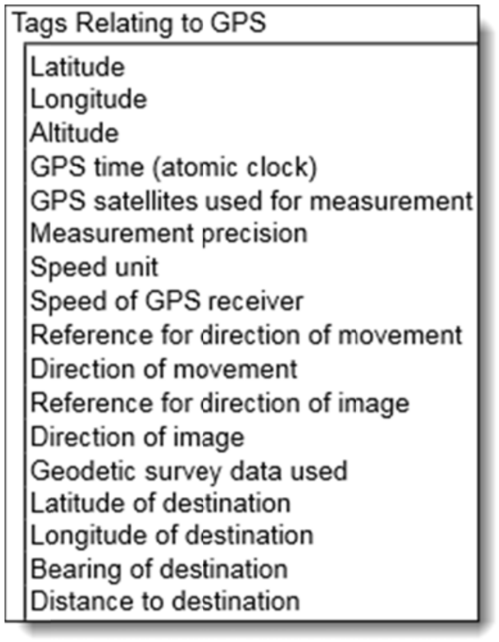
The proliferation of mobile devices has resulted in a novel vulnerability: the geolocation vulnerability. Geolocation vulnerabilities are a form of sensitive information disclosure; however, disclosure of geolocation data deserves special consideration because it affects the physical security of a user. Attackers that exploit geolocation vulnerabilities are usually not trying to compromise data, but rather are trying to compromise physical possessions, collect information to be used for blackmail, or in extreme circumstances, even commit acts of physical violence.

Example Attack

Geolocation vulnerabilities occur in many forms as geolocation data can be included in requests to other websites, automatically included in social networking status updates, or broadcast with wireless traffic. For this document, one of the most widespread vulnerabilities has been selected as a case study: geolocation data included in file metadata.

EXIF Data

The Exchangeable Image File Format (EXIF) is a metadata file format that is widely used by mobile devices. EXIF can be applied to a wide variety of file types including JPEG, TIFF, and WAV. Applications that use the EXIF libraries to insert metadata into files should be aware of the wealth of information an attacker could obtain from a picture shared on the Internet. Below is an excerpt from the EXIF 2.2 Specification that details some of the supported geolocation data that an attacker can obtain from image, audio, and video files created with a smartphone:



Tags Relating to GPS
Latitude
Longitude
Altitude
GPS time (atomic clock)
GPS satellites used for measurement
Measurement precision
Speed unit
Speed of GPS receiver
Reference for direction of movement
Direction of movement
Reference for direction of image
Direction of image
Geodetic survey data used
Latitude of destination
Longitude of destination
Bearing of destination
Distance to destination

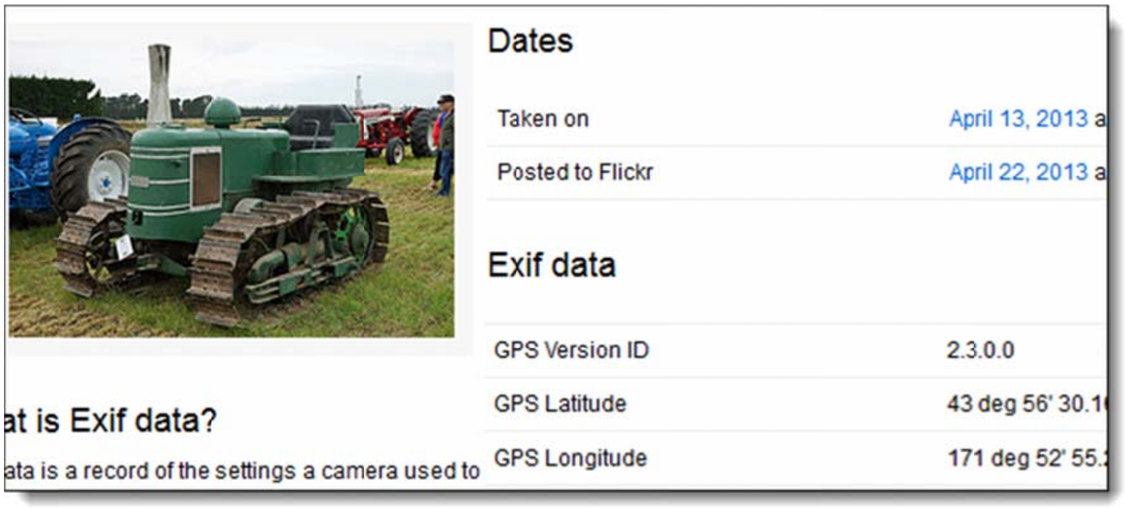
Figure 130 - Supported GPS data in EXIF spec 2.2

There is a trade-off when considering whether to enable these features in an application.

Geolocation

Clearly, this data has value to users, but it also has value to investors, developers, and advertisers. The data collected are not so dangerous that they should be discarded for 100% of all usage cases. However, as a developer it is strongly recommended that geolocation data be handled with great care, as it can be valuable in a variety of side-channel attacks.

For example, consider the consequences of allowing employees with high security clearances to use a social media site such as Flickr. Flickr encourages its users to post frequently, and to make photos available to the public. Flickr also does not strip EXIF data from its images, and even provides an API to view the EXIF data of any posted image with no special tools, as shown below:



Dates

Taken on	April 13, 2013 a
Posted to Flickr	April 22, 2013 a

Exif data

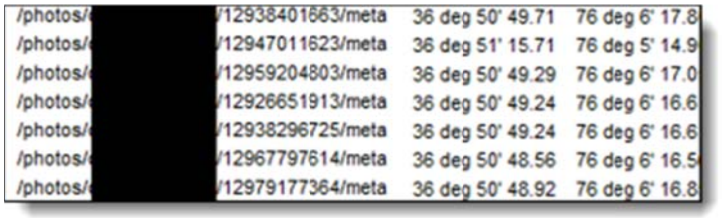
GPS Version ID	2.3.0.0
GPS Latitude	43 deg 56' 30.1
GPS Longitude	171 deg 52' 55.

at is Exif data?

ata is a record of the settings a camera used to

Figure 131 - Using the `/meta/` URL to view EXIF data on Flickr

Flickr does not use any form of effective anti-automation, so large amounts of photographs can be parsed in a very short period:

