

OWASP TOP 10

2007 RELEASE CANDIDATE 1



© 2002-2006 OWASP Foundation

This document is licensed under the Creative Commons [Attribution-ShareAlike 2.5](https://creativecommons.org/licenses/by-sa/2.5/) license. You must attribute your version to the OWASP Top 10 or the OWASP Foundation.



Table of Contents

Introduction.....	3
A1 Cross site scripting.....	6
A2 Injection Flaws.....	8
A3 Insecure Remote File Include	10
A4 Insecure direct object reference	12
A5 Cross Site Request Forgery.....	14
A6 Information Leakage and improper error handling.....	16
A7 Malformed input	18
A8 Broken Authorization	20
A9 Insecure Cryptography and communications	22
A10 Privilege Escalation.....	24
Where to go from here?	26
References.....	28

INTRODUCTION

Welcome to the OWASP Top 10 2007! This totally re-written edition details the most commonly discovered “weaknesses” (vulnerabilities), and demonstrates how to protect against them.

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 weaknesses. The Top 10 provides basic methods to protect against these weaknesses – 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 still 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 apps 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 these efforts.

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 MITRE results are as follows:

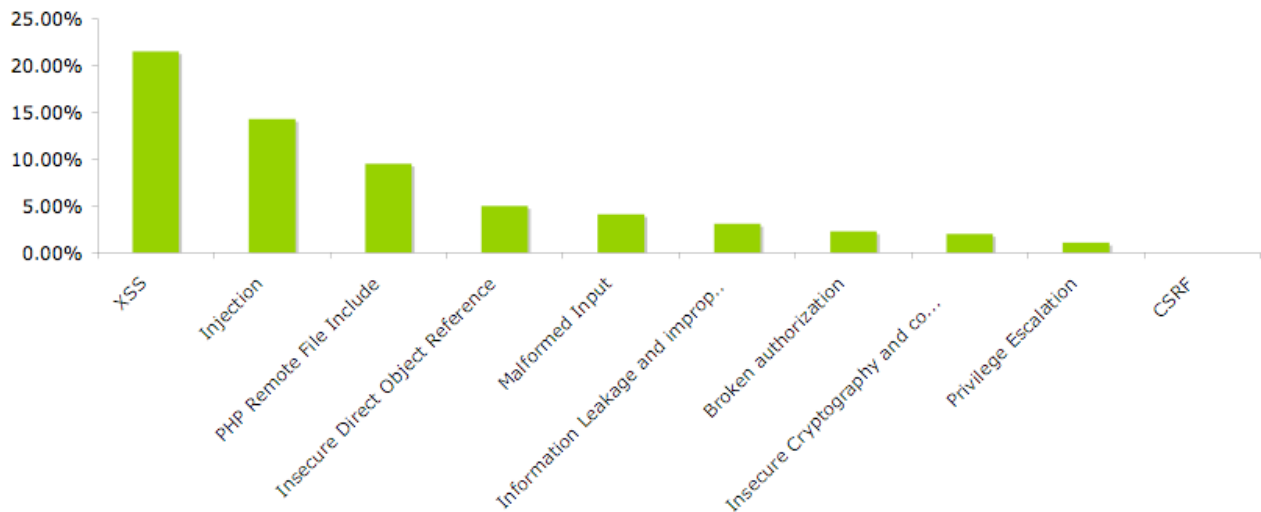


Figure 1: Top 10 Web application vulnerabilities for year to date 2006



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 protecting against CSRF today, particularly high value applications and applications which deal with sensitive data.

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

All of the protection recommendations provide solutions for the three most prevalent web application frameworks: PHP, ASP.NET and J2EE. Other common web application frameworks, such as Ruby on Rails or Perl, can easily adapt the recommendations to suit their specific needs.

BIASES

The Top 10 is necessarily biased 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 kiddy”) 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.

