WEB APPLICATION TOP SECURITY CONTROLS

SECURITY POLICY

VER.	DRAFT	CHECK	APPROVAL	DATE	DESCRIPTION
0.3	Lutech			20/09/2013	First draft
DISTRIBUTION LIST					
Lutech, ACER					

TABLE OF CONTENTS

1 INTRODUCTION 3

2 GOALS 3

3 SECURITY CONTROLS 4

3.1 APPLICATION CONTROLS 4
3.1.1 CONFIGURATION & DEPLOY 4
3.1.2 IDENTITY MANAGEMENT 5
3.1.3 AUTHENTICATION 7
3.1.4 AUTHORIZATION 13
3.1.5 SESSION MANAGEMENT 16
3.1.6 DATA VALIDATION 28
3.1.7 CRYPTOGRAPHY 34
3.1.8 LOGGING AND AUDIT 35
3.1.9 CLIENT SIDE 37
3.1.10 WEB SERVICES 42
3.2 NETWORK CONTROLS 45
3.2.1 CONFIGURATION & DEPLOY 45
3.3 OPERATIVE SYSTEM CONTROLS 46
3.3.1 CONFIGURATION & DEPLOY 46
3.3.2 LOGGING AND AUDIT 47
3.4 DATABASE AND MIDDLEWARE CONTROLS 47
3.4.1 CONFIGURATION & DEPLOY 47
3.4.2 ERROR HANDLING 55
3.4.3 CRYPTOGRAPHY 58
3.4.4 LOGGING AND AUDIT 63
3.4.5 BACKUP & RECOVERY 64

1 INTRODUCTION

The main requirements to ensure the security of a web application (and more generally of an Information System) are based on:

- Confidentiality: The information managed by web application should be accessible directly or indirectly only to users who have the right and which are expressly authorized to know them;
- Integrity: the information managed by web application must be protected from alterations, such as changes, damage or improper cancellations, the work of unauthorized users, or even due to accidental events;
- Availability: the information managed by web application must always be accessible to users who are entitled, within the time and in the manner prescribed.

Confidentiality, Integrity and Availability usually given in the literature as "CIA triad" can be considered the fundamental pillars of Information Security and are the basis of other concepts or strictly associated with other security principles, such as privacy, reliability of an application and traceability.

2 GOALS

Purpose of this document is describes which functional and non-functional requirements are necessary when writing secure web applications by defining security controls for the following areas:

- Application controls (chapter 3.1)
- Network controls (chapter 3.2)
- Operative System controls (chapter 3.3)
- Database and Middleware controls (chapter 3.4)

The policy contains detailed security controls (e.g. use a digital certificate for secure transmission of data) originating from the main best practices, standards and laws:

- OWASP Top Ten, which provides security controls to mitigate most Web Application Security Risks
- OWASP Testing Guide
- OWASP Development Guide
- OWASP Application Security Verification Standard (ASVS)
- Regulation (EU) No 45/2001 of the European Parliament
- Regulation (EU) No 1227/2011 of the European Parliament
- Detailed technological control objectives derived from international standards
    1. ISO/IEC 27002 Information technology -- Security techniques -- Code of practice for information security management
    2. ISO/IEC 27034-1 Information technology -- Security techniques -- Application security, providing guidance to assist organizations in integrating security into the processes used for managing their applications.

3 SECURITY CONTROLS

3.1 Application Controls

3.1.1 Configuration & Deploy

3.1.1.1 Application admin interfaces

Description
An application may require an administrator interface to enable a privileged user to access functionality that may make changes to how the site functions. Such changes may include:

- user account provisioning
- site design and layout
- data manipulation
- configuration changes

In many instances, such interfaces are usually implemented with little thought of how to separate them from the normal users of the site.

Remediation
Enable external administrative interface access only to authorized IPs or use authorized internal accounts through VPN connection.