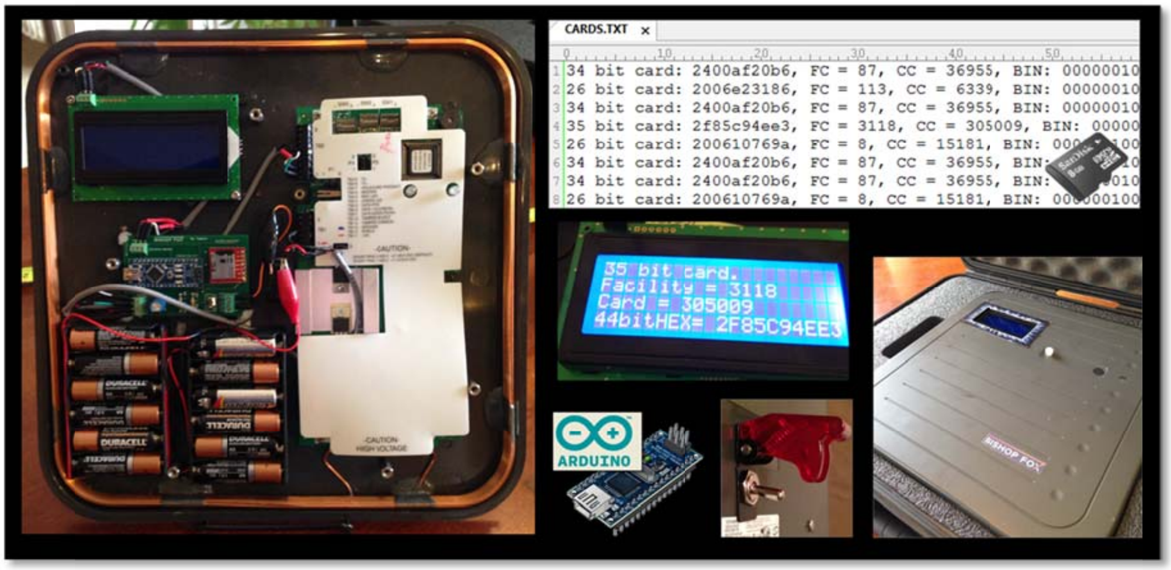


/photos/	[redacted]	/12938401663/meta	36 deg 50' 49.71	76 deg 6' 17.8
/photos/	[redacted]	/12947011623/meta	36 deg 51' 15.71	76 deg 5' 14.9
/photos/	[redacted]	/12959204803/meta	36 deg 50' 49.29	76 deg 6' 17.0
/photos/	[redacted]	/12926651913/meta	36 deg 50' 49.24	76 deg 6' 16.6
/photos/	[redacted]	/12938296725/meta	36 deg 50' 49.24	76 deg 6' 16.6
/photos/	[redacted]	/12967797614/meta	36 deg 50' 48.56	76 deg 6' 16.5
/photos/	[redacted]	/12979177364/meta	36 deg 50' 48.92	76 deg 6' 16.8

Figure 132 - Flickr GPS EXIF data; each record took less than one second to obtain with Burp

Armed with this information, an attacker could analyze the movements and habits of key personnel from a distant location, and intercept those employees to steal sensitive equipment or badge identification data. Tools such as the RFID Thief developed by Fran Brown of Bishop Fox (pictured below) can be used to remotely read RFID badge data for cloning:

Geolocation


CARDS.TXT

```
34 bit card: 2400af20b6, FC = 87, CC = 36955, BIN: 00000010
26 bit card: 2006e23186, FC = 113, CC = 6339, BIN: 00000010
34 bit card: 2400af20b6, FC = 87, CC = 36955, BIN: 00000010
35 bit card: 2f85c94ee3, FC = 3118, CC = 305009, BIN: 00000
26 bit card: 200610769a, FC = 8, CC = 15181, BIN: 000[illegible]100
34 bit card: 2400af20b6, FC = 87, CC = 36955, BIN:[illegible]10
34 bit card: 2400af20b6, FC = 87, CC = 36955, BIN: [illegible]010
26 bit card: 200610769a, FC = 8, CC = 15181, BIN: 00[illegible]00100
```

```
35 bit card.
Facility = 3118
Card = 305009
44bitHEX= 2F85C94EE3
```

Figure 133 - The RFID Thief

While an attacker could use the RFID Thief tool at the target company to steal RFID badge data, the preferred rendezvous is coffee shops, or restaurants the mark often visits.

The above is just one example in which the security of a company or its clients is compromised by geolocation vulnerabilities. Applications that disclose the location of employees working in remote or hostile locations expose them to kidnapping or other physical attack. While physical security is not typically of concern to the web developer, this is a case where decisions made by the developer directly affects the physical security of the application's users.

Black Box Review

Examine data streams on mobile platforms and identify geolocation data use. EXIFTool is a free utility for Windows and Mac that is used to examine images for sensitive EXIF data:

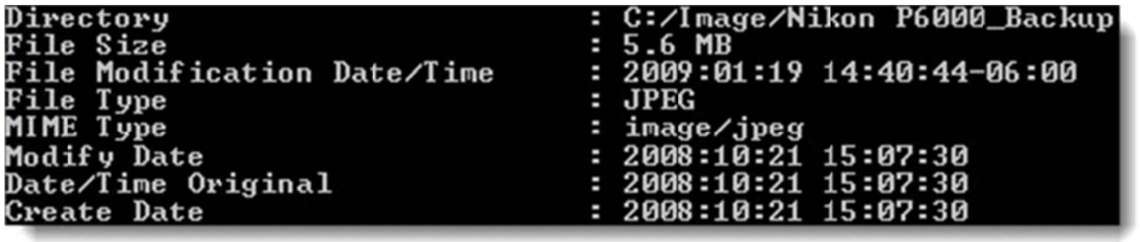

```
Directory                       : C:/Image/Nikon P6000_Backup
File Size                       : 5.6 MB
File Modification Date/Time     : 2009:01:19 14:40:44-06:00
File Type                       : JPEG
MIME Type                       : image/jpeg
Modify Date                     : 2008:10:21 15:07:30
Date/Time Original              : 2008:10:21 15:07:30
Create Date                     : 2008:10:21 15:07:30
```

Figure 134 - EXIFTool reading EXIF data from a picture

During a black box review, photos, sound files, and videos generated by the application should be examined to determine if the application exposes sensitive geolocation data to the public. If it does, consider adding functionality to strip the EXIF data from files before releasing.

Geolocation

White Box Code Review

Read the documentation of file creation libraries and determine if they include EXIF data. If the EXIF data is desired for internal statistical analysis, consider stripping it out of the files before exposing them to the Internet. EXIFTool has wrappers for .NET, PHP, Java and many other languages, and those wrappers should be used to modify EXIF data programmatically.

Remediation and Mitigation Techniques

The following sections demonstrate several techniques and tools that can be used to dynamically remove sensitive geolocation data.