How this edition maps to the OWASP Top 10 2004, and the raw MITRE ranking

Top 10 2007	Top 10 2004	Mitre 2006 raw rank
A1. XSS	A4. XSS	1
A2. Injection Flaws	A6. Injection Flaws	2
A3. Remote file include (NEW)		3
A4. Improper direct object reference	A2. Broken access control A3. Broken authentication and session management	5
A5. CSRF (NEW)		36
A6. Information leakage and improper error handling	A7. Improper error handling	6
A7. Malformed input	A1. Invalidated input	7
A8. Broken authorization	A2. Broken access control	9
A9. Insecure storage and cryptography	A8. Insecure Storage	11
A10. Privilege escalation	A4. Authorization – Priv escalation	12

Top 10 2007	Top 10 2004	Mitre 2006 raw rank
Out of scope	A5. Buffer overflows	4, 8, and 10
Unranked	A9. Denial of service	17
Unranked	A10. Insecure configuration management	29

Why we have dropped some important issues

Buffer overflows, integer overflows and format string issues are extremely serious vulnerabilities for programs written in systems languages such as C or C++. Remediation is covered by the traditional non-web application security community, such as SANS, CERT, and system 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

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/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. However, when deploying your application, you should consult the OWASP Guide 3.0 and the OWASP Testing Guide 2.0 for detailed information regarding secure configuration management and testing:

<http://www.owasp.org/index.php/Configuration> (Guide 3.0)

http://www.owasp.org/index.php/Testing_for_infrastructure_configuration_management (Testing Guide)

THANKS

We thank the MITRE Project for making hard vulnerability discovery data freely available for use.



A1 CROSS SITE SCRIPTING

Cross site scripting, better known as XSS, is the most pernicious and easily found web application security issue. XSS allows attackers to deface web sites, insert hostile content, conduct phishing attacks, take over the user's browser using JavaScript malware, and force users to conduct commands not of their own choosing – an attack known as cross-site request forgeries, better known as CSRF.

ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to cross-site scripting.

AUTOMATIC CONTROLS OR DETECTION

Automatic controls: Web application firewalls, browsers with basic checks. Limited usefulness and capabilities

Automatic code detection: NP complete problem => low-lying fruit only. Code scanning tools will be able to highlight unsafe API, such as `print()` and `echo()` in PHP and `<%=` or `system.out.print()` in J2EE, but will not be able to detect system output problems where custom code exists (such as custom tag libraries) or if unusual methods are used to change browser content, such as dynamically replacing innerHTML (or aliasing it in heavy duty) or obfuscated JavaScript. Strongly urge manual code review and thorough testing to eliminate all issues.

