

# OWASP TOP 10

2007 RELEASE CANDIDATE 1



## THE TEN MOST CRITICAL WEB APPLICATION SECURITY VULNERABILITIES

### 2007 UPDATE

*© 2002-2007 OWASP Foundation*

*This document is licensed under the Creative Commons [Attribution-ShareAlike 2.5](https://creativecommons.org/licenses/by-sa/2.5/) license*



## TABLE OF CONTENTS

Table of Contents .....	2
Introduction .....	3
Summary .....	4
Methodology .....	5
A1 – Cross Site Scripting (XSS) .....	8
A2 – Injection Flaws.....	11
A3 – Malicious File Execution .....	13
A4 – Insecure Direct Object Reference .....	17
A5 – Cross Site Request Forgery (CSRF) .....	19
A6 – Information Leakage and Improper Error Handling .....	22
A7 – Broken Authentication and Session Management .....	24
A8 – Insecure Cryptographic Storage.....	26
A9 – Insecure Communications .....	28
A10 – Failure to Restrict URL Access .....	30
Where To Go From Here.....	32
References .....	35

## INTRODUCTION

Welcome to the OWASP Top 10 2007! This totally re-written edition lists the most serious web application vulnerabilities, discusses how to protect against them, and provides links to more information.

### AIM

**The primary aim of the OWASP Top 10 is to educate developers, designers, architects and organizations** about the consequences of the most common web application security vulnerabilities. The Top 10 provides basic methods to protect against these vulnerabilities – a great start to your secure coding security program.

**Security is not a one-time event.** It is insufficient to secure your code just once. By 2008, this Top 10 will have changed, and without changing a line of your application's code, you may be vulnerable. Please review the advice in [Where to go from here](#) for more information.

**A secure coding initiative must deal with all stages of a program's lifecycle.** Secure web applications are *only* possible when a secure SDLC is used. Secure programs are secure by design, during development, and by default. There are at least 300 issues that affect the overall security of a web application. These 300+ issues are detailed in the [OWASP Guide](#), which is essential reading for anyone developing web applications today.

**This document is first and foremost an education piece, not a standard.** Please do not adopt this document as a policy or standard without [talking to us](#) first! If you need a secure coding policy or standard, OWASP has secure coding policies and standards projects in progress. Please consider joining or financially assisting with these efforts.



## SUMMARY

<b>A1 – Cross Site Scripting (XSS)</b>	XSS flaws occur whenever an application takes user supplied data and sends it to a web browser without first validating or encoding that content. XSS allows attackers to execute script in the victim's browser which can hijack user sessions, deface web sites, etc.
<b>A2 – Injection Flaws</b>	Injection flaws, particularly SQL injection, are common in web applications. Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. The attacker's hostile data tricks the interpreter into executing unintended commands or changing data.
<b>A3 – Insecure Remote File Include</b>	Code vulnerable to remote file inclusion allows attackers to include hostile code and data, resulting in devastating attacks, such as total server compromise.
<b>A4 – Insecure Direct Object Reference</b>	A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter. Attackers can manipulate those references to access other objects without authorization.
<b>A5 – Cross Site Request Forgery (CSRF)</b>	A CSRF attack forces a logged-on victim's browser to send a pre-authenticated request to a vulnerable web application, which then forces the victim's browser to perform a hostile action to the benefit of the attacker.
<b>A6 – Information Leakage and Improper Error Handling</b>	Applications can unintentionally leak information about their configuration, internal workings, or violate privacy through a variety of application problems. Attackers use this weakness to violate privacy, or conduct further attacks.
<b>A7 – Broken Authentication and Session Management</b>	Account credentials and session tokens are often not properly protected. Attackers compromise passwords, keys, or authentication tokens to assume other users' identities.
<b>A8 – Insecure Cryptographic Storage</b>	Web applications rarely use cryptographic functions properly to protect data and credentials. Attackers use weakly protected data to conduct identity theft and other crimes, such as credit card fraud.
<b>A9 – Insecure Communications</b>	Applications frequently fail to encrypt network traffic when it is necessary to protect sensitive communications.
<b>A10 – Failure to Restrict URL Access</b>	Frequently, the only protection for sensitive areas of an application is links or URLs are not presented to unauthorized users. Attackers can use this weakness to access and perform unauthorized operations.

Table 1: Top 10 Web application vulnerabilities for 2006

## METHODOLOGY

Our methodology for the Top 10 2007 was simple: take the [MITRE Vulnerability Trends for 2006](#), and distill the Top 10 *web application security* issues. The ranked results are as follows:

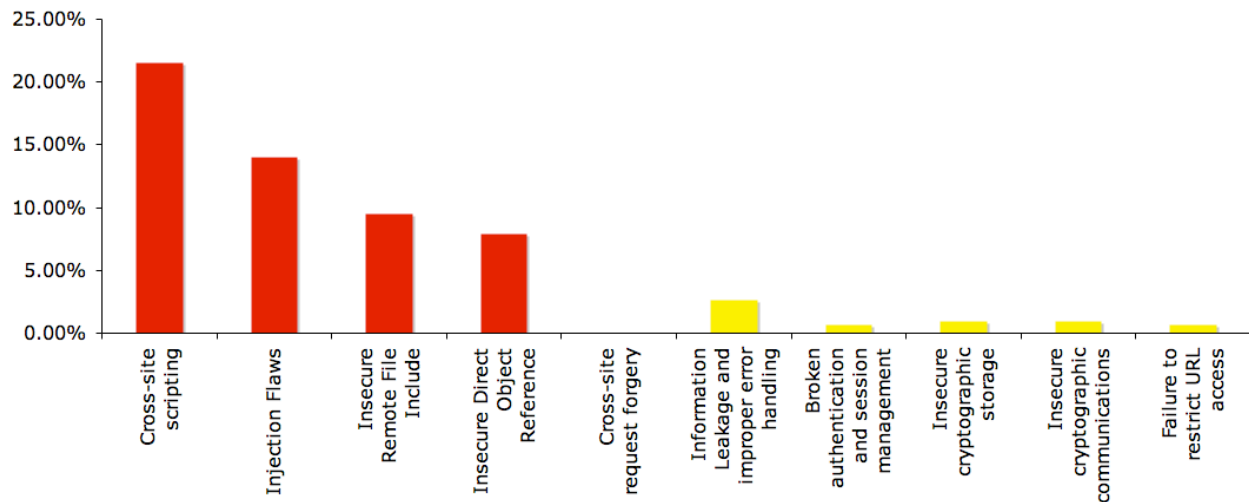


Figure 2: MITRE data on Top 10 web application vulnerabilities for 2006

Although we have tried to preserve the order, we have deliberately not chosen some weaknesses, such as buffer overflows as they are not widely applicable to web application security. In addition, we have transformed some raw findings into “meta” issues to fully capture the root cause of an issue represented by that data.

Cross Site Request Forgery (CSRF) is the major new addition to this edition of the OWASP Top 10. Although raw data ranks it at #36, we feel that it is important enough that applications should start protection efforts today, particularly for high value applications and applications which deal with sensitive data.

We have not included language specific (C, C++, etc) issues, such as buffer overflows, format string attacks, or any of the other common weaknesses which plague desktop and server software. If you are delivering programs for desktop or server platforms, or are including tools, plug-ins, or external programs to be called by your web application, we strongly recommend you reference the [OWASP Guide](#) and the books in the references section for more information on how to build or use these safely.

All of the protection recommendations provide solutions for the three most prevalent web application frameworks: Java EE, ASP.NET, and PHP. Other common web application frameworks, such as Ruby on Rails or Perl can easily adapt the recommendations to suit their specific needs.