PHP

PHPExifTool is the PHP wrapper for ExifTool. The developers of PHPExifTool recommend using Composer to install their libraries. Once installed, the following commands are used to strip all EXIF data from file `image.jpg`:

```
use PHPExiftool\Writer;
use PHPExiftool\Driver\Metadata;
use PHPExiftool\Driver\MetadataBag;
use PHPExiftool\Driver\Tag\IPTC\ObjectName;
use PHPExiftool\Driver\Value\Mono;

$Writer = Writer::create();
$Writer->erase(true);
$Writer->write('image.jpg');
```

Figure 135 - PHPExifTool erasing the metadata

Java

Unlike PHP, in Java, a wrapper object is not required for manipulating images. Instead, the native `BufferedImage` class can be used to open, write, and then close the image. As metadata is not supported by this Java class, it is not written back to the file:

```
BufferedImage image = ImageIO.read(new File("image.jpg"));
ImageIO.write(image, "jpg", new File("image.jpg"));
```

Figure 136 - Erasing Image EXIF data in Java

ASP.NET

The following ASP.NET class, developed by Mikael Svenson, implements a method for high performance JPEG EXIF data removal. It solves issues the native `JpegEncoder` class has with writing images that will be used cross-platform:

```
using System.IO;

namespace ExifRemover
{
    public class JpegPatcher
    {
```

Geolocation

```
        public Stream PatchAwayExif(Stream inStream, Stream outStream)
        {
            byte[] jpegHeader = new byte[2];
            jpegHeader[0] = (byte)inStream.ReadByte();
            jpegHeader[1] = (byte)inStream.ReadByte();

            //check if it's a jpeg file
            if (jpegHeader[0] == 0xff && jpegHeader[1] == 0xd8)
            {
                SkipAppHeaderSection(inStream);
            }
            outStream.WriteByte(0xff);
            outStream.WriteByte(0xd8);

            int readCount;
            byte[] readBuffer = new byte[4096];
            while ((readCount = inStream.Read(readBuffer, 0, readBuffer.Length)) > 0)
                outStream.Write(readBuffer, 0, readCount);

            return outStream;
        }

        private void SkipAppHeaderSection(Stream inStream)
        {
            byte[] header = new byte[2];
            header[0] = (byte)inStream.ReadByte();
            header[1] = (byte)inStream.ReadByte();

            while (header[0] == 0xff && (header[1] >= 0xe0 && header[1] <= 0xef))
            {
                int exifLength = inStream.ReadByte();
                exifLength = exifLength << 8;
                exifLength |= inStream.ReadByte();

                for (int i = 0; i < exifLength - 2; i++)
                {
                    inStream.ReadByte();
                }
                header[0] = (byte)inStream.ReadByte();
                header[1] = (byte)inStream.ReadByte();
            }
            inStream.Position -= 2; //skip back two bytes
        }
    }
}
```

Figure 137 - ASP.NET EXIF scrubber

Additional Resources

OWASP - HTML5 Security Cheat Sheet

- https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet

Kodak - EXIF 2.2 Standard

- http://www.kodak.com/global/plugins/acrobat/en/service/digCam/exifStandard2.pdf

Bishop Fox - RFID Hacking: Live Free or RFID Hard

- https://www.youtube.com/watch?v=1fszkxcJt7U

Geolocation

ExifTool by Phil Harvey

- http://www.sno.phy.queensu.ca/~phil/exiftool/

XenoActive.org - Removing Image Metadata with ExifTool

- http://www.xenoactive.org/2011/02/removing-image-metadata-with-exiftool/

PHP - EXIFTool Wrapper

- https://github.com/romainneutron/PHPExiftool

Mikael Svenson - Removing EXIF Data in ASP.NET

- http://techmikael.blogspot.com/2009/07/removing-exif-data-continued.html

HTTP Security Headers

Description

HTTP security headers can be deployed to increase web application security against malicious attacks including clickjacking, man-in-the-middle (MITM), and malicious script execution. This section covers the commonly deployed headers that can be used to improve security.

HTTP Strict Transport Security (HSTS)

Using the HTTP strict transport security (HSTS) header is recommended for web applications that can be served exclusively over TLS/SSL. Once a browser receives this header, it will no longer send any further requests over regular HTTP. This will prevent any leaks from Man-in-the-Middle (MITM) attacks, and will prevent users from being able to bypass SSL error prompts. To enable HSTS, the application server issues a header such as the following:

```
Strict-Transport-Security: max-age=31556926
```

Figure 138 - Example Strict Transport header

In the above example, the max-age field of 31556926 seconds, or one year, is used to indicate to the client how long the HSTS directive should be considered valid. This means the client (typically a browser) will only connect over HTTPS for an entire year, provided the client settings are not reset (for example, by clearing the browser settings cache). It is important to understand that HSTS overrides any attempt to connect to the corresponding server over insecure HTTP. For example, should a browser encounter a HTTP link for a site that previously issued an HSTS header the client would ignore the HTTP link and issue the request over HTTPS instead.

If subdomains are also to be covered by HSTS, the `includeSubDomains` flag should be appended to the end of the header. The following is an example header that enforces HTTPS for all subdomains:

```
Strict-Transport-Security: max-age=31556926; includeSubDomains
```

Figure 139 - Example HSTS header for subdomains

To enable the strict transport security header in Apache, edit the `httpd.conf` to include the following:

```
<VirtualHost yourhost.com:443>
       Header always set Strict-Transport-Security "max-age=31556926"
</VirtualHost>
```

Figure 140 - Enabling Strict Transport via `http.conf`

The above setting will enable HSTS for a single domain, and instruct clients to cache the setting for one year (31556926 seconds). The above configuration does not issue the `includeSubDomains` flag.

In nginx you can add HSTS support by adding a custom header in the `nginx.conf` file:

```
add_header Strict-Transport-Security max-age=31556926; includeSubDomains
```

Figure 141 - Enabling strict transport security via `nginx.conf`

The above nginx configuration is identical to the previously shown Apache configuration, except for the presence of the `includeSubDomains` flag. For more information regarding how to properly set HSTS in server technologies other than Apache and nginx, please refer to the appropriate server documentation.

X-Frame-Options

Clickjacking is an attack performed by loading a target site inside of an embedded iFrame, and overlaying the frame with deceptive CSS to trick a user into performing actions on the target site. For this reason, clickjacking is often referred to as a "UI redress attack" because the attacker is "redressing" the victim site with CSS. The malicious overlay will trick the user into clicking on various parts of the screen in order to perform web actions on the “hidden” target site.

A notable example of clickjacking in the real-world involved an attack against the Adobe Flash Player camera and microphone security prompt. Security researcher Guy Aharonovsky discovered that the Adobe Flash player prompts a user before a Flash application is allowed to access the camera and microphone, and that the prompt could be hidden by overlaying it with a deceptive CSS layer. His proof of concept humorously demonstrated this issue by overlaying the security prompts with a game titled "Camera Clickjacking – The Game." The following screenshot shows the proof-of-concept posted by the researcher:

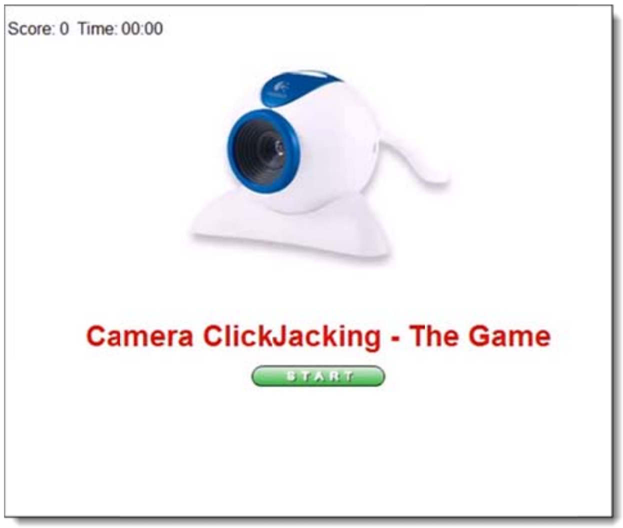
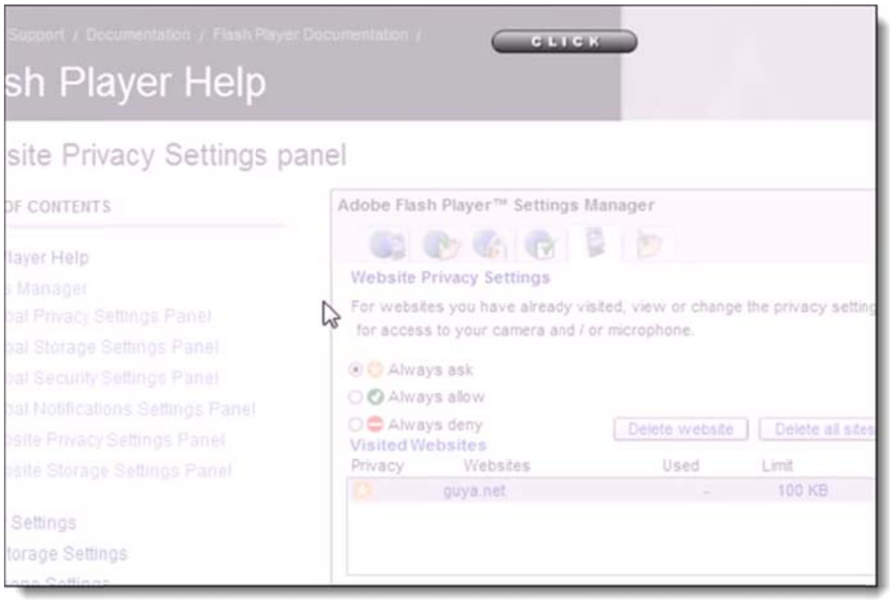


Figure 142 - Flash player camera clickjacking proof of concept

The proof-of-concept presented itself as a simple game where the user would attempt to click on a moving button. However, the user was actually clicking through the various security prompts from flash player and allowing the malicious website to take full control over the client microphone and webcam. The following example, directly taken from the proof-of-concept video released by Aharonovsky, shows the attack with the CSS masking layer set to be slightly transparent:


Support / Documentation / Flash Player Documentation /
CLICK
sh Player Help
site Privacy Settings panel
OF CONTENTS
layer Help
Manager
bal Privacy Settings Panel
bal Storage Settings Panel
bal Security Settings Panel
bal Notifications Settings Panel
site Privacy Settings Panel
site Storage Settings Panel
Settings
torage Settings
Adobe Flash Player™ Settings Manager
Website Privacy Settings
Always ask
Always allow
Always deny
Delete website
Delete all sites
Visited Websites

For websites you have already visited, view or change the privacy setting for access to your camera and / or microphone.

Privacy	Websites	Used	Limit
	guya.net	-	100 KB

Figure 143 - Semitransparent proof-of-concept

As can be seen above, the user interface was redressed using CSS in order to trick the user into clicking on the various security prompts presented by the Flash player. While the proof-of-concept was clearly not designed to be stealthy, it demonstrated the heavy ramifications of clickjacking attacks.

While the above attack illustrates clickjacking as it was used to exploit the Adobe Flash Player, the issue is much broader than just security prompts for cameras and microphones: any site that lacks proper clickjacking protections can be vulnerable to similar issues.

The remediation for clickjacking attacks is straightforward, and can be implemented by simply adding the `X-Frame-Options` header to a Web application. This HTTP header will instruct compliant browsers to not embed the corresponding Web site in an iFrame from untrusted domains, thus making it difficult for an attacker overlay CSS onto the target application.

The following is an example of the `X-Frame-Options` header as it is used to prevent the Web application from being displayed in an iFrame:

```
X-Frame-Options: SAMEORIGIN
```

Figure 144 - Example `X-Fame-Options` header

The `SAMEORIGIN` value instructs the client to only respect iFrame embedding attempts from pages originating from the same application.

To implement the `X-Frame-Options` header, edit the Apache `httpd.conf` to include the following:

```
<VirtualHost yourhost.com:443>
       Header always set X-Frame-Options "SAMEORIGIN"
</VirtualHost>
```

Figure 145 - Enabling X-Frame-Options via `http.conf`

As is made clear above, this sample configuration file will issue an `X-Frame-Options` header set to the value `SAMEORIGIN`.

In nginx you can add support by adding a custom header in the `nginx.conf` file:

```
add_header X-Frame-Options ALLOW-FROM http://whitelisted-domain.com
```

Figure 146 - Enabling X-Frame-Options via `nginx.conf`

Unlike the Apache example, the nginx configuration file specifies that only the domain `whitelisted-domain.com` can embed the web application in an iFrame. It should be noted that major browsers such as Chrome, Opera, and Safari do not yet support the `X-Frame-Options ALLOW-FROM` method. Additionally, support for wildcards and multiple whitelisted domains are not permitted per the RFC specification.

For more information on implementing clickjacking protection in legacy browsers, see the article titled *Clickjacking Defense Cheat Sheet* in the *Additional Resources* section.

Content Security Policy (CSP)

Content Security Policy (CSP) headers allow for more fine-tuned control over what content is allowed to be loaded into a Web application. CSP is used instruct compliant browsers to enforce tighter script execution controls, based on the type of CSP header that is issued. For example, CSP can instruct client browsers to not honor any JavaScript that has not been directly sourced through a script tag. This has the effect of preventing many cross-site scripting attacks that rely on the ability to inject inline JavaScript. CSP headers are set in the following format:

```
Content-Security-Policy: [resource type] [type] http://origin1.com
http://origin2.com
```

First, the resource type is defined as `script-src` for JavaScript, `style-src` for CSS style sheets, or `frame-src` for iFrames. Second, the rule type is set to either `none`, which allows no content of said type to be loaded, or `self`, which will allow CSP exceptions for the originating domain. Finally, a list of space-separated origin values are defined to set domains that content can be loaded from. Domain name wildcards are supported to allow for greater flexibility when specifying the list of permitted origins.

The following CSP example illustrates a policy that will instruct the client to only load JavaScript from the current page and Google Analytics:

```
Content-Security-Policy: script-src 'self' http://google-analytics.com
```

HTTP Security Headers

This policy will only allow JavaScript included from the same domain to execute, even inlined `<script/>` tags included in the page will not execute; a strict CSP can prevent the vast majority of XSS vulnerabilities.