VULNERABILITY

There are three known types of cross-site scripting: reflected, persistent, 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'];
```

Persistent XSS takes user 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 an individual. DOM injection can be triggered through reflected and persistent cross-site scripting, and are prevalent in Ajax applications, or can be used against any application suffering a reflected or persistent XSS.

PROTECTION

For all platforms:

- Strongly validate input data for length, type, syntax and business rules before accepting the data to be displayed or stored. Use “accept known good” validation strategy.
- Ensure that all user supplied data is displayed via escaping mechanisms:
 - Java: Use Struts output mechanisms such as `<bean:write ... >`, or JSTL `<c:out` is used with `escapeXML="true"`. Do NOT use `<%= ... %>`
 - .NET: Use the Microsoft Anti-Cross Site Scripting Library freely available from MSDN. **Do not** assign form fields data from the Request object: `username.Text = Request.QueryString("username");`
 - PHP: Ensure output is passed through `htmlspecialchars()` or `htmlspecialchars()`

It is insufficient to look for “<” “>” and other similar characters. This sanitization method is weak and has been attacked successfully. Many XSS attacks do not rely upon these characters.

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
- RSnake, XSS Cheat Sheet, <http://ha.ckers.org/xss.html>
- Klein, A., DOM Injection, <http://www.webappsec.org/projects/articles/071105.shtml>

ANTI-XSS LIBRARIES

.NET

Anti-XSS library

<http://www.microsoft.com/downloads/details.aspx?FamilyID=efb9c819-53ff-4f82-bfaf-e11625130c25&DisplayLang=en>

J2EE

OWASP Stinger Project – provides excellent input validation rules

http://www.owasp.org/index.php/OWASP_Stinger_Version_3

Output validation – use Struts `<bean:write ...>` or `<bean:message ...>` or JSTL `<c:out ...>` with `escapeXML="true"` (which is the default).

PHP

OWASP Filter Project

http://www.owasp.org/index.php/OWASP_PHP_Filters

Output encoding

Use `htmlspecialchars(content)` for output encoding, and `urlencode(url)` for URLs.



A2 INJECTION FLAWS

Injections, particularly SQL injections, are common in web applications. Injections are possible due to intermingling of user-supplied data within dynamic queries or within poorly constructed stored procedures.

Injections allow attackers:

- To create, read, update, or delete any arbitrary data available to the application
- In the worst case scenario, to completely compromise the database system and systems around it

ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to injection attacks.

AUTOMATIC CONTROLS OR DETECTION

Automatic controls: Web application firewalls. Limited usefulness and capabilities.

Automatic code detection: Code scanning tools can find insecure API. Replace with safer API.

VULNERABILITY

If user input is not validated and handled properly, it is vulnerable. Check if user input is supplied to dynamic queries, such as:

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

PROTECTION

The key method to avoid injections is the use of safer API, such as strongly typed parameterized queries and object relational mapping (ORM) libraries. These interfaces handle all data escaping, or do not require escaping.

In all languages,

- Strongly validate user data to eliminate suspect input using “accept known good” validation strategy
- Connect to the database using the least privilege possible
- Ensure detailed messages from the data layer are not shown to the user
- **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 dependant recommendation:

- J2EE – use strongly typed prepared statements, or ORMs such as Hibernate or Spring
- .NET – use strongly typed parameterized queries, such as `SqlCommand` with `SqlParameter`
- PHP – use PDO with strongly typed parameterized queries

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
- SQL Injection: <http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>
- Advanced SQL Injection, http://www.ngssoftware.com/papers/advanced_sql_injection.pdf
- More Advanced SQL Injection, http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf



A3 INSECURE REMOTE FILE INCLUDE

PHP is the most common web application language and framework in use today. By default, PHP allows file functions to access resources on the Internet. When PHP scripts allow user input to influence file names, remote file inclusion can be the result. This attack allows (but is not limited to):

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

ENVIRONMENTS AFFECTED

By default, PHP 4.0.4 and later and 5.x are vulnerable to remote file inclusion.

AUTOMATIC CONTROLS OR DETECTION

Automatic controls: Web application firewalls. Limited usefulness and capabilities. File system auditing is post-attack

Automatic code detection: Code scanning tools can find insecure API, but may have false positives. Check and remediate.

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.

PROTECTION

- 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` 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
- 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()
```

- Be extremely cautious if data is passed to `system()` `eval()` `passthru()` or ``` (the backtick operator)

SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5220>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5205>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5188>

REFERENCES

- OWASP Guide 3.0, http://www.owasp.org/index.php/File_System#Includes_and_Remote_files
- OWASP Testing Guide 2.0, 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)



A4 INSECURE DIRECT OBJECT REFERENCE

Insecure direct object references are a very common access control failure, and can occur in several ways:

- At the data layer of your application, via the use of improperly authorized identifiers. For example, allowing users to select records they should not have access to, by changing a simple numeric argument in the URL (e.g changing <http://www.example.org/foo.php?cartID=23> to <http://www.example.org/foo.php?cartID=24>)
- Via direct access to objects stored on the file system, and accessed via “..” or via guessing or brute forcing access to the object via URL references. Directory traversal (file access via “..” or variants) allows attackers access to controlled resources, such as password files, configuration files, database credentials or other files of the attacker’s choosing.

As this is such a simple attack, and the results so damaging, it is essential that applications protect against this attack.