## BIASES

The methodology described above necessarily biases the Top 10 towards discoveries by the security researcher community. This pattern of discovery is similar to the methods of [actual attack](#), particularly as it relates to entry-level (“script kiddie”) attackers. Protecting your software against the Top 10 will provide a modicum of protection against the most common forms of attack, but far more importantly, help set a course for improving the security of your software.



## MAPPING

There have been changes to the headings, even where content maps closely to previous content. We no longer use the WAS XML naming scheme as it has not kept up to date with modern vulnerabilities, attacks, and countermeasures. The table below depicts how this edition maps to the Top 10 2004, and the raw MITRE ranking:

OWASP Top 10 2007	OWASP Top 10 2004	MITRE 2006 Raw Ranking
A1. Cross Site Scripting (XSS)	A4. Cross Site Scripting (XSS)	1
A2. Injection Flaws	A6. Injection Flaws	2
A3. Insecure Remote File Include (NEW)		3
A4. Insecure Direct Object Reference	A2. Broken Access Control (split in 2007 T10)	5
A5. Cross Site Request Forgery (CSRF) (NEW)		36
A6. Information Leakage and Improper Error Handling	A7. Improper Error Handling	6
A7. Broken Authentication and Session Management	A3. Broken Authentication and Session Management	14
A8. Insecure Cryptographic Storage	A8. Insecure Storage	8
A9. Insecure Communications (NEW)	Discussed under A10. Insecure Configuration Management	8
A10. Failure to Restrict URL Access	A2. Broken Access Control (split in 2007 T10)	14
	A1. Unvalidated Input	7
	A5. Buffer Overflows	4, 8, and 10
	A9. Denial of Service	17
	A10. Insecure Configuration Management	29

## WHY WE HAVE DROPPED SOME IMPORTANT ISSUES

**Unvalidated input** is a major challenge for any development team, and is at the root of many application security problems. In fact, many of the other items in the list recommend validating input as a part of the solution. We still strongly recommend creating a centralized input validation mechanism as a part of your web applications. For more information, read the following data validation articles at OWASP:

- [http://www.owasp.org/index.php/Data\\_Validation](http://www.owasp.org/index.php/Data_Validation)
- [http://www.owasp.org/index.php/Testing\\_for\\_Data\\_Validation](http://www.owasp.org/index.php/Testing_for_Data_Validation)

**Buffer overflows, integer overflows and format string issues** are extremely serious vulnerabilities for programs written in languages such as C or C++. Remediation for these issues is covered by the traditional non-web application security community, such as SANS, CERT, and programming language tool vendors. If your code is

written in a language that is likely to suffer buffer overflows, we encourage you to read the buffer overflow content on OWASP:

- [http://www.owasp.org/index.php/Buffer\\_overflow](http://www.owasp.org/index.php/Buffer_overflow)
- [http://www.owasp.org/index.php/Testing\\_for\\_Buffer\\_Overflow](http://www.owasp.org/index.php/Testing_for_Buffer_Overflow)

**Denial of service** is a serious attack that can affect any site written in any language. The ranking of DoS by MITRE is insufficient to make the Top 10 this year. If you have concerns about denial of service, you should consult the OWASP site and Testing Guide:

- [http://www.owasp.org/index.php/Category:Denial\\_of\\_Service\\_Attack](http://www.owasp.org/index.php/Category:Denial_of_Service_Attack)
- [http://www.owasp.org/index.php/Testing\\_for\\_Denial\\_of\\_Service](http://www.owasp.org/index.php/Testing_for_Denial_of_Service)

**Insecure configuration management** affects all systems to some extent, particularly PHP. However, the ranking by MITRE does not allow us to include this issue this year. When deploying your application, you should consult the latest OWASP Guide and the OWASP Testing Guide for detailed information regarding secure configuration management and testing:

- <http://www.owasp.org/index.php/Configuration>
- [http://www.owasp.org/index.php/Testing\\_for\\_infrastructure\\_configuration\\_management](http://www.owasp.org/index.php/Testing_for_infrastructure_configuration_management)

## VULNERABILITIES, NOT ATTACKS

The previous edition of the Top 10 contained a mixture of attacks, vulnerabilities and countermeasures. This time around, we have focused solely on vulnerabilities. If organizations use this document to secure their applications, and reduce the risks to their business, it will lead to a direct reduction in the likelihood of:

- Phishing attacks that can exploit any of these vulnerabilities, particularly XSS, and weak or non-existent authentication or authorization checks (A1, A4, A7, A10)
- Privacy violations from poor validation, business rule and weak authorization checks (A2, A4, A6, A7, A10)
- Identity theft through poor or non-existent cryptographic controls (A8 and A9), remote file include (A3) and authentication, business rule, and authorization checks (A4, A7, A10)
- Systems compromise through remote file include (A3) and end of business class of data alteration or destruction attacks via Injections (A2)
- Financial loss through unauthorized transactions and CSRF attacks (A4, A5, A7, A10)
- Reputation loss through exploitation of any of the above vulnerabilities (A1 ... A10)

Once an organization moves away from worrying about reactive controls, and moves forward to proactively reducing risks applicable to their business, they will improve compliance with regulatory regimes, reduce operational costs, and hopefully will have far more robust and secure systems as a result.

## ACKNOWLEDGEMENTS

We thank the MITRE Project for making *Vulnerability Type Distribution in CVE* data freely available for use. The OWASP Top Ten project is led and sponsored by [Aspect Security](http://www.aspectsecurity.com).





## A1 – CROSS SITE SCRIPTING (XSS)

Cross site scripting, better known as XSS, is the most prevalent and pernicious web application security issue. XSS flaws occur whenever an application takes data that originated from a user and sends it to a web browser without first validating or encoding that content.

XSS allows attackers to execute script in the victim's browser, which can hijack user sessions, deface web sites, insert hostile content, conduct phishing attacks, and take over the user's browser using scripting malware. The malicious script is usually JavaScript but any scripting language supported by the victim's browser is a potential target for this attack.

### ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to cross site scripting.

### VULNERABILITY

There are three known types of cross site scripting: reflected, stored, and DOM injection. Reflected XSS is the easiest to exploit – a page will reflect user supplied data directly back to the user:

```
echo $_REQUEST['userinput'];
```

Stored XSS takes hostile data, stores it in a file, a database, or other back end system, and then at a later stage, displays the data to the user, unfiltered. This is extremely dangerous in systems such as CMS, blogs, or forums, where a large number of users will see input from other individuals.

With DOM based XSS attacks, the site's JavaScript code and variables are manipulated rather than HTML elements. Alternatively, attacks can be a blend or hybrid of all three types. The danger with cross site scripting is not the type of attack, but that it is possible.

Attacks are usually implemented in JavaScript, which is a powerful scripting language. Using JavaScript allows attackers to manipulate any aspect of the rendered page, including adding new elements (such as adding a login tile which forwards credentials to a hostile site), manipulating any aspect of the internal DOM tree, and deleting or changing the way the page looks and feels. JavaScript allows the use of XMLHttpRequest, which is typically used by sites using AJAX technologies, even if victim site does not use AJAX today.

Using XMLHttpRequest, it is sometimes possible to get around a browser's same source origination policy - thus forwarding victim data to hostile sites, and to create complex worms and malicious zombies that last as long as the browser stays open. AJAX attacks do not have to be visible or require user interaction to perform dangerous cross site request forgery (CSRF) attacks (see A-5).

### VERIFYING SECURITY

The goal is to verify that all the parameters in the application are validated and/or encoded before being included in HTML pages.