CSP policies can also be stacked by appending a semicolon. For example, if a page should not allow frames or extraneous objects to be loaded, a filter such as the following could be applied:

```
Content-Security-Policy: frame-src 'none'; object-src: 'none'
```

The above policy specifies that content such as iFrames or objects should not be loaded, even if the page contains HTML for these resources. If an attacker injected malicious HTML to load an external iFrame, compliant browsers would refuse to load the injected content thereby thwarting the attack. This allows for an extra layer of protection against XSS and other external resource attacks.

The following resources can be controlled using the CSP headers:

- `connect-src`: Limits what origins scripts can connect to via AJAX, EventSource, and WebSockets technologies
- `font-src`: Limits what origins the page can load fonts from
- `frame-src`: Limits what origins the page can load frames from
- `img-src`: Limits what origins images can be loaded from
- `media-src`: Limits what origins media such as videos and audio can be loaded from
- `object-src`: Limits what origins objects such as flash and other plugins can be loaded from
- `style-src`: Limits what origins CSS content can be loaded from
- `script-src`: Limits what origins JavaScript can be loaded from

The following example shows the advantage of using strict CSP when securing a Web application. In the following code snippet, a Web application page containing malicious JavaScript is loaded inside the document body:

```
<?php
header("Content-Security-Policy: \"default-src 'none'; script-src 'self'
127.0.0.1\"");
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<head>
    <meta content="text/html;charset=utf-8" http-equiv="Content-Type">
    <meta content="utf-8" http-equiv="encoding">
    <title>Anti-JavaScript Page</title>
</head>
<body>
    <h1>No JavaScript Allowed!</h1>
    <div id="content"></div>
    <script>alert( 'XSS' );</script>
    <script src="example.js"></script>
</body>
```

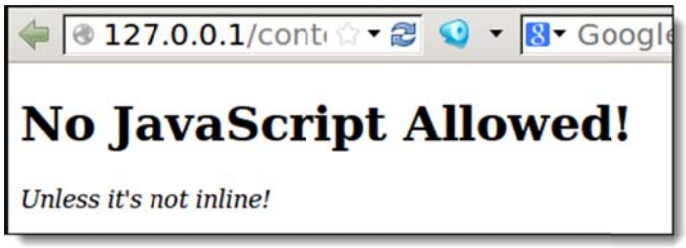
Figure 147 - Example injection of inline JavaScript

The external `example.js` file is also included, which contains JavaScript to change the `content div` upon execution:

```
document.getElementById( "content" ).innerHTML = "<i>Unless it's not inline!</i>";
```

Figure 148 - `example.js`

The PHP `header` function sets a CSP that forbids the use of inline JavaScript on the example page. Upon loading the page, no JavaScript popup is displayed, despite the presence of that code (i.e., `alert('xss');`):

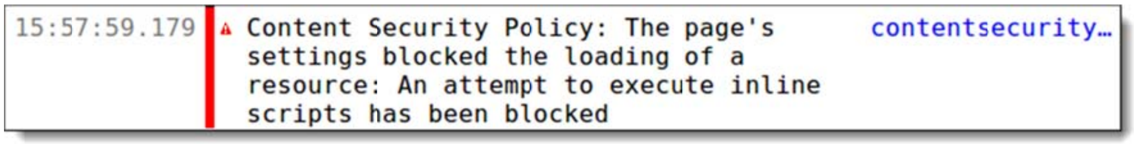


No JavaScript Allowed!

Unless it's not inline!

Figure 149 - Example page

Upon further examination, Firefox has issued a security exception stating that it will not load inline JavaScript, due to the CSP header:



```
15:57:59.179  Content Security Policy: The page's        contentsecurity…
              settings blocked the loading of a
              resource: An attempt to execute inline
              scripts has been blocked
```

Figure 150 - Content security policy message

Unlike the inline JavaScript, the locally sourced script in `external.js` is loaded and executed normally. The CSP has allowed safe non-inline JavaScript while preventing the injected JavaScript from executing. Tightly restricting script execution in this manner prevents the majority of XSS attacks from occurring, as an attacker cannot inject JavaScript into a page directly. The problem of XSS has been effectively mitigated, barring the discovery of a vulnerability that allows an attacker to upload a file directly to the remote server, or modify the contents of an existing JavaScript file. It should be noted that CSP may not protect against certain types of DOM based JavaScript injection attacks.

When creating a CSP for your web application, first determine what resources are required for the Web application to function. Build a whitelist of allowed domains and develop a CSP that allows these resources. It is important to approach CSP with a whitelist strategy, as opposed to a blacklist, in order to create effective policies. For an easy way to generate content security policies, see *Content Security Policy Header Generator* in the *Additional Resources* section.

HTTP Security Headers

Additional Resources

Clickjacking Defense Cheat Sheet - OWASP
- https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet

HTTP Strict Transport Security - OWASP
- https://www.owasp.org/index.php/HTTP_Strict_Transport_Security

Camera ClickJacking - The Game
- http://guya.net/security/clickjacking/game.html

An Introduction to Content Security Policy - HTML5 Rocks
- http://www.html5rocks.com/en/tutorials/security/content-security-policy/

Content Security Policy Header Generator
- http://cspisawesome.com/

Busting Frame Busting a Study of Clickjacking Vulnerabilities on Popular Sites
- http://www.elie.im/publication/busting-frame-busting-a-study-of-clickjacking-vulnerabilities-on-popular-sites#.U2ANh1d-eCQ

OAuth 2.0 Security

Description

OAuth version 2.0 is an open standard used for delegated authorization (IETF RFC #6749), which provides a secure way for a user to issue specific resource access to a third party without revealing their password.

OAuth Transactions Must Be Done Over HTTPS

The OAuth 2.0 RFC specifically states that all OAuth transactions must be performed over a secure connection. If a transaction is done over HTTP instead of HTTPS the bearer token can be intercepted by an attacker and used to perform authenticated actions as the user. Since many APIs require the bearer token to be sent on every request, all transactions should be wrapped in SSL to prevent interception.

Do Not Cache Bearer Tokens

Bearer tokens should never be cached, since they allow access to protected user resources they should be treated and stored with the same level of security used to store user credentials. Transmitting bearer tokens should be done via the `Authorization` or `WWW-Authenticate` HTTP headers as they are not cached by any modern web browsers. This also applies to web servers as they can potentially leak these tokens via web access logs. Ensure that all forms of caching are disabled and follow the secure transmission procedures outline in the *Transport Security* section of this guide.

Secure Storage of Bearer and Refresh Tokens

Due to the sensitive nature of bearer and refresh tokens they should never be stored in plaintext in a backend database. If a web application is vulnerable to SQL injection these tokens could be stolen and used to access user resources. For this reason, all tokens should first be encrypted before being stored in a back end database.

Additional Verification of Client Identity

When performing OAuth authorization a web application should not use only tokens to validate the identity of a user. For additional security a web application should also validate the client's IP address to ensure the token has not been stolen by an attacker. Depending on the nature of the service, this can be a useful step in preventing tokens from being used maliciously by a third party.