ENVIRONMENTS AFFECTED

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

AUTOMATIC CONTROLS OR DETECTION

Automatic controls: File system or database auditing is post-attack. Web application firewalls must be trained for “correct” values in **all** forms and URLs. Unlikely to happen considering WAF rule set limitations (often limited to 256 rules). Browsers do not provide forced browsing controls.

Automatic code detection: Code scanning tools cannot detect this vulnerability, but might highlight weak API for file directory traversal vulnerability.

VULNERABILITY

The primary attack method for this vulnerability is called “forced browsing”, which encompasses from methods such as embedding links within e-mails through to advanced attacks using Ajax techniques and XMLHttpRequest to force a logged in session to access a particular resource or process a particular transaction.

If code allows arguments to process records such as:

<http://www.example.org/purchase.do?cartID=23>

Where the action processes shopping carts based upon an a simple ID, it may be possible to force the user to do something they had not intended to do.

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.

PROTECTION

- Strongly validate user input using “accept known good” as a strategy
- Ensure that GET requests only allow navigation, not permanent change
- Ensure all methods have appropriate access control to prevent unauthenticated or unauthorized users to access or process objects they are not authorized to access or change.
- For intrinsic value data and high value transactions, consider an out of band secondary step, such as e-mail notifications, SMS random token authentication of the transaction, or fob-based transaction signing to prevent forced browsing attacks from working silently and successfully in the background
- For highly sensitive intrinsic value data (such as health records or similar), review and implement COBIT or ISO 17799 or other mandated regulatory controls. Consider including audit records of access and detect unusual patterns of activity
- Hide server-side filenames from the user. See the [A3 PHP Remote File inclusion](#) for a code example
- J2EE, use Java’s security manager to allow access only to nominated files
- .NET, use virtual web sites to limit access, and ensure all sensitive files are outside of the webroot
- PHP, use open_basedir to prevent file access outside the chosen base directory of the application

Unix

- Consider using a chroot’ed jail or other virtualization technique to limit access and damage.
- Do not run the web server as root or other highly privileged user
- Ensure file system permissions do not allow the web server to read files outside the htdocs directory

Windows

- Consider upgrading to Windows Server 2003 R2 and IIS 6.0 or later. IIS 6.0 has a strong defense in depth strategy to prevent access to local files via the web server’s low privilege account.
- Move the webroot to a non-system drive. If all else fails, this is a last defense in depth control

SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4369>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3934>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3488>

REFERENCES

- OWASP, http://www.owasp.org/index.php/Testing_for_Directory_Traversal
- OWASP, http://www.owasp.org/index.php/Forced_browsing



A5 CROSS SITE REQUEST FORGERY

Cross-site request forgeries (CSRF) are not a new attack, but attacking applications is generally trivial and devastating if they are not specifically robust against CSRF. If your application processes value transactions or sensitive information, you must protect your application today.

ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to cross-site request forgeries.

Attacks are primarily mounted by XSS vulnerabilities. However, they can also be mounted through Microsoft Office compound format (such as Word or Excel files), [PDF XSS attack](#), [QuickTime](#), [Shockwave](#) and [Flash](#) files if the attacker can upload specially crafted files and the media file format allows for remote code execution.

This list is not exhaustive and may change as research continues into CSRF vulnerabilities.

AUTOMATIC CONTROLS OR DETECTION

Automatic controls: File system or database auditing is post-attack. Web application firewalls must be trained for “correct” values in **all** forms and URLs. Unlikely to happen considering limitations of WAF rule sets (often limited to 256 rules). Browsers do not currently provide any anti-CSRF controls.

Automatic code detection: Code scanning tools may detect this vulnerability by highlighting potential XSS vectors. Code must be verified manually if a custom tag library (J2EE) is used.