Automated approaches: Both vulnerability scanning tools and static code analysis tools can find simple XSS problems, particularly reflected ones. They frequently can't find complex XSS flaws, such as when the injection occurs in a peculiar HTML structure or script. They are also not likely to find instances of DOM based XSS.

Manual approaches: If a centralized validation and encoding mechanism is used, the most efficient way to verify security is to check the code. If a distributed implementation is used, then the verification will be considerably more time-consuming. Testing is time-consuming because the attack surface of most applications is so large.

## PROTECTION

The best protection for XSS is a combination of "whitelist" validation of all incoming data and appropriate encoding of all output data. Validation allows the detection of attacks, and encoding prevents any successful script injection from running in the browser.

Preventing XSS across an entire application requires a consistent architectural approach:

- **Input validation.** Use a standard input validation mechanism to validate all input data for length, type, syntax, and business rules before accepting the data to be displayed or stored. Use an "accept known good" validation strategy. Reject invalid input rather than attempting to sanitize potentially hostile data.
- **Strong output encoding.** Ensure that all user-supplied data is HTML entity encoded before rendering in HTML, taking the approach to encode all characters other than a very limited subset. This is the approach of the Microsoft Anti-XSS library, and the forthcoming OWASP PHP Anti-XSS library.
- **Do not use "blacklist" validation** to detect XSS in input or to encode output. Searching for and replacing just a few characters ("`<`" "`>`" and other similar characters) is weak and has been attacked successfully.

Language specific recommendations:

- Java: Use Struts output mechanisms such as `<bean:write ... >`, or use the default JSTL `escapeXML="true"` attribute in `<c:out ... >`. Do NOT use `<%= ... %>` unnested (that is, outside of a properly encoded output mechanism).
- .NET: Use the Microsoft Anti-XSS Library 1.5 freely available from MSDN. Do not assign form fields data directly from the Request object: `username.Text = Request.QueryString("username");` without using this library. Understand which .NET controls automatically encode output data.
- PHP: Ensure output is passed through `htmlspecialchars()` or `htmlspecialchars()` or use the soon to be released OWASP PHP Anti-XSS library.

## SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4206>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-3966>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5204>

## REFERENCES

- OWASP – Cross site scripting, [http://www.owasp.org/index.php/Cross\\_Site\\_Scripting](http://www.owasp.org/index.php/Cross_Site_Scripting)
- OWASP – Testing for XSS, [http://www.owasp.org/index.php/Testing\\_for\\_Cross\\_site\\_scripting](http://www.owasp.org/index.php/Testing_for_Cross_site_scripting)



- OWASP Stinger Project (A Java EE validation filter) – [http://www.owasp.org/index.php/Category:OWASP Stinger Project](http://www.owasp.org/index.php/Category:OWASP_Stinger_Project)
- OWASP PHP Filter Project - [http://www.owasp.org/index.php/OWASP PHP Filters](http://www.owasp.org/index.php/OWASP_PHP_Filters)
- OWASP Encoding Project - [http://www.owasp.org/index.php/Category:OWASP Encoding Project](http://www.owasp.org/index.php/Category:OWASP_Encoding_Project)
- RSnake, XSS Cheat Sheet, <http://ha.ckers.org/xss.html>
- Klein, A., DOM Based Cross Site Scripting, <http://www.webappsec.org/projects/articles/071105.shtml>
- .NET Anti-XSS Library - <http://www.microsoft.com/downloads/details.aspx?FamilyID=efb9c819-53ff-4f82-bfaf-e11625130c25&DisplayLang=en>

## A2 – INJECTION FLAWS

Injection flaws, particularly SQL injection, are common in web applications. Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. The attacker's data tricks the interpreter into executing unintended commands. Injection flaws allow attackers to create, read, update, or delete any arbitrary data available to the application. In the worst case scenario, these flaws allow an attacker to completely compromise the application.

### ENVIRONMENTS AFFECTED

All web application frameworks that use interpreters are vulnerable to injection attacks.

### VULNERABILITY

If user input is passed into an interpreter without validation or encoding, the application is vulnerable. Check if user input is supplied to dynamic queries, such as:

```
$sql = "SELECT * FROM table WHERE id = '" . $_REQUEST['id'] . "'";
```

### VERIFYING SECURITY

The goal is to verify that user data cannot modify the meaning of commands and queries sent to any of the interpreters invoked by the application.

Automated approaches: Many vulnerability scanning tools search for injection problems, particularly SQL injection. Detecting whether the injection worked or not is difficult and prone to error. Static analysis tools that search for uses of unsafe interpreter APIs are useful, but frequently cannot verify that appropriate validation or encoding might be in place to protect against the vulnerability.

Manual approaches: The most efficient and accurate approach is to check the code that invokes interpreters. The reviewer should verify the use of a safe API or that appropriate validation and/or encoding has occurred. Testing can be extremely time-consuming and spotty because the attack surface of most applications is so large.

### PROTECTION

Avoid the use of interpreters when possible. If you must invoke an interpreter, the key method to avoid injections is the use of safe APIs, such as strongly typed parameterized queries and object relational mapping (ORM) libraries. These interfaces handle all data escaping, or do not require escaping. Note that while safe interfaces solve the problem, validation is still recommended in order to detect attacks.

Using interpreters is dangerous, so it's worth it to take extra care, such as the following:

- **Enforce least privilege** when connecting to databases and other backend systems
- **Avoid detailed error messages** that are useful to an attacker
- **Do not send dynamic queries** into a parameterized interface
- **Be careful when using stored procedures** as they can be injectable



- **Do not use dynamic query interfaces** (such as `mysql_query()` or similar)
- **Do not use simple escaping functions**, such as PHP's `addslashes()` or character replacement functions like `str_replace("'", "''")`. These are weak and have been successfully exploited by attackers.

Language specific recommendations:

- Java EE – use strongly typed `PreparedStatement`, or ORMs such as Hibernate or Spring
- .NET – use strongly typed parameterized queries, such as `SqlCommand` with `SqlParameter` or an ORM like Hibernate.
- PHP – use PDO with strongly typed parameterized queries (using `bindParam()`)

## SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5121>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4953>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4592>

## REFERENCES

- OWASP, [http://www.owasp.org/index.php/SQL\\_Injection](http://www.owasp.org/index.php/SQL_Injection)
- OWASP, [http://www.owasp.org/index.php/Guide\\_to\\_SQL\\_Injection](http://www.owasp.org/index.php/Guide_to_SQL_Injection)
- OWASP, [http://www.owasp.org/index.php/Reviewing\\_Code\\_for\\_SQL\\_Injection](http://www.owasp.org/index.php/Reviewing_Code_for_SQL_Injection)
- OWASP, [http://www.owasp.org/index.php/Testing\\_for\\_SQL\\_Injection](http://www.owasp.org/index.php/Testing_for_SQL_Injection)
- SQL Injection, <http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>
- Advanced SQL Injection, [http://www.ngssoftware.com/papers/advanced\\_sql\\_injection.pdf](http://www.ngssoftware.com/papers/advanced_sql_injection.pdf)
- More Advanced SQL Injection, [http://www.nextgenss.com/papers/more\\_advanced\\_sql\\_injection.pdf](http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf)
- Hibernate, an advanced object relational manager (ORM) for J2EE and .NET, <http://www.hibernate.org/>
- J2EE Prepared Statements, <http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html>
- How to: Protect from SQL injection in ASP.Net, <http://msdn2.microsoft.com/en-us/library/ms998271.aspx>
- PHP PDO functions, <http://php.net/pdo>

## A3 – MALICIOUS FILE EXECUTION

Malicious file execution vulnerabilities are found in many applications. Developers will often directly use or concatenate potentially hostile input with file or stream functions, or improperly trust input files. On many platforms, frameworks allow the use of external object references, such as URLs or file system references. When the data is insufficiently checked, this can lead to arbitrary remote and hostile content being included, processed or invoked by the web server.