3.1.1.2 Database Credentials/Connection Strings

Description
Sometimes developers write database credentials and connection strings in clear text on source code.

```
...
private static String passwd = "mYv3rYSECr3tPWD";
...
db = MySQLdb.connect(host="db.server.com", user="admin",
passwd="NOBODYwillEVERguess", db="sales")
...
String url = "jdbc:mysql://" + serverName +
"/access?user=webclient&password=ILoveJuliet";
...
for i in 01 02 03 04; do ./remove_temp_files.sh --machine=appserv$i --
rootpassword="*d3H%sS-W"; done
...
```

Although software developers might not realize, in fact their source code is (or becomes) very often publicly available. It might be kept in a CVS repository, which is browseable on the Web. Other developers or code maintainers might send out parts of code by e-mail or post in on the Web without being aware that they reveal passwords. Even if source code is kept on a secure file system during development, it will almost certainly be moved around later (maybe some years later), as the team is reorganized, file servers are upgraded etc. - and new locations might not be protected anymore. Compiled program can be easily reverse engineered. And last but not least, passwords should be changed regularly - and changing hardcoded passwords could be a lot of hassle (recompilation of the source code, new release etc.). For all these reasons, software developers should avoid hardcoding secret information (passwords, etc.) in the source code.

Remediation
There isn't one simple solution to this problem, and it has to be solved on case-to-case basis. Below are suggestions of some of the possible solutions:

- Ask the user for a password. In many cases, it is the user who should know the password, not the program. This is the best solution, if only it can be applied (unfortunately, it won't work for batch scripts, web applications connecting to a database etc.).
- Keep secrets in a separate file. It is much easier to guard a separate config/properties file that is known to contain password(s), than to keep an eye on multiple locations of passwords in source code (that might get refactored etc.). Such file should have very restrictive file permissions, and should not end up in CVS etc. Watch out: if it is a web application, make sure that the password file is not just downloadadable by everyone
- Encrypt. If only your program already has/knows a secret (another password, decryption key etc.), you can use it to encrypt (and later decrypt) other secrets information. Libraries providing encryption algorithms like 3DES, RSA, Rijndael etc. are available for all common programming languages and platforms. Even encrypted, it's better to place the secret data in a separate file.
- Keep secrets in a database. If well protected, database is a good place to safely store passwords. Obviously, it won't work for the passwords that open the database (famous question of what came first, the hen or the egg), so you have to choose another solution.
- Use already existing credentials. For example AFS and/or Kerberos tokens are available when a script is executed from acrontab.
- Hash user's passwords. If your software stores passwords of your users (clients), keep them hashed instead of in plain text.

In general, do not include any credentials in your source code, including (but not limited to) usernames, passwords, certificates, token IDs, or phone numbers.
Such constants belong in properly protected properties or configuration files. ESAPI has an encrypted properties mechanism you can use to protect clear text credentials in such files.

REF: https://security.web.cern.ch/security/recommendations/en/password_alternatives.shtml

3.1.2 Identity Management

3.1.2.1 User Registration Process

Description
Registration process should provide no information regarding the existence of an user.
Furthermore it is necessary to check the validity of user data like email.

Remediation
Check user data to avoid the use of temporary email, fake data, etc.
Use a password strength checker for encourage user to use a strong password [see 3.1.2.2].
Use an automatic USER-ID [see 3.1.2.6].
The account will be active only after verification, for example can be sent an e-mail to the mailbox entered in the form to activate the account.

3.1.2.2 Password policy

Description
Weak passwords can be easily guessed or easily cracked.

Remediation
Registration form should encourage the use of strong passwords. In general can be introduce an internal control that checks password strength.
For example:

- The minimum length of the password must be at least eight characters.

- The format of the password must be alphanumeric, it must contain both alphabetic and numeric characters.
- Is recommended the use of combinations of alphabetic characters, whether uppercase (A to Z) and lowercase (a through z), numbers (0 to 9) and special characters (!, "£, $, etc.).
- Password must not contain the user ID of the user and repetitions or sequences of consecutive characters.
- The lifetime password must be 180 days.
- In case of change, the new password must be different from the five previous passwords used.