VULNERABILITY

A typical CSRF attack against forums might take the form of setting the user’s avatar to the application’s logout page. This will force an image to be displayed to all users which logs them out:

```

```

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

```

```

This attack is particularly devastating when coupled with Ajax techniques, which allow arbitrary code to run in the browser, including multiple invisible requests to the server. If the attacker can force victims to run their specialist Ajax attack, almost all the protections mentioned here would not be sufficient to protect against CSRF. The only method of preventing loss or damage in this case is strong transaction signing and process changes to prevent permanent or damaging loss from a single step attack.

PROTECTION

Simple CSRF attacks can be prevented. These strategies should be inherent in all web applications, regardless of value:

- Important! Ensure that there are no XSS vulnerabilities in your application. Also see A1 Cross site scripting
- Use random tokens (name and value):

```
<form action="/transfer.do" method="post">
<input type="hidden" name="437843674673" value="43847384383">
...
</form>
```

- Always ensure that authorization is appropriate – don't let anonymous or unauthorized users access privileged operations
- Never use GET requests (URLs) for sensitive data or to perform value transactions. Use only POST methods when processing sensitive data from the user. Requiring POST raises the bar for attackers but is ineffectual for Ajax enabled attacks
- For sensitive data or value transactions, only use SSL, transaction signing, or out of band confirmation steps, such as e-mail notifications, SMS random token authentication, or transaction signing fobs or a combination of all three.
- For high value applications, strongly consider stopping the transaction if data is detected in the GET, cookie, server or environment request objects along with logging / auditing the inappropriate attempt
- Strongly encourage your users to upgrade to modern browsers. In particular, IE 7.0 protects against many unknown CSRF and XSS attacks by preventing rendering of unsafe constructs regularly used by attackers

Advanced CSRF attacks can bypass many of these restrictions. However, these suggestions will diminish returns and attack surface area dramatically.

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

REFERENCES

- OWASP CSRF, <http://www.owasp.org/index.php/CSRF>
- OWASP CSRF Guard, http://www.owasp.org/index.php/CSRF_Guard
- RSnake, "What is CSRF?", <http://ha.ckers.org/blog/20061030/what-is-csrf/>



A6 INFORMATION LEAKAGE AND IMPROPER ERROR HANDLING

Systems can unintentionally leak information about their configuration, internal workings, or violate privacy through inadequate controls. Products 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, but 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.

AUTOMATIC CONTROLS OR DETECTION

Automatic controls: File system or database auditing is post-attack. Web application firewalls must be trained to prevent output leakage. Browsers do not currently provide any anti-leakage controls.

Automatic code detection: Code scanning tools may detect this vulnerability by highlighting potential stack traces, `system.out.print` and friends. However, common application design methods means that some instances will be missed.

VULNERABILITY

- Detailed error handling, where inducing an error displays too much information, such as stack traces, failed SQL statements, or other debugging information
- Internal workings may 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.

PROTECTION

To prevent information leakage is simple:

- 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 the same 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
- OWASP, http://www.owasp.org/index.php/Category:Sensitive_Data_Protection_Vulnerability



A7 MALFORMED INPUT

Poor input validation affects usability, availability, performance, and security. The mantra must be “correct then fast” when it comes to web applications. Validation rarely adds any appreciable overhead to a properly written system, and allows processing of valid and authorized transactions rather than wasting time on malformed or unauthorized data.

ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to malformed input.

AUTOMATIC CONTROLS OR DETECTION

Automatic controls: File system or database auditing is post-attack. Web application firewalls must be trained to understand all form and URL inputs (unlikely). Browsers do not currently provide any malformed input controls. . Some frameworks have limited protection against certain attacks. For example, .NET 1.1 and later does not allow “<script” to be placed in the URL. This, of course, can be bypassed.

Automatic code detection: NP complete problem => Code scanning tools will miss some validation hot spots or provide false negatives / positives. Best tools will trace data and check taint. However, they are not clever enough to understand if the validation mechanism is sufficient. Must be verified manually, particularly if custom validation regime in place.