This allows attackers to perform:

- Remote code execution
- Remote root kit installation and complete system compromise
- On Windows, internal system compromise may be possible through the use of PHP's SMB file wrappers

This attack is particularly prevalent on PHP, and extreme care must be taken with any stream or file function to ensure that user supplied input does not influence file names.

### ENVIRONMENTS AFFECTED

All web application frameworks that allow uploaded files to be executed are vulnerable to remote file include. By default, PHP 4.0.4 and later and 5.x are vulnerable to remote file inclusion. Other environments are susceptible if they allow file upload into web directories.

### VULNERABILITY

A common vulnerable construct is:

```
include $_REQUEST['filename'];
```

Not only does this allow evaluation of remote hostile scripts, it can be used to access local file servers (if PHP is hosted upon Windows) due to SMB support in PHP's file system wrappers.

Other methods of attack include:

- Hostile data being uploaded to session files, log data, and via image uploads (typical of forum software)
- Using compression or audio streams, such as `zlib://` or `ogg://` which do not inspect the internal PHP URL flag and thus allow access to remote resources even if `allow_url_fopen` or `allow_url_include` is disabled
- Using PHP wrappers, such as `php://input` and others to take input from the request POST data rather than a file
- Using PHP's `data: wrapper`, such as `data:;base64,PD9waHAgcGhwaW5mbygpOz8+`

As this list is extensive (and periodically changes), it is vital to use a properly designed security architecture and robust design when dealing with user supplied inputs influencing the choice of server side filenames and access.



Although PHP examples have been given, this attack is also applicable in different ways to .NET and J2EE. Applications written in those frameworks need to pay particular attention to code access security mechanisms to ensure that filenames supplied by or influenced by the user do not allow security controls to be obviated.

For example, it is possible that XML documents submitted by an attacker will have a hostile DTD that forces the XML parser to load a remote DTD, and parse and process the results. An Australian security firm has demonstrated this approach to port scanning behind firewalls. See [SIF01] in this chapter's references for more information.

The damage this particular vulnerability causes is directly related to the strength of the sandbox / platform isolation controls in the framework. As PHP is rarely isolated and has no sandbox concept or security architecture, the damage is far worse for an attack than other platforms with limited or partial trust, or are contained within a suitable sand box, such as when a web app is running under a JVM with the security manager properly enabled and configured (which is rarely the default).

## VERIFYING SECURITY

Automated approaches: Vulnerability scanning tools will have difficulty identifying the parameters that are used in a file include or the syntax for making them work. Static analysis tools can search for the use of dangerous APIs, but cannot verify that appropriate validation or encoding might be in place to protect against the vulnerability.

Manual approaches: A code review can search for code that might allow a file to be included in the application, but there are many possible mistakes to recognize. Testing can also detect these vulnerabilities, but identifying the particular parameters and the right syntax can be difficult.

## PROTECTION

Preventing remote file include flaws takes some careful planning at the architectural and design phases, through to thorough testing. In general, a well-written application will not use user-supplied input in any filename for any server-based resource (such as images, XML and XSL transform documents, or script inclusions), and will have firewall rules in place preventing new outbound connections to the Internet or internally back to any other server. However, many legacy applications will continue to have a need to accept user supplied input.

Among the most important considerations are:

- Consider a variable naming scheme to assist with taint checking:

```
$hostile = &$_POST; // refer to POST variables, not $_REQUEST
$safe['filename'] = validate_file_name($hostile['unsafe_filename']); // make it safe
```

Therefore any operation based upon hostile input is immediately obvious:

```
 require_once($_POST['unsafe_filename'] . 'inc.php');
```

```
 require_once($safe['filename'] . 'inc.php');
```

- Strongly validate user input using "accept known good" as a strategy
- Hide server-side filenames from the user. For example, instead of including `$language . ".lang.php"`, use an array index like this:

```
<select name="language"><option value="1">Français</option></select>
```

```
...
```

```
$language = intval($_POST['language']);
```

```
if ($language > 0) {
```

```
    require_once($lang[$language]); // lang is array of strings eg "fr.lang.php"
```

```
}
```

- Disable `allow_url_fopen` and `allow_url_include` in `php.ini` and consider building PHP locally to not include this functionality.
- Add firewall rules to prevent web servers making new connections to external web sites and internal systems. For high value systems, isolate the web server in its own VLAN or private subnet.
- Ensure that file and streams functions (`stream_*`) are carefully vetted. Ensure that the user input is not supplied any function which takes a filename argument, including:

```
include() include_once() require() require_once() fopen() imagecreatefromXXX() file()
file_get_contents() copy() delete() unlink() upload_tmp_dir() $_FILES move_uploaded_file()
```

- Be extremely cautious if data is passed to `system()`, `eval()`, `passthru()` or ``` (the backtick operator).
- Check that any files taken from the user for legitimate purposes cannot be otherwise obviated, such as including user supplied data in the session object, avatars and images, PDF reports, temporary files, and so on.
- With PHP, consider implementing a chroot jail or other sand box mechanisms such as virtualization to isolate applications from each other
- With J2EE, ensure that the security manager is enabled and properly configured and that the application is demanding permissions appropriately
- With ASP.NET, please refer to the documentation on partial trust, and design your applications to be segmented in trust, so that most of the application exists in the lowest possible trust state possible

## SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0360>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5220>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4722>

## REFERENCES

- OWASP Guide, [http://www.owasp.org/index.php/File\\_System#Includes\\_and\\_Remote\\_files](http://www.owasp.org/index.php/File_System#Includes_and_Remote_files)
- OWASP Testing Guide, [http://www.owasp.org/index.php/Testing\\_for\\_Directory\\_Traversal](http://www.owasp.org/index.php/Testing_for_Directory_Traversal)
- OWASP PHP Top 5, [http://www.owasp.org/index.php/PHP\\_Top\\_5#P1:Remote\\_Code\\_Execution](http://www.owasp.org/index.php/PHP_Top_5#P1:Remote_Code_Execution)



- Stefan Esser,  
[http://blog.php-security.org/archives/45-PHP-5.2.0-and-allow\\_url\\_include.html](http://blog.php-security.org/archives/45-PHP-5.2.0-and-allow_url_include.html)
- [SIF01] Sift Networks, Web Services: Teaching an old dog new tricks,  
[http://www.ruxcon.org.au/files/2006/web\\_services\\_security.ppt](http://www.ruxcon.org.au/files/2006/web_services_security.ppt)
- [http://www.owasp.org/index.php/OWASP\\_Java\\_Table\\_of\\_Contents#Defining\\_a\\_Java\\_Security\\_Policy](http://www.owasp.org/index.php/OWASP_Java_Table_of_Contents#Defining_a_Java_Security_Policy)
- Microsoft - Programming for Partial Trust, [http://msdn2.microsoft.com/en-us/library/ms364059\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms364059(VS.80).aspx)



## A4 – INSECURE DIRECT OBJECT REFERENCE

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter. Unless an access control check is in place, an attacker can manipulate those references to access other objects without authorization.

### ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to attacks on insecure direct object references.

### VULNERABILITY

Many applications expose their internal object references to users. Attackers use parameter tampering to change references and violate the intended but unenforced access control policy. Frequently, these references point to file systems and databases, but any exposed application construct could be vulnerable.