3.1.2.3 Password change policy

Description
Passwords must be easily changed. In order to verify the user's identity, password change must be made through a secure process.

Remediation
The password change must take place after entering the old password. Every time that a password change is made, the application must send a notification email to the user with password change occurred. The new password becomes active after clicking on the link in the e-mail notification. Each operation must be logged [see. 3.1.8.2].

3.1.2.4 Account Provisioning Process

Description
The process of creating a new user, must guarantee that insert information is correct and error free furthermore, the exchange of information must take place over a trust channel.

Remediation
The user credential during the registration process and during the log-in process should not be sent in clear [see 3.1.3.1].
After each new registration and change of user data it must be sent an e-mail to the user with a link to a "electronic document" page that summarizes the information. The pdf is digitally signed.
Each operation must be logged [see. 3.1.8.2].

3.1.2.5 User Enumeration and Guessable User Account

Description
Attackers verifies if is possible to collect a set of valid usernames by interacting with the authentication mechanism of the application. This will be useful for the brute force testing, in which verify if, given a valid username, it is possible to find the corresponding password. Often, web applications reveal when a username exists on system, either as a consequence of a misconfiguration or as a design decision. For example, sometimes, when we submit wrong credentials, we receive a message that states that either the username is present on the system or the provided password is wrong. The information obtained can be used by an attacker to gain a list of users on system. This information can be used to attack the web application, for example, through a brute force or default username/password attack.

Remediation
Ensure that the application answers in the same manner for every client request that produces a failed authentication. The application should answer in the same manner for every failed attempt of authentication.
For Example:

```
Credentials submitted are not valid
```

3.1.2.6 Weak/Unenforced username policy

Description
In some cases the userIDs are created with specific policies of administrator or company. For example we can view a user with a userID created in sequential order:

```
CN000100
CN000101
…
```

Sometimes the usernames are created with a REALM alias and then a sequential numbers:

```
R1001 - user 001 for REALM1
R2001 - user 001 for REALM2
```

Attackers can create simple shell scripts that compose UserIDs and submit a request with tool like wget to automate a web query to discern valid userIDs. To create a script we can also use Perl and CURL.
Other possibilities are:

- userIDs associated with credit card numbers, or in general numbers with a pattern
- userIDs associated with real names, e.g. if Freddie Mercury has a userID of "fmercury", then you might guess Roger Taylor to have the userID of "rtaylor"

Again, attackers can guess a username from the information received from an LDAP query or from Google information gathering, for example, from a specific domain. Google can help to find domain users through specific queries or through a simple shell script or tool.

Remediation
Use a strong username policy. Username must not be predictable and not linked to the individual, his data or the privileges associated to the account.

3.1.2.7 Password and account recovery process

Description
It must be defined a secure password e user-ID recovery process.

Remediation
The loss of the password involves sending an e-mail message containing the new password that must be changed by the user at first logon.
The outcome of the password change must always be notified by email.
The demand for new password and User ID can not be concurrent.
Each operation must be logged [see. 3.1.8.2].

3.1.3 Authentication

3.1.3.1 Credential transported over an Unencrypted Channel

Description
In order to log into a web site, usually, the user has to fill a simple form that transmits the inserted data with the POST method. What is less obvious is that this data can be passed using the HTTP protocol, that means in a non-secure way, or using HTTPS, which encrypts the data. To further complicate things, there is the possibility that the site has the login page accessible via HTTP (making us believe that the transmission is insecure), but then it actually sends data via HTTPS. In this case, attackers may retrieve sensitive information by simply sniffing the net with a sniffer tool.