VULNERABILITY

It is rare to find web applications that properly and comprehensively validate user input. Typically, data is taken from the user and stored or processed:

```
mysql_query("UPDATE table SET admin='".$$_REQUEST['isadmin']."' WHERE userid='".$$_REQUEST['uid']."'");
```

This is not limited to just PHP – it’s unfortunately extremely popular in almost all applications. Even well written applications may miss one or two variables. Often results from queries are not checked, and this can cause the application to crash if the application is fuzzed.

PROTECTION

- Validate data for type, length, syntax, signedness, and sanity. Use “accept known good” rather than “reject known bad” (akin to virus definitions, which by their nature are a Sypisphean waste of time) or “sanitize bad characters” or even worse, “no validation”. The XSS cheat sheet demonstrates why “sanitize bad characters” is doomed to fail.
- Client side validation is useful for the 99.99% of your users – they are legitimate and not trying to attack you. Consider implementing it. However, client side validation is trivial to bypass by attackers and thus must be replicated on the server side.
- Use tainting techniques to ensure that you only deal with “clean” data:

```
$clean = array();
```

```
$clean['isadmin'] = isset($_POST['isadmin']) ? ($_POST['isadmin'] == 'checked') : false;
```

...

PDO - query

This technique allows you to validate data exactly once and ensures that only clean data is used within business logic.

- .NET use the inbuilt form validation mechanisms
- J2EE use Struts validate.xml for a free validation mechanism. Spring.

SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6852>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6833>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6816>

REFERENCES

- OWASP, http://www.owasp.org/index.php/Category:OWASP_Validation_Project
- OWASP, http://www.owasp.org/index.php/Category:Input_Validation_Vulnerability
- OWASP, http://www.owasp.org/index.php/How_to_add_validation_logic_to_HttpServletRequest



A8 BROKEN AUTHORIZATION

Applications rarely – if ever – assert file system permissions, or check that dynamic private data, such as messages, accounts or similar are actually owned by the requestor. Coupled with high privilege web application server permissions, or all files being owned by a low privilege account, attackers can see information, or act upon records for which they have no permission. This is a very common penetration testing technique and is highly useful for motivated attackers.

ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to weak permissions. IIS 6.0 suffers less from this issue due to the very low permission account and high isolation of the default configuration. However, it can still suffer from these attacks.

AUTOMATIC CONTROLS OR DETECTION

Automatic controls: File system or database auditing is post-attack. Web application firewalls must be linked to authentication / authorization engine and be trained to understand acceptable paths for all role types (unlikely). Browsers do not currently provide any authorization checks.

Automatic code detection: NP complete problem => Code scanning tools will miss some authorization hot spots or provide false negatives / positives. However, they are not clever enough to understand if the authorization mechanism is sufficient. Must be verified manually, particularly if custom authentication / authorization regime in place, such as an external authorization engine such as SiteMinder or WebSEAL.

VULNERABILITY

- Often applications allow access to “hidden” resources, such as static XML or system generated reports, trusting security through obscurity to hide
- Many applications run under high privilege accounts which grants access to system files or other application files (such as the database server) when no access is necessary
- It is quite common for applications to offer a list of items, such as bank accounts or items for sale. Often, the item’s actual unique identifier is sent to the application for further processing. If this unique identifier is not tested for ownership, an attacker may be able to see other people’s items.

```
SELECT * FROM table WHERE id='id';
```

This example may allow records to be selected regardless of role.

PROTECTION

- During design, architects and designers should come up with a “least privilege” permission map and identify all likely resources
- During implementation, developers should ensure their code runs under a low privilege account, and positively asserts permissions on new files, whilst checking return results on opening existing files.