For example, if code allows user input to specify filenames or paths, it may allow attackers to jump out of the application's directory, and access other resources.

```
<select name="language"><option value="fr">Français</option></select>
...
require_once ($_REQUEST['language']."lang.php");
```

Such code can be attacked using a string like "../../../etc/passwd%00" using [null byte injection](#) (see the [OWASP Guide](#) for more information) to access any file on the web server's file system.

Similarly, references to database keys are frequently exposed. An attacker can attack these parameters simply by guessing or searching for another valid key. Often, these are sequential in nature. In the example below, even if an application does not present any links to unauthorized carts, and no SQL injection is possible, an attacker can still change the cartID parameter to whatever cart they want.

```
int cartID = Integer.parseInt( request.getParameter( "cartID" ) );
String query = "SELECT * FROM table WHERE cartID=" + cartID;
```

### VERIFYING SECURITY

The goal is to verify that the application does not allow direct object references to be manipulated by an attacker.

Automated approaches: Vulnerability scanning tools will have difficulty identifying which parameters are susceptible to manipulation or whether the manipulation worked. Static analysis tools really cannot know which parameters must have an access control check before use.

Manual approaches: Code review can trace critical parameters and identify whether they are susceptible to manipulation in many cases. Penetration testing can also verify that manipulation is possible. However, both of these techniques are time-consuming and can be spotty.



## PROTECTION

The best protection is to avoid exposing direct object references to users by using an index, map, or other indirect method that is easy to validate. If a direct object reference must be used, ensure that the user is authorized before using it.

Establishing a standard way of referring to application objects is important:

- Avoid exposing your private object references to users whenever possible
- Validate any private object references extensively with an "accept known good" approach
- Verify authorization to all referenced objects

If you must expose a file system reference, use an index value or a map to prevent path and filename manipulation.

```
http://www.example.com/application?file=1
```

If you must expose direct references to database structures, ensure that SQL statements and other database access methods only allow authorized records to be shown:

```
int cartID = Integer.parseInt( request.getParameter( "cartID" ) );  
  
User user = (User)request.getSession().getAttribute( "user" );  
  
String query = "SELECT * FROM table WHERE cartID=" + cartID + " AND userID=" + user.getID();
```

## SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0329>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4369>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0229>

## REFERENCES

- OWASP, [http://www.owasp.org/index.php/Testing\\_for\\_business\\_logic](http://www.owasp.org/index.php/Testing_for_business_logic)
- OWASP, [http://www.owasp.org/index.php/Testing\\_for\\_Directory\\_Traversal](http://www.owasp.org/index.php/Testing_for_Directory_Traversal)
- OWASP, [http://www.owasp.org/index.php/Category:Access\\_Control\\_Vulnerability](http://www.owasp.org/index.php/Category:Access_Control_Vulnerability)

## A5 – CROSS SITE REQUEST FORGERY (CSRF)

Cross site request forgery is not a new attack, but is simple and devastating. A CSRF attack forces a logged-on victim's browser to send a request to a vulnerable web application, which then performs the chosen action on behalf of the victim, to the benefit of the attacker. This vulnerability is extremely widespread, as any web application that authorizes requests based only on credentials that are automatically submitted by the browser is vulnerable. Unfortunately, today, most web applications rely solely on automatically submitted credentials such as session cookies, Basic Authentication credentials, source IP addresses, SSL certificates, Windows domain credentials, etc.

This vulnerability is also known by several other names including Session Riding, One-Click Attacks, Cross Site Reference Forgery, Hostile Linking, and Automation Attack. The acronym XSRF is also frequently used. OWASP and MITRE have both standardized on the term Cross Site Request Forgery and CSRF.

### ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to CSRF.

### VULNERABILITY

A typical CSRF attack against a forum might take the form of directing the user to invoke some function, such as the application's logout page. The following tag in any web page viewed by the victim will generate a request which logs them out:

```

```

If an online bank allowed its application to process requests, such as transfer funds, a similar attack might allow:

```

```

Both of these attacks work because the user's authorization credential (typically the session cookie) would automatically be included with such requests by the browser, even though the attacker didn't supply that credential.

If the tag containing the attack can be posted to a vulnerable application, then the likelihood of finding logged in victims is significantly increased, similar to the increase in risk between stored and reflected XSS flaws. XSS flaws are not required for a CSRF attack to work, although any application with XSS flaws is susceptible to CSRF because a CSRF attack can exploit the XSS flaw to steal any non-automatically submitted credential that might be in place to protect against a CSRF attack. Many application worms have used both techniques in combination.

When building defenses against CSRF attacks, you must also focus on eliminating XSS vulnerabilities in your application since such flaws can be used to get around most CSRF defenses you might put in place.

### VERIFYING SECURITY

The goal is to verify that the application protects against CSRF attacks by generating and then requiring some type of authorization token that is not automatically submitted by the browser.



Automated approaches: Vulnerability scanners should be able to detect the lack of CSRF protection in applications with a little bit of training. Static analysis tools, on the other hand, will have a difficult time recognizing a custom mechanism for protecting against this attack.

Manual approaches: Penetration testing is a quick way to verify that CSRF protection is in place. To verify that the mechanism is strong and properly implemented, checking the code is the most efficient course.

## PROTECTION

Applications must ensure that they are not relying on credentials or tokens that are automatically submitted by browsers. The only solution is to use a custom token that the browser will not 'remember' and then automatically include with a CSRF attack.

The following strategies should be inherent in all web applications:

- Ensure that there are no XSS vulnerabilities in your application (see A1 – Cross Site Scripting)
- Insert custom random tokens into every form and URL that will not be automatically submitted by the browser. For example,

```
<form action="/transfer.do" method="post">  
<input type="hidden" name="8438927730" value="43847384383">  
...  
</form>
```

and then verify that the submitted token is correct for the current user. Such tokens can be unique to that particular function or page for that user, or simply unique to the overall session. The more focused the token is to a particular function and/or particular set of data, the stronger the protection will be, but the more complicated it will be to construct and maintain.

- For sensitive data or value transactions, re-authenticate or use transaction signing to ensure that the request is genuine. Consider sending an e-mail or phoning the customer if the activity seems suspicious, to alert the user and potentially back out the transaction.
- **Do not** use GET requests (URLs) for sensitive data or to perform value transactions. Use only POST methods when processing sensitive data from the user.
- For ASP.NET, [set a ViewStateUserKey](#). (See references). This provides a similar type of check to a random token as described above.

While these suggestions will diminish your exposure dramatically, advanced CSRF attacks can bypass many of these restrictions. The strongest technique is the use of unique tokens, and eliminating all XSS vulnerabilities in your application.

## SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0192>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5116>
- MySpace Worm Explanation <http://namb.la/popular/tech.html>

- An attack which uses Quicktime to perform CSRF attacks  
[http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9005607&intsrc=hm\\_list](http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9005607&intsrc=hm_list)

## REFERENCES

- OWASP CSRF, [http://www.owasp.org/index.php/Cross-Site\\_Request\\_Forgery](http://www.owasp.org/index.php/Cross-Site_Request_Forgery)
- OWASP, [https://www.owasp.org/index.php/Testing\\_for\\_CSRF](https://www.owasp.org/index.php/Testing_for_CSRF)
- OWASP CSRF Guard, [http://www.owasp.org/index.php/CSRF\\_Guard](http://www.owasp.org/index.php/CSRF_Guard)
- OWASP PHP CSRF Guard, [http://www.owasp.org/index.php/PHP\\_CSRF\\_Guard](http://www.owasp.org/index.php/PHP_CSRF_Guard)
- RSnake, "What is CSRF?", <http://ha.ckers.org/blog/20061030/what-is-csrf/>
- Microsoft, ViewStateUserKey details,  
[http://msdn2.microsoft.com/en-us/library/ms972969.aspx#securitybarriers\\_topic2](http://msdn2.microsoft.com/en-us/library/ms972969.aspx#securitybarriers_topic2)



## A6 – INFORMATION LEAKAGE AND IMPROPER ERROR HANDLING

Applications can unintentionally leak information about their configuration, internal workings, or violate privacy through a variety of application problems. Applications can also leak internal state via how long they take to process certain operations or via different responses to differing inputs, such as displaying the same error text with different error numbers. Web applications will often leak information about their internal state through detailed or debug error messages.

### ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to information leakage and improper error handling.

### VULNERABILITY

Applications frequently generate error messages and display them to users. Many times these error messages are quite useful to attackers, as they reveal implementation details or information that is useful in exploring a vulnerability. There are several common examples of this:

- Detailed error handling, where inducing an error displays too much information, such as stack traces, failed SQL statements, or other debugging information.
- Functions that produce different results based upon different inputs. For example, supplying the same username but different passwords to a login function should produce the same text for no such user, and bad password. However, many systems produce different error codes.

### VERIFYING SECURITY

The goal is to verify that the application does not leak information via error messages or other means.

Automated approaches: Vulnerability scanning tools can and probably will cause error messages to be generated. Detecting whether the messages leak information is the challenge. Static analysis tools can search for the use of APIs that leak information, but will not be able to verify the meaning of those messages.

Manual approaches: A code review can search for improper error handling and other patterns that leak information, but it is time-consuming. Testing will also generate error messages, but knowing what error paths were covered is a challenge.

### PROTECTION

Developers should use tools like OWASP's WebScarab to try to make their application generate errors. Applications that have not been tested in this way will almost certainly generate unexpected error output. Applications should also include a standard exception handling architecture to prevent unwanted information from leaking to attackers.

Preventing information leakage requires discipline. The following practices have proven effective:

- Ensure that the entire software development team shares a common approach to exception handling.
- Disable or limit detailed error handling. In particular, do not display debug information to end users, stack traces, or path information.

- Ensure that secure paths that have multiple outcomes return similar or identical error messages in roughly the same time. If this is not possible, consider imposing a random wait time for all transactions to hide this detail from the attacker.

#### SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4899>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3389>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0580>

#### REFERENCES

- OWASP, [http://www.owasp.org/index.php/Error\\_Handling](http://www.owasp.org/index.php/Error_Handling)
- OWASP, [http://www.owasp.org/index.php/Category:Sensitive\\_Data\\_Protection\\_Vulnerability](http://www.owasp.org/index.php/Category:Sensitive_Data_Protection_Vulnerability)



## A7 – BROKEN AUTHENTICATION AND SESSION MANAGEMENT

Proper authentication and session management is critical to web application security. Flaws in this area most frequently involve the failure to protect credentials and session tokens through their lifecycle. These flaws can lead to the hijacking of user or administrative accounts, undermine authorization and accountability controls, and cause privacy violations.

### ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to authentication and session management flaws.

### VULNERABILITY

Flaws in the main authentication mechanism are not uncommon, but weaknesses are more often introduced through ancillary authentication functions such as logout, password management, timeout, remember me, secret question, and account update.

### VERIFYING SECURITY

The goal is to verify that the application properly authenticates users and properly protects identities and their associated credentials.

Automated approaches: Vulnerability scanning tools have a very difficult time detecting vulnerabilities in custom authentication and session management schemes. Static analysis tools are also not likely to detect authentication and session management problems in custom code.

Manual approaches: Code review and testing, especially in combination, are quite effective at verifying that the authentication, session management, and ancillary functions are all implemented properly.

### PROTECTION

Authentication relies on secure communication and credential storage. First ensure that SSL is the only option for all authenticated parts of the application (see A9 – Insecure Communications) and that all credentials are stored in hashed or encrypted form (see A8 – Insecure Cryptographic Storage).

Preventing authentication flaws takes careful planning. Among the most important considerations are:

- Use a single authentication mechanism with appropriate strength and number of factors
- Use the container provided session management mechanism and no custom cookies
- Create a new session upon successful authentication
- Ensure that every page has a logout link, and that logout destroys all server side session state and client side cookies
- Ensure ancillary authentication functions (questions and answers, password reset) are as strong as the main ones and don't contain flaws
- **Do not** expose any credentials in URLs or logs (no session rewriting)



## SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6145>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6229>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6528>

## REFERENCES

- OWASP, [http://www.owasp.org/index.php/Guide to Authentication](http://www.owasp.org/index.php/Guide_to_Authentication)
- OWASP, [http://www.owasp.org/index.php/Reviewing Code for Authentication](http://www.owasp.org/index.php/Reviewing_Code_for_Authentication)
- OWASP, [http://www.owasp.org/index.php/Testing for authentication](http://www.owasp.org/index.php/Testing_for_authentication)



## A8 – INSECURE CRYPTOGRAPHIC STORAGE

Protecting sensitive data with cryptography has become a key part of most web applications. Simply failing to encrypt sensitive data is very widespread. Applications that do encrypt frequently contain poorly designed cryptography, either using inappropriate ciphers or making serious mistakes using strong ciphers. These flaws can lead to disclosure of sensitive data and compliance violations.

### ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to insecure cryptographic storage.

### VULNERABILITY

Preventing cryptographic flaws takes careful planning. The most common problems are:

- Not encrypting sensitive data
- Using home grown algorithms
- Insecure use of strong algorithms
- Continued use of proven weak algorithms (MD5, SHA-1, RC3, RC4, etc...)
- Hard coding keys, and storing keys in unprotected stores

### VERIFYING SECURITY

The goal is to verify that the application properly encrypts sensitive information in storage.

Automated approaches: Vulnerability scanning tools cannot verify cryptographic storage at all. Code scanning tools can detect use of known cryptographic APIs, but cannot detect if it is being used properly or if the encryption is performed in an external component.

Manual approaches: Like scanning, testing cannot verify cryptographic storage. Code review is the best way to verify that an application encrypts sensitive data and has properly implemented the mechanism and key management. This may involve the examination of the configuration of external systems in some cases.

### PROTECTION

The most important aspect is to ensure that everything that should be encrypted is actually encrypted. Then you must ensure that the cryptography is implemented properly. As there are so many ways of using cryptography improperly, the following recommendations should be taken as part of your testing regime to help ensure secure cryptographic materials handling:

- **Do not** allow unqualified staff to try to create cryptographic algorithms. Use only approved public algorithms such as AES, RSA public key cryptography, and SHA-256 or better for hashing.
- **Do not** use weak algorithms, such as MD5 / SHA1. Favor safer alternatives, such as SHA-256 or better.

- Generate keys offline and store private keys with extreme care
- Ensure that infrastructure credentials such as database credentials or MQ queue access details are securely encrypted and not easily decrypted by local or remote users
- Ensure that encrypted data stored on disk is not easy to decrypt. For example, database encryption is worthless if the database connection pool provides unencrypted access. All this does is slow down the database and make queries very slow.
- Under PCI Data Security Standard requirement 3, you must protect cardholder data. PCI DSS compliance is mandatory by 2008 for merchants and anyone else dealing with credit cards. Good practice is to never store unnecessary data, such as the magnetic stripe information or the primary account number (PAN, otherwise known as the credit card number). If you store the PAN, the DSS compliance requirements are hefty and continuing. For example, you are NEVER allowed to store the CVV number (the three digit number on the rear of the card) under any circumstances. For more information, please see the PCI DSS Guidelines and implement controls as necessary.

#### SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6145>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1664>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-1101> (True of most Java EE servlet containers, too)

#### REFERENCES

- OWASP, <http://www.owasp.org/index.php/Cryptography>
- OWASP, [http://www.owasp.org/index.php/Guide\\_to\\_Cryptography](http://www.owasp.org/index.php/Guide_to_Cryptography)
- OWASP, [http://www.owasp.org/index.php/Insecure\\_Storage](http://www.owasp.org/index.php/Insecure_Storage)
- OWASP, [http://www.owasp.org/index.php/How\\_to\\_protect\\_sensitive\\_data\\_in\\_URL's](http://www.owasp.org/index.php/How_to_protect_sensitive_data_in_URL's)
- PCI Data Security Standard v1.1, [https://www.pcisecuritystandards.org/pdfs/pci\\_dss\\_v1-1.pdf](https://www.pcisecuritystandards.org/pdfs/pci_dss_v1-1.pdf)
- Bruce Schneier, <http://www.schneier.com/>
- CryptoAPI Next Generation, <http://msdn2.microsoft.com/en-us/library/aa376210.aspx>



## A9 – INSECURE COMMUNICATIONS

Applications frequently fail to encrypt network traffic when it is necessary to protect sensitive communications. Encryption (usually SSL) must be used for all authenticated connections, especially internet accessible web pages but backend connections as well. Otherwise, the application will expose an authentication or session token. In addition, encryption should be used whenever sensitive data, such as credit card or health information is transmitted. Applications that fall back or can be forced out of an encrypting mode can be abused by attackers.

The PCI standard requires that all credit card information being transmitted over the internet be encrypted.

### ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to insecure communications.

### VULNERABILITY

Failure to encrypt sensitive communications means that an attacker who can sniff traffic from the network will be able to access the conversation, including any credentials or sensitive information transmitted. Consider that different networks will be more or less susceptible to sniffing. However, it is important to realize that eventually a host will be compromised on almost every network, and attackers will quickly install a sniffer to capture the credentials of other systems.

Using SSL for communications with end users is critical, as they are very likely to be using insecure networks to access applications. Because HTTP includes authentication credentials or a session token with every single request, all authenticated traffic needs to go over SSL, not just the actual login request.

Encrypting communications with backend servers is also important. Although these networks are likely to be more secure, the information and credentials they carry is more sensitive and more extensive. Therefore using SSL on the backend is quite important.

Encrypting sensitive data, such as credit cards and social security numbers, has become a privacy and financial regulation for many organizations. Neglecting to use SSL for connections handling such data creates a compliance risk.

### VERIFYING SECURITY

The goal is to verify that the application properly encrypts all authenticated and sensitive communications.

Automated approaches: Vulnerability scanning tools can verify that SSL is used on the front end, and can find many SSL related flaws. However, these tools do not have access to backend connections and cannot verify that they are secure. Static analysis tools may be able to help with analyzing some calls to backend systems, but probably will not understand the custom logic required for all types of systems.

Manual approaches: Testing can verify that SSL is used and find many SSL related flaws on the front end, but the automated approaches are probably more efficient. Code review is quite efficient for verifying the proper use of SSL for all backend connections.

## PROTECTION

The most important protection is to use SSL on any authenticated connection or whenever sensitive data is being transmitted. There are a number of details involved with configuring SSL for web applications properly, so understanding and analyzing your environment is important.

- Use SSL for all connections that are authenticated or transmitting sensitive or value data, such as credentials, credit card details, health and other private information.
- Ensure that communications between infrastructure elements, such as between web servers and database systems are appropriately protected via the use of transport layer security or protocol level encryption for credentials and intrinsic value data.
- IE 7.0 provides a green bar for high trust SSL certificates, but this is not a suitable control to prove safe use of cryptography alone. It just means you paid a lot more for a certificate than most folks.
- Under PCI Data Security Standard requirement 4, you must protect cardholder data in transit. PCI DSS compliance is mandatory by 2008 for merchants and anyone else dealing with credit cards. In general, client, partner, staff and administrative online access to systems must be encrypted using SSL or similar. For more information, please see the PCI DSS Guidelines and implement controls as necessary.

## SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6430>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-4704>
- [http://www.schneier.com/blog/archives/2005/10/scandinavian\\_at\\_1.html](http://www.schneier.com/blog/archives/2005/10/scandinavian_at_1.html)

## REFERENCES

- OWASP *Testing Guide*, Testing for SSL / TLS, [https://www.owasp.org/index.php/Testing\\_for\\_SSL-TLS](https://www.owasp.org/index.php/Testing_for_SSL-TLS)
- OWASP Guide, [http://www.owasp.org/index.php/Guide\\_to\\_Cryptography](http://www.owasp.org/index.php/Guide_to_Cryptography)
- Foundstone - SSL Digger, [http://www.foundstone.com/index.htm?subnav=services/navigation.htm&subcontent=/services/overview\\_s3i\\_des.htm](http://www.foundstone.com/index.htm?subnav=services/navigation.htm&subcontent=/services/overview_s3i_des.htm)
- NIST, SP 800-52 Guidelines for the selection and use of transport layer security (TLS) Implementations, <http://csrc.nist.gov/publications/nistpubs/800-52/SP800-52.pdf>
- NIST SP 800-95 Guide to secure web services, <http://csrc.nist.gov/publications/drafts.html#sp800-95>



## A10 – FAILURE TO RESTRICT URL ACCESS

Frequently, the only protection for a URL is that links to that page are not presented to unauthorized users. However, a motivated, skilled, or just plain lucky attacker may be able to find and access these pages, invoke functions, and view data. Security by obscurity is not sufficient to protect sensitive functions and data in an application. Access control checks must be performed before a request to a sensitive function is granted which ensure the user is authorized to access that function.

### ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to failure to restrict URL access.

### VULNERABILITY

The primary attack method for this vulnerability is called "forced browsing", which encompasses guessing links and brute force techniques to find unprotected pages. Applications frequently allow access control code to evolve and spread throughout a codebase, resulting in a complex model that is difficult to understand for developers and security specialists alike. This complexity makes it likely that errors will occur and pages will be missed, leaving them exposed.

Some common examples of these flaws include:

- "Hidden" or "special" URLs, rendered only to administrators or privileged users in the presentation layer, but accessible to all users if they know it exists, such as `/admin/adduser.php` or `/approveTransfer.do`. This is particularly prevalent with menu code.
- Applications often allow access to "hidden" files, such as static XML or system generated reports, trusting security through obscurity to hide them.
- Code that enforces an access control policy but is out of date or insufficient. For example, imagine `/approveTransfer.do` was once available to all users, but since SOX controls were brought in, it is only supposed to be available to approvers. A fix might have been to not present it to unauthorized users, but no access control is actually enforced when requesting that page.
- Code that evaluates privileges on the client but not on the server, as in this [attack on MacWorld 2007](#), which approved "Platinum" passes worth \$1700 via JavaScript on the browser rather than on the server.

### VERIFYING SECURITY

The goal is to verify that access control is enforced consistently in the presentation layer and the business logic for all URLs in the application.

Automated approaches: Both vulnerability scanners and static analysis tools have difficulty with verifying URL access control, but for different reasons. Vulnerability scanners have difficulty guessing hidden pages and determining which pages should be allowed for each user, while static analysis engines struggle to identify custom access controls in the code and link the presentation layer with the business logic.

Manual approaches: The most efficient and accurate approach is to use a combination of code review and security testing to verify the access control mechanism. If the mechanism is centralized, the verification can be quite efficient. If the mechanism is distributed across an entire codebase, verification can be more time-consuming. If the mechanism is enforced externally (WebSEAL or SiteMinder), the configuration must be examined and tested.