There are four cases of study:

- Sending data with POST method through HTTP
- Sending data with POST method through HTTPS

- Sending data with POST method via HTTPS on a page reachable via HTTP
- Sending data with GET method through HTTPS

Sending data with POST method through HTTP
Suppose that the login page presents a form with fields User, Pass, and the Submit button to authenticate and give access to the application. If we look at the header of our request with WebScarab, we get something like this:

```
POST http://www.example.com/AuthenticationServlet HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14)
Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/index.jsp
Cookie: JSESSIONID=LVrRRQQXgwyWpW7QMnS49vtW1yBdqn98CGlkP4jTvVCGdyPkmn3S!
Content-Type: application/x-www-form-urlencoded
Content-length: 64

delegated_service=218&User=test&Pass=test&Submit=SUBMIT
```

From this example the tester can understand that the POST sends the data to the page www.example.com/AuthenticationServlet simply using HTTP. So, in this case, data are transmitted without encryption and a malicious user could read our username and password by simply sniffing the net with a tool like Wireshark.

Sending data with POST method through HTTPS
Suppose that our web application uses the HTTPS protocol to encrypt data we are sending (or at least for those relating to the authentication). In this case, trying to access the login page and to authenticate, the header of our POST request would be similar to the following:

```
POST https://www.example.com:443/cgi-bin/login.cgi HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14)
Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: https://www.example.com/cgi-bin/login.cgi
Cookie: language=English;
Content-Type: application/x-www-form-urlencoded
Content-length: 50

Command=Login&User=test&Pass=test
```

We can see that the request is addressed to www.example.com:443/cgi-bin/login.cgi using the HTTPS protocol. This ensures that our data are sent through an encrypted channel and that they are not readable by other people.

Sending data with POST method via HTTPS on a page reachable via HTTP
Now, suppose to have a web page reachable via HTTP and that then only data sent from the authentication form are shipped via HTTPS. This situation occurs, for example, when we are on a portal of a big company that offers various information and services publicly available, without identification, but which has also a private section accessible from the home page

through a login. So when we try to login, the header of our request will look like the following example:

```
POST https://www.example.com:443/login.do HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14)
Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/homepage.do
Cookie: SERVTIMSESSIONID=s2JyLkvDJ9ZhX3yr5BJ3DFLkdphH0QNSJ3VQB6pLhjkW6F
Content-Type: application/x-www-form-urlencoded
Content-length: 45

User=test&Pass=test&portal=ExamplePortal
```

We can see that our request is addressed to www.example.com:443/login.do using HTTPS. But if we have a look at the referrer field in the header (the page from which we came), it is www.example.com/homepage.do and is accessible via simple HTTP. Although we are sending data via HTTPS, this deployment can allow SSLStrip attacks (a type of Man-in-the-middle attack).

Sending data with GET method through HTTPS

In this last example, suppose that the application transfers data using the GET method. This method should never be used in a form that transmits sensitive data such as username and password, because they are displayed in clear in the URL and this entails a whole set of security issues. So this example is purely demonstrative, but, in reality, it is strongly suggested to use the POST method instead. This is because when the GET method is used, the URL that it requests is easily available from, for example, the server logs exposing your sensitive data to information leakage.

```
GET https://www.example.com/success.html?user=test&pass=test HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14)
Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: https://www.example.com/form.html
If-Modified-Since: Mon, 30 Jun 2008 07:55:11 GMT
If-None-Match: "43a01-5b-4868915f"
```

You can see that the data is transferred in clear text in the URL and not in the body of the message as before. But we must consider that TLS/SSL is a level 5 protocol, a lower level than HTTP, so the whole HTTP package is still encrypted and the URL is unreadable to an attacker. It is not a good practice to use the GET method in these cases, because the information contained in the URL can be stored in many servers such as proxy and web servers, leaking the privacy of the user's credentials.

Remediation

It is recommended to display a security warning message to the user whenever the non-TLS login page is requested. This security warning should urge the user to always type "HTTPS" into the browser or bookmark the secure login page. This approach will help educate users on the correct and most secure method of accessing the application.