- For access to non-static resources, a full authorization matrix should be checked prior to granting access, for example ensuring that only administrators have access to streamed PDF reports or a customer has access only to their own record and no other.
- During packaging, the installer or installation method should ensure that files are created with lowest possible privileges, owned by a low privilege user account
- Ensure that all methods of obtaining user records such as SQL statements only allow records which the user has access to be shown

```
SELECT * FROM table WHERE access_roles IN ('role') AND id='id';
```

SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-0700>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1392>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3604>

REFERENCES

- OWASP, <http://www.owasp.org/index.php/Authorization>
- OWASP, <http://www.owasp.org/index.php/Category:Authorization>
- OWASP, http://www.owasp.org/index.php/Access_Control_In_Your_J2EE_Application



A9 INSECURE CRYPTOGRAPHY AND COMMUNICATIONS

Applications often implement very poorly designed cryptography, often using appropriate ciphers incredibly poorly, such as using 3DES or AES with no salts and storing the key in the clear. Even more frequently, applications do not use SSL for sensitive or value data, such as credentials, credit card details, health and other private information. Considering that SSL certificates cost around \$USD 20, it is difficult to fathom why so many sites fail such basic confidentiality requirements.

ENVIRONMENTS AFFECTED

Many web applications are vulnerable to insecure cryptography and communications, particularly at the design and deployment phases.

AUTOMATIC CONTROLS OR DETECTION

Automatic controls: File system or database auditing is post-attack. Web application firewalls are generally obscured by the use of SSL and will not detect encrypted attacks. Browsers do not currently provide any cryptographic controls other than rejecting unknown cipher suites or in some cases, malformed or revoked certificates. 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.

Automatic code detection: Code scanning tools can detect use of known cryptographic API, but cannot detect if it is being used properly, particularly if part of the system is not contained within the code (such as HSM interfaces, or access to encrypted databases). Must be verified manually, particularly if custom validation regime in place.

VULNERABILITY

There are several aspects to insecure cryptography use:

- Inappropriate use of home grown algorithms
- Insecure use of strong algorithms
- Continuing use of proven weak algorithms (MD5, SHA-1, RC0 .. RC4, etc)
- Lack of SSL for intrinsic value data (credentials, data protected under privacy laws, and so on)
- Lack of storage confidentiality, particularly as it relates to credit card storage, infrastructure credentials and keys to encrypted data stores

PROTECTION

As there are so many methods to use cryptography badly, the following statements should be taken as part of your testing regime to ensure secure cryptographic materials handling:

- There are probably less than 1000 skilled and qualified cryptographers on the planet. Very few of them have created widely used strong encryption and hashing algorithms that have been subject to worthy cryptanalysis. Under no circumstances should unqualified folks try to create cryptographic algorithms. Use only approved public algorithms such as AES, RSA public key cryptography, and SHA-256 or better for hashing.
- Retire weak algorithms, such as MD5 / SHA1 in favor of safer alternatives, such as SHA-256
- Generate keys offline and store private keys with extreme care
- Ensure that private keys are not stored on front-end web servers if at all possible

- 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 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
- Ensure that encrypted data stored on disk is not trivially decryptable. 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. Often indexes are stored on disk without encryption, thus negating any potential theft of disk controls.
- If at all possible, do not store credit card details. Under PCI DSS Guidelines (see references), you MUST NOT store credit card details without good business reason. Even with reasonable justification the controls required to protect such data are fairly hefty. Good practice is store the authorization number, not the credit card number. You are NEVER allowed to store the CVV number (the three digit number of the rear of the card). For more information, please refer to the PCI DSS Guidelines.

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 J2EE servlet containers, too)

REFERENCES

- OWASP, <http://www.owasp.org/index.php/Cryptography>
- OWASP, http://www.owasp.org/index.php/Insecure_Storage
- OWASP, http://www.owasp.org/index.php/How_to_protect_sensitive_data_in_URL%27s
- Bruce Schneier, Master cryptographer, <http://www.schneier.com/>
- CryptoAPI NG, <http://msdn2.microsoft.com/en-us/library/aa376210.aspx>



A10 PRIVILEGE ESCALATION

Privilege escalation is a very common web application security error. Application authors often consider “special” or “hidden” URLs to be hidden from normal or unauthenticated users, and thus fail to protect the application from motivated, skilled or just plain lucky attackers. Another common failing is to assume that all users are a single role (such as “user”) rather than create additional levels such as “Admin” or “approvers”. This is particularly prevalent in older applications upgraded with new privileged functionality.