## PROTECTION

Taking the time to plan authorization by creating a matrix to map the roles and functions of the application is a key step in achieving protection against unrestricted URL access. Web applications must enforce access control on every URL and business function. It is not sufficient to put access control into the presentation layer and leave the business logic unprotected. It is also not sufficient to check once during the process to ensure the user is authorized, and then not check again on subsequent steps. Otherwise, an attacker can simply skip the step where authorization is checked, and forge the parameter values necessary to continue on at the next step.

Enabling URL access control takes some careful planning. Among the most important considerations are:

- Ensure that the enforcement of your access control matrix is part of the business, architecture, and design of the application.
- Ensure that all URLs and business functions are protected by an effective access control mechanism that verifies the user's role and entitlements prior to any processing taking place. Make sure this is done during every step of the way, not just once towards the beginning of any multi-step process.
- Perform a penetration test prior to deployment or code delivery to ensure that the application cannot be misused by a motivated skilled attacker.
- **Do not** assume that users will be unaware of special or hidden URLs or APIs. Always ensure that administrative and high privilege actions are protected.
- **Block access** to all file types that your application should never serve. Ideally, this filter would follow the "accept known good" approach and only allow file types that you intend to serve, e.g., .html, .pdf, .php. This would then block any attempts to access log files, xml files, etc. that you never intend to serve directly.

## SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0147>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0131>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-1227>

## REFERENCES

- OWASP, [http://www.owasp.org/index.php/Testing\\_for\\_Directory\\_Traversal](http://www.owasp.org/index.php/Testing_for_Directory_Traversal)
- OWASP, [http://www.owasp.org/index.php/Forced\\_browsing](http://www.owasp.org/index.php/Forced_browsing)
- OWASP, [http://www.owasp.org/index.php/Guide\\_to\\_Authorization](http://www.owasp.org/index.php/Guide_to_Authorization)



## WHERE TO GO FROM HERE

The OWASP Top 10 is just the beginning of your web application security journey.

*The world's six billion people can be divided into two groups: group one, who know why every good software company ships products with known bugs; and group two, who don't. Those in group 1 tend to forget what life was like before our youthful optimism was spoiled by reality. Sometimes we encounter a person in group two ...who is shocked that any software company would ship a product before every last bug is fixed.*

*Eric Sink, Guardian May 25, 2006*

Most of your users and customers are in group two. How you deal with this problem is an opportunity to improve your code and the state of web application security in general. Billions of dollars are lost every year, and many millions of people suffer identity theft and fraud due to the vulnerabilities discussed in this document.

## FOR ARCHITECTS AND DESIGNERS

To properly secure your applications, you must know what you're securing (asset classification), know the threats and risks of insecurity, and address these in a structured way. Designing any non-trivial application requires a good dose of security.

- Ensure that you apply "just enough" security based upon threat risk modeling and asset classification
- Ask questions about business requirements, particularly missing non-functional requirements
- Work through the [OWASP Secure Software Contract Annex](#) with your customer
- Encourage safer design – include defense in depth and simpler constructs
- Ensure that you have considered confidentiality, integrity, and availability
- Ensure your designs are consistent with security policy and standards, such as COBIT or PCI DSS 1.1

## FOR DEVELOPERS

Many developers already have a good handle on web application security basics. To ensure effective mastery of the web application security domain requires practice. Anyone can destroy (i.e. perform penetration testing) – it takes a master to build secure software. Aim to become a master.

- Consider [joining OWASP](#) and attending [local chapter](#) meetings
- Ask for secure code training if you have a training budget. Ask for a training budget if you don't have one
- Design your features securely – consider defense in depth and simplicity in design
- Adopt coding standards which encourage safer code constructs
- Refactor existing code to use safer constructs in your chosen platform, such as parameterized queries



- Review the [OWASP Guide](#) and start applying selected controls to your code. Unlike most security guides, it is designed to help you build secure software, not break it
- Test your code for security defects and make this part of your unit and web testing regime
- Buy a copy of "*The Security Development Lifecycle*" (see [HOW1] in the book references) and adopt many of its practices.

## FOR OPEN SOURCE PROJECTS

Open source is a particular challenge for web application security. There are literally millions of open source projects, from one developer personal "itches" through to major projects such as Apache, Tomcat, and large scale web applications, such as PostNuke.

- Consider [joining OWASP](#) and attending [local chapter](#) meetings
- If your project has more than 4 developers, consider making at least one developer a security person
- Design your features securely – consider defense in depth and simplicity in design
- Adopt coding standards which encourage safer code constructs
- Adopt the responsible disclosure policy to ensure that security defects are handled properly
- Buy a copy of "*The Security Development Lifecycle*" and adopt many of its practices.

## FOR APPLICATION OWNERS

Application owners in commercial settings are often time and resource constrained. Application owners should:

- Work through the [OWASP Secure Software Contract Annex](#) with the software producers
- Ensure business requirements include non-functional requirements (NFRs) such as security requirements
- Encourage designs which include secure by default features, defense in depth and simplicity in design
- Employ (or train) developers who have a strong security background
- Test for security defects throughout the project: design, build, test, and deployment
- Allow resources, budget and time in the project plan to remediate security issues

## FOR C-LEVEL EXECUTIVES

Your organization must have a secure development life cycle (SDLC) in place that suits your organization. A reasonable SDLC not only includes testing for the Top 10, it includes:

- For off the shelf software, ensure purchasing policies and contracts include security requirements
- For custom code, adopt secure coding principles in your policies and standards



- Train your developers in secure coding techniques and ensure they keep these skills up to date
- Train your architects, designers, and business people in web application security fundamentals
- Adopt the responsible disclosure policy to ensure that security defects are handled properly

## REFERENCES

### OWASP PROJECTS

OWASP is the premier site for web application security. The [OWASP site](#) hosts many [projects](#), [forums](#), [blogs](#), [presentations](#), [tools](#), and [papers](#). OWASP hosts two major [web application security conferences](#) per year, and has over 80 local [chapters](#).

The following OWASP projects are most likely to be useful:

- [OWASP Guide to Building Secure Web Applications](#)
- [OWASP Testing Guide](#)
- [OWASP Code Review Project](#) (in development)
- [OWASP PHP Project](#) (in development)
- [OWASP Java Project](#)
- [OWASP .NET Project](#)

### BOOKS

- [GAL1] Gallagher T., Landauer L., Jeffries B., "*Hunting Security Bugs*", Microsoft Press, ISBN 073562187X
- [HOW1] Howard M., Lipner S., "*The Security Development Lifecycle*", Microsoft Press, ISBN 0735622140
- [HOW2] Howard M., Le Blanc D., "Writing Secure Code", 2nd ed., Microsoft Press, ISBN 0735617228
- [SCH1] Schneier B., "*Practical Cryptography*", Wiley, ISBN 047122894X

### WEB SITES

- OWASP, <http://www.owasp.org>
- MITRE, Common Weaknesses – Vulnerability Trends, <http://cwe.mitre.org/documents/vuln-trends.html>
- SANS Top 20, <http://www.sans.org/top20/>
- PCI Security Standards Council, publishers of the PCI standards, relevant to all organizations processing or holding credit card data, <https://www.pcisecuritystandards.org/>
- PCI DSS v1.1, [https://www.pcisecuritystandards.org/pdfs/pci\\_dss\\_v1-1.pdf](https://www.pcisecuritystandards.org/pdfs/pci_dss_v1-1.pdf)
- Build Security In, US CERT, <https://buildsecurityin.us-cert.gov/daisy/bsi/home.html>