CSRF Prevention and OAuth Callback URI Validation

Clients should prevent CSRF attacks on the OAuth callback URI by validating the resource owner intends to allow OAuth access to their resources. This has been a large problem for many services as an attacker will accept the OAuth prompt and use the callback URI in a CSRF attempt to authenticate with another user's account.

One example of a CSRF attack done with OAuth callback URIs was performed by Nir Goldshlager against Facebook's OAuth service. Goldshalger noticed that it was possible to modify the OAuth callback URI to point towards an arbitrary Facebook subdomain through a regular expression bypass trick. The Facebook OAuth URI follows the following format:

```
https://www.facebook.com/dialog/oauth/?app_id=
APP_ID&next=REDIRECT_URL&state=YOUR_STATE_VALUE&scope=COMMA_SEPARATED_LIST_OF_PERMI
SSION_NAMES
```

Figure 151 - Facebook OAuth callback example URL

The `next` parameter was not properly validated and could be bypassed by appending a `%23%09` to the original redirect domain followed by the attacker's domain. The following is an example of a maliciously formed URL to steal the OAuth access token:

```
https://www.facebook.com/connect/uiserver.php?app_id=220764691281998&next=https%3A%
2F%2Ftouch.facebook.com%2F%23%09!%2Fapps%2Ftestestestte%2F&display=page&fbconnect=1
&method=permissions.request&response_type=token
```

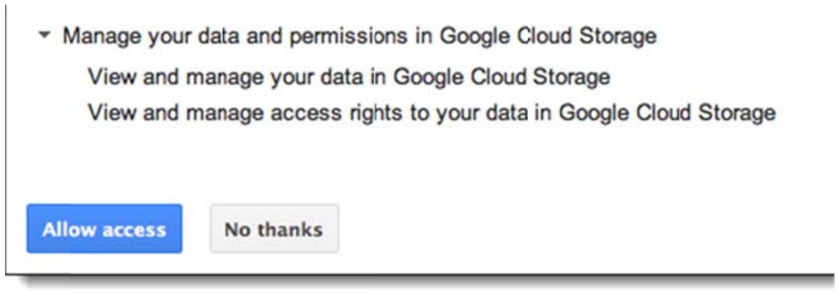
Figure 152 - Facebook OAuth exploit proof of concept

Goldshlager set the `APP_ID` to the internal Facebook messenger application, which the user does not need to accept permissions for. The `next` parameter was spoofed using the regular expression bypass and replaced with an attacker control Facebook page. This Facebook page then took the token and passed it on to Goldshlager's server. This gave Goldshalger full ability to post to walls, delete photos, and access all of the victim's user information.

Flaws of this type are often found in various OAuth frameworks, if an attacker can modify the callback parameter of an OAuth transaction the application is likely vulnerable to CSRF.

Proper Validation of User Consent

When implementing OAuth for a web application it is important that a user is made aware when they are granting access to a third party application. The following is an example OAuth security prompt for Google's cloud storage service:


▾ Manage your data and permissions in Google Cloud Storage

View and manage your data in Google Cloud Storage

View and manage access rights to your data in Google Cloud Storage

Figure 153 - Google OAuth verification prompt

This prompt accurately informs the user of the permissions the user is granting upon clicking the "Allow Access" button. All implementations of OAuth should be careful to inform a user of the full scope of permissions they are giving to a third party. This page should also be secured against clickjacking, CSRF, and other attacks that would take this consent away from the user.

OAuth 2.0 Security

Token Expiration and Revocation

Depending on the nature of the application, giving OAuth access without expiration is not always necessary. Tokens without expiration could be stolen from third parties and used to perform authorized actions on behalf of the user long after the initial access been granted. Token expiration prevents this by forcing a client to refresh their token after a certain period. When combined with IP address restrictions it can be a powerful tool to stop rogue tokens from compromising user security. Revocation of tokens should also be available to remove third party access to the user's resources after they have been granted. In the event a 3rd party is comprised, the ability to revoke all tokens is very important to stop an attacker from re-using the OAuth tokens.

Additional Resources

RFC 6819 - OAuth 2.0 Threat Model and Security Considerations

- http://tools.ietf.org/html/rfc6819

Egor Homakov: The Most Common OAuth2 Vulnerability

- http://homakov.blogspot.com/2012/07/saferweb-most-common-oauth2.html

AppSec Street Fighter - SANS Institute | Four Attacks on OAuth - How to Secure Your OAuth Implementation | SANS Institute

- http://software-security.sans.org/blog/2011/03/07/oauth-authorization-attacks-secure-implementation

Weak Server Side Controls

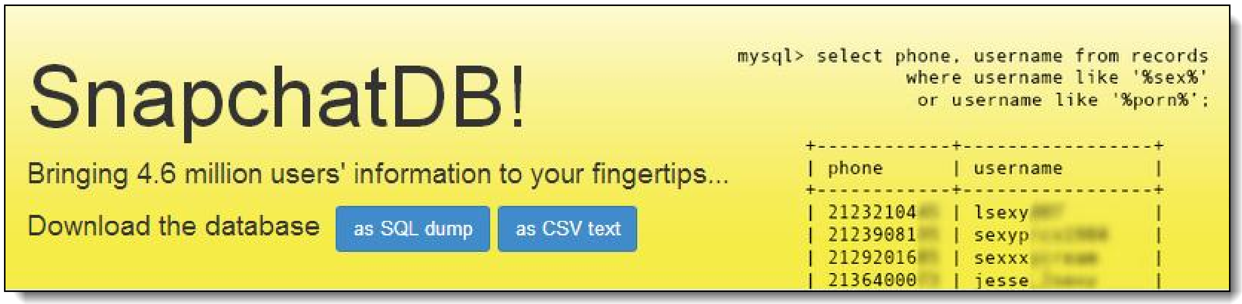
Description

Due to the hidden nature of mobile app communication, many developers will fail to implement proper security measures for mobile interfaces. Despite being served via a mobile device, reverse engineering can allow an attacker to analyze and exploit mobile APIs just like any other service. For this reason, it is important to ensure proper rate limiting and security measures are taken when developing mobile APIs to guard against potential attacks.

Example Attack

Snapchat, a popular application used to share self-expiring pictures with friends, suffered from multiple weak server side control issues. The Snapchat application allowed users to input a phone number to check if the number was currently using the Snapchat service. This was originally intended to allow users to easily add their friends but was not being properly rate limited. The developers had made the mistake of assuming that because the option was only available to mobile users so it would not be exploited.

Despite the vulnerability being reported to Snapchat, the development team did not implement access controls to limit these queries. Soon after the full disclosure, the entire Snapchat database was dumped and uploaded to a newly created SnapchatDB website:


SnapchatDB!
Bringing 4.6 million users' information to your fingertips...
Download the database
as SQL dump
as CSV text

```
mysql> select phone, username from records
                where username like '%sex%'
                 or username like '%porn%';

+------------+----------------+
| phone      | username       |
+------------+----------------+
| 21232104[redacted] | lsexy[redacted]    |
| 21239081[redacted] | sexyp[redacted]    |
| 21292016[redacted] | sexxx[redacted]    |
| 21364000[redacted] | jesse[redacted]    |
```

Figure 154 - The SnapchatDB.info banner

This website offered information about the dumped content as well as links to the entire leaked database for anyone to download. Snapchat responded by applying access controls to the find friends features after receiving a large amount of bad publicity over the issue.