ENVIRONMENTS AFFECTED

All web application frameworks are vulnerable to privilege escalation attacks.

AUTOMATIC CONTROLS OR DETECTION

Automatic controls: File system or database auditing is post-attack. Web application firewalls must be connected to the authorization engine and be trained to understand which roles may access all paths (unlikely). Browsers do not currently provide any authorization controls.

Automatic code detection: Code scanning tools can detect use of known authorization API, but cannot detect if it is being used properly, particularly if part of the system is not contained within the code (such as external authorization engines such as WebSEAL or SiteMinder). Often code scanners miss the **lack** of authorization code and thus leave the code vulnerable. Code architecture or design will also be missed by code scanners. Code must be verified manually, particularly if custom validation regime in place.

Automatic testing: fuzzers are excellent tools to discover this particular vulnerability, but will require time to sift through false positives.

VULNERABILITY

Applications are often at risk if bad assumptions are made about user’s access rights or privileges. Common examples are:

- “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
- Code that has an access control matrix but is out of date or insufficient. For example, if /approveTransfer.do was once available to all users, but since SOX controls were brought in, it is only supposed to be available to approvers.
- 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.

PROTECTION

There are several methods to prevent privilege escalation:

- Ensure that all value transactions have an effective access control mechanism that verifies the user’s role and entitlement prior to any processing taking place
- Ensure that an access control matrix is part of the business, architecture, and design of the application

- Do not send authorization tokens to the client, and do not accept authorization decisions from the client. Do not evaluate authorization on the client without also evaluating it on the server
- 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
- Perform a penetration test prior to deployment or code delivery to ensure that the application cannot be misused by a motivated skilled attacker

SAMPLES

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0147>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6878>
- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6192>

REFERENCES

- OWASP, <http://www.owasp.org/index.php/Authorization>
- OWASP, http://www.owasp.org/index.php/Testing_for_business_logic
- OWASP, http://www.owasp.org/index.php/Category:Access_Control_Vulnerability



WHERE TO GO FROM HERE?

The 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 thousands if not hundreds of thousands suffer identity theft and fraud.

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
- 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 secure policy and standards, such as COBIT or ISO 17799

We strongly recommend you buy a copy of "Security Development Lifecycle" and adopt many of its practices.

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 attend a [local chapter](#) meeting
- Ask for secure code training if you have a training budget. Ask for a training budget if you don't have one.
- Refactor your code to use safer constructs in your chosen platform, such as parameterized queries
- Download 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

We strongly recommend you buy a copy of "Security Development Lifecycle" 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 attend a [local chapter](#) meeting
- 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

We strongly recommend you buy a copy of “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:

- 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

We strongly recommend you buy a copy of “Security Development Lifecycle” and adopt many of its practices.

FOR C-LEVEL EXECUTIVES

Your organization must have a secure development life cycle in place that suits the 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 folks in web application security fundamentals
- Adopt the responsible disclosure policy to ensure that security defects are handled properly

We strongly recommend you buy a copy of “Security Development Lifecycle” and adopt many of its practices.



REFERENCES

OWASP PROJECTS

OWASP is the premier site for web application security. The [OWASP site](#) hosts many [projects](#), [forums](#), [blogs](#), [presentations](#), downloads, and [papers](#). OWASP hosts two major [web application security conferences](#) per year, and has over 100 local [chapters](#).

The following OWASP projects are most likely to be useful:

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

BOOKS

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

WEB SITES

- Build Security In, US CERT, <https://buildsecurityin.us-cert.gov/daisy/bsi/home.html>
- SANS Top 20, <http://www.sans.org/top20/>
- MITRE, Common Weaknesses – Vulnerability Trends, <http://cwe.mitre.org/documents/vuln-trends.html>