Remediation and Mitigation Techniques

Remediation of weak server side controls can be accomplished by applying the same scrutiny to mobile APIs as any other web application. Just because an API is only used through a mobile device does not mean it is secure from attack. As demonstrated above, the false believe that obscurity equals security can often result in compromise. A mobile application should employ the same rate limiting, input validation, and data scrutiny as any regular web application.

Weak Server Side Controls

Additional Resources

Snapchat - GSFD

- http://gibsonsec.org/snapchat/fulldisclosure/

Greyhats expose 4.5 million Snapchat phone numbers using “theoretical” hack (updated)

- http://arstechnica.com/security/2014/01/greyhats-expose-4-5-million-snapchat-phone-numbers-using-theoretical-hack/

Insecure Mobile Data Storage

Description

Insecure data storage occurs when sensitive information is stored on a mobile device in an unencrypted form. Any information stored on a mobile device is accessible by an attacker, and in most cases an attacker will have the ability to perform actions an application cannot (access the keychain, ability to change filesystem permissions, ability to modify code, etc). An attacker who gains access to a mobile device gains access to the data stored on that device.

Sensitive data such as user credentials, session information, API keys, PII are often insecurely stored within a local database, XML, plist, log, or configuration files.

Remediation and Mitigation Techniques

The root cause of insecure data storage is the failure to encrypt sensitive data on a local filesystem. The best remediation for insecure data storage is to not persistently store sensitive data on mobile devices. All data stored on a device should be considered readable by an attacker. Jailbreak or root detection should not be considered a sufficient means of protecting sensitive data. Should business requirements dictate that sensitive data must be stored on the device, the following best practices are recommended:

- Enforce the use of strong peer-reviewed encryption algorithms.
- Implement an authenticated encryption scheme such as AES-GCM whenever possible.
- Never create or use homegrown encryption libraries.
- For iOS and Android devices, the operating system provides a secure keychain to store sensitive data.
- Disable SSL response caching using no-cache and no-store response headers.

Additional Resources

Securing and Encrypting Data on iOS

- http://code.tutsplus.com/tutorials/securing-and-encrypting-data-on-ios--mobile-21263

iFunBox

- http://www.i-funbox.com/

iExplorer

- http://www.macroplant.com/iexplorer/

OWASP: Insecure Data Storage

- https://www.owasp.org/index.php/Mobile_Top_10_2014-M2

Android: Security Tips

- http://developer.android.com/training/articles/security-tips.htm

Mobile Transport Layer Security

Description

Proper transport security is an important aspect of ensuring a secure digital environment. At a core it is ensuring that data being send and received is unmodified, validated, and private. Implementing TLS/SSL improperly can lead to Man in the Middle (MITM) attacks causing loss of user credentials, session tokens, and more.

It is important to use TLS for any situation where a user sends sensitive information. If TLS was implemented only for logging in the user in but not for other application requests, the user's session cookies could be stolen. If the user's session id is sent over HTTP then an attacker can comprise the session via a man in the middle attack. It is important to use TLS when any sensitive information is sent, proper verification should also be made to ensure no leaking of data over plaintext occurs.

Web Views

When using a Web view to display content in a mobile application it is important to follow normal web application security best practices. Mixing TLS pages with non-TLS content is insecure due to the possibility of injection rogue data or scripts into non-TLS content via a Man in the Middle (MITM) attack. If a page is securely loaded using TLS but also includes an external resource such as JavaScript over a non-secure connection the included content could be modified to include a JavaScript key logger or other dangerous scripts. For this reason it is important to keep non-TLS content from being loaded in TLS secured pages.

HTTP cookies can be set with a “Secure” flag that will specify that the cookie can only to be sent over HTTPS connections and prevents the possibility of an attacker redirecting a user to an HTTP webpage and stealing session cookies via an insecure HTTP request.

Certificate Pinning

Transport Layer Security (TLS) provides three key factors for protecting a connection: confidentiality, integrity, and authenticity. To provide authenticity to SSL/TLS connections, a server's SSL/TLS certificate must be digitally signed by a trusted Certificate Authority (CA). In order to confer trust to CAs the operating systems maintains a list of public keys belonging to CAs that have been designated as trusted. These public keys can then be used by clients to verify that server certificates are signed by a trusted CA. However, by default any CA can create a certificate for any domain.

Applications that implement certificate pinning, or strict SSL certificate validation, take this a step further by requiring that the leaf certificate thumbprint matches a specific, often hard-coded value, and/or that server-supplied SSL certificates are signed by a single specific CA. In many cases, the trusted CA is owned exclusively by the application vendor.

The reliance on third-party certificate authorities is designed for scenarios in which the client does not know in advance to which host it will be connecting. Typical web browsers are a good example of this. However, in native iOS or Android applications the service endpoints are known in advance. There is therefore an opportunity for a higher level of security by forcing the application to verify that all SSL connections are secured with a specific SSL leaf certificate signed by a trusted CA. The lack of certificate pinning is a missed opportunity for

Mobile Transport Layer Security

extra protection against reverse engineering, compromised root authorities, and untrusted 3rd party certificates.

Remediation and Mitigation Techniques

The root cause of certificate pinning vulnerabilities is the failure to perform strict validation of SSL certificates.

- Verify the signing authority on existing certificates by creating a whitelist of `SubjectPublicKeyInfo` fingerprints using a cryptographically secure hashing algorithm such as SHA-256.
- Optionally, create a root authority specific to your organization and have all applications trust certificates signed only by this authority.
  Implement jailbreak detection to deter attackers from modifying the strict certificate validation code.

Additional Resources

HTTP Strict Transport Security – OWASP
- https://www.owasp.org/index.php/HTTP_Strict_Transport_Security

Making sure a web page is not cached, across all browsers - Stack Overflow
- https://stackoverflow.com/questions/49547/making-sure-a-web-page-is-not-cached-across-all-browsers

How to fix a website with blocked mixed content - Security | MDN
- https://developer.mozilla.org/en-US/docs/Security/MixedContent/How_to_fix_website_with_mixed_content

SecureFlag – OWASP
- https://www.owasp.org/index.php/SecureFlag#web.xml

PHP: Runtime Configuration - Manual
- http://www.php.net/manual/en/session.configuration.php

PHP: setcookie – Manual
- http://www.php.net/manual/en/function.setcookie.php

httpCookies Element (ASP.NET Settings Schema)
- http://msdn.microsoft.com/en-us/library/vstudio/ms228262%28v=vs.100%29.aspx

OWASP - Certificate and Public Key Pinning
- https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning

OWASP - Pinning Cheatsheet
- https://www.owasp.org/index.php/Pinning_Cheat_Sheet

Your App Shouldn’t Suffer SSL’s Problems
- http://www.thoughtcrime.org/blog/authenticity-is-broken-in-ssl-but-your-app-ha/

Android Pinning
- https://github.com/moxie0/AndroidPinning

Insufficient Jailbreak or Root Detection

Description

Insufficient jailbreak or root detection occurs when an application attempts to detect the existence of unsigned code on an iOS or the existence of a root user on an Android device in such a way that the detection mechanism can be easily bypassed. The absence of jailbreak or root detection allows attackers to more easily reverse engineer and attack mobile applications as well as backend infrastructure. Additionally, attackers may jailbreak or root a stolen device to forensically recover sensitive data.

Remediation and Mitigation Techniques

Jailbreak or root detection can be expensive and time consuming to implement and is not necessary for all types of applications. If an application does not perform sensitive transactions, store sensitive data, or implement sensitive business logic, there is a good chance that the application does not need jailbreak or root detection. Jailbreak and root detection are means to deter attackers, however, there is no foolproof detection technique that cannot be bypassed by a determined attacker.

iOS Application Jailbreak Protection Techniques

- Apply rigorous anti-jailbreaking countermeasures that are implemented in a low-level language such as C or assembly. The anti-jailbreaking routines should be obfuscated and updated with each release.
- Avoid the use of any Objective-C code within anti-jailbreaking features, since Objective-C methods are easier to modify.
- Implement detection methods utilizing functions such as `_dyld_image_count()` and `_dyld_get_image_name()` to determine which dynamic libraries are currently loaded by the process.
- Do not make library calls; e.g. no `memcpy()`, no `memset()`, no `open()`, no `fork()`.
- The only “safe” calls are syscalls (using the SVC assembly instruction) and even those can be hooked, patched, or bypassed with a little effort.
- Make sure that anti-jailbreaking code is always inlined in an application (this can be achieved by marking the jailbreak function with “`__attribute__((always_inline))`” in its declaration).
- Make pervasive use of anti-jailbreaking. Do not just check once during the application startup process.
- Consider use of commercial obfuscation and runtime integrity solutions such as Metaforic or Arxan.
- Report jailbroken devices back to your servers and maintain a historic record of jailbreak events.
- Randomize detection checks throughout the application. Do not rely on a single function, or method.

Android Application Root Detection Techniques

- Implement a check to scan the SD card for known rooting tools and applications.
- Add checks for other common root tools such as `tcpdump` and `gdb`.
- Determine if `/etc/security/otacerts.zip` exists. This file is often renamed

Insufficient Jailbreak or Root Detection

or removed to prevent updates that remove root access.

- Randomize detection checks throughout the application; do not rely on a single function, or method.

Additional Resources

Bypassing Jailbreak Detection

- http://theiphonewiki.com/wiki/Bypassing_Jailbreak_Detection

App Minder

- http://appminder.nesolabs.de/

Ultimate Guide to Rooting Android and Jailbreak iOS

- http://dottech.org/23370/how-to-root-android-jailbreak-ios-iphone-ipad-homebrew-webos/

Arxan

- http://www.arxan.com/

Metaforic

- http://metaforic.com/

Gone in 59 seconds: tips and tricks to bypass AppMinder’s Jailbreak detection

- http://reverse.put.as/2013/06/30/gone-in-59-seconds-tips-and-tricks-to-bypass-appminders-jailbreak-detection/

Accountability and Data Ownership

Description

In the past, the company that owned the data also owned the hardware. Today, with the rising popularity of cloud computing, this is no longer the case. Just because the cloud provider is in physical possession of the data does not absolve the data owner from responsibility if the cloud provider suffers a compromise.

Example Attack

Consider a case where your company has procured a certain amount of cloud space to store the financial and medical records of your company. Your server runs alongside other servers from different companies on the same physical hardware. All these virtual servers are at some level administered by the cloud provider. Since they have physical access to the hardware, they can inspect the memory state of all the machines. If an administrative workstation owned by the hosting provider is compromised, all the data stored on the virtual machines will be compromised as well.

Consider precisely how space is allocated to a virtual host. When you purchase space from a cloud provider, the standard procedure is to procure any free space available on an ad-hoc basis and allocate it to the new server. If the physical space allocated happens to be the same physical space that once housed on of your company's virtual hosts, a malicious user may be able to recover confidential information if the cloud provider does not properly sanitize the physical space before being re-allocated.

Be aware of what geographical location your hosting provider operates from. If the provider operates out of a foreign country, it is likely that export restrictions and special compliance laws apply.

Remediation and Mitigation Techniques

The remediation strategy for these types of issues is a combination of education (in the case of laws and policies) and encryption (to prevent compromise of data). When possible, securely delete all confidential information from a cloud instance before decommissioning it.

Additional Resources

OWASP - Accountability and Data Ownership

- https://www.owasp.org/index.php/Cloud-10_Accountability_and_Data_Ownership

IJCA - Distributed Accountability for Data Sharing in Cloud

- http://research.ijcaonline.org/volume59/number8/pxc3884039.pdf

Cloud Security

Description

When considering hosting an application or its data in the cloud, developers should take into consideration what type of information is being stored and/or processed. Additional measures should be taken to protect data that is considered high-impact if compromised. Strong encryption should be applied to the sensitive data while at rest (configuration files, logs, databases), in transit (SSL/TLS), and in use (in memory) when stored in the cloud. Once a decision has been made on whether the risk of storing the information in the cloud is considered acceptable, the security mechanisms available on the cloud-hosting provider should be reviewed for best practices. For an example, when using Amazon AWS for cloud hosting, the following best practices should be followed:

AWS Security

- Use Amazon's identity access and management (IAM) to create users, groups, and permissions for users and applications to access your AWS resources.
- Users start with no permissions; only add the permissions necessary for a user or application to complete their tasks. These permissions can be very granular including only allowing read access to a specific bucket, to only allowing a user to create EC2 instances.
- Use the AWS SDK and IAM Roles to automatically rotate access keys for your application. This allows the developers to focus on writing the application instead of properly and securely storing AWS access keys.
- Enable a strong password policy and force users to reset their passwords. If setting up IAM for the first time, create this policy before creating any users in order to ensure that all users use are force to create strong passwords.
- For users with permissions to modify critical infrastructure, access sensitive data, or administer other users, require the use of multi-factor authentication (MFA). Both hardware and software (mobile device) options are available.
- Never create access tokens for your root/master account. Instead, create a user with the required permissions and create access tokens for that user. If an attacker were to compromise root/master access keys, they would have access to perform any task in the AWS environment including adding/modifying/deleting data, backups, or infrastructure including servers.
- If access keys were created for the root/master account, identify all instances of the key in developer and production code and migrate them to a newly created key with locked-down permissions. Once migrated and tested, delete any remaining root/master access keys.