The login page and all subsequent authenticated pages must be exclusively accessed over TLS. The initial login page, referred to as the "login landing page", must be served over TLS. Failure to utilize TLS for the login landing page allows an attacker to modify the login form action, causing the user's credentials to be posted to an arbitrary location. Failure to utilize TLS for authenticated pages after the login enables an attacker to view the unencrypted session ID and compromise the user's authenticated session.

https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

3.1.3.2 Weak Lockout mechanism

Description
Lockouts are counter-productive if implemented badly.
In general, the threshold governor should be implemented to prevent any one IP address monopolizing authentication CPU, network, and IO resources. If necessary, such as for compliance with a national security standard, a configurable soft lockout of approximately 15-30 minutes should apply, with an error message stating the reason and when the account will become active again.

Remediation
Ensure any lockout mechanism protects against both same username, many passwords and many usernames and same password attacks. This can only be done using the threshold governor approach as shown above.
The following timeouts are suggested, but any value will do as long as it severely impacts both vertical and horizontal brute force attacks:

- One failed attempt: At least 5 seconds
- Two failed attempts: At least 15 seconds
- Three failed attempts: At least 45 seconds

If there's an obvious brute force attempt (for example, more than 100 attempts per minute), the IP address and/or session should be banned for a period of time, such as 15 minutes. In such cases, error messages should make it clear why this action has been taken.
There should be a logical difference between administrative lockout and failed login lockout, so that re-enabling all users en masse does not unlock administratively locked users.
Applications should have the facility to alert the user (for example by e-mail) as to failed login attempts and allow them to change their password.
Each event must be logged [see. 3.1.8.2].

3.1.3.3 Bypassing authentication schema

Description
Problems related to Authentication Schema could be found at different stages of the software development life cycle (SDLC), like design, development, and deployment.
Examples of design errors include a wrong definition of application parts to be protected, the choice of not applying strong encryption protocols for securing authentication data exchange, and many more.
Problems in the development phase are, for example, the incorrect implementation of input validation functionalities, or not following the security best practices for the specific language.
In addition, there may be issues during the application setup (installation and configuration activities), due to a lack in required technical skills, or due to poor documentation available.
There are several methods to bypass the authentication schema in use by a web application:

- Direct page request (forced browsing)
- Parameter Modification
- Session ID Prediction
- SQL Injection

Direct page request

If a web application implements access control only on the login page, the authentication schema could be bypassed. For example, if a user directly requests a different page via forced browsing, that page may not check the credentials of the user before granting access. Attempt to directly access a protected page through the address bar in your browser to test using this method.

Parameter Modification
Another problem related to authentication design is when the application verifies a successful login on the basis of a fixed value parameters. A user could modify these parameters to gain access to the protected areas without providing valid credentials. In the example below, the "authenticated" parameter is changed to a value of "yes", which allows the user to gain access. In this example, the parameter is in the URL, but a proxy could also be used to modify the parameter, especially when the parameters are sent as form elements in a POST request.

```
http://www.site.com/page.asp?authenticated=no

$ nc www.site.com 80
GET /page.asp?authenticated=yes HTTP/1.0


HTTP/1.1 200 OK
Date: Sat, 11 Nov 2006 10:22:44 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
</HEAD><BODY>
<H1>You Are Auhtenticated</H1>
</BODY></HTML>
```

Session ID Prediction
Many web applications manage authentication using session identification values (SESSION ID). Therefore, if session ID generation is predictable, a malicious user could be able to find a valid session ID and gain unauthorized access to the application, impersonating a previously authenticated user.

SQL Injection (HTML Form Authentication)
SQL Injection is a widely known attack technique. We are not going to describe this technique in detail in this section because there is a full description on chapter 3.1.6.4.

Remediation
For "Direct page request" issue use an access control method on all pages.
For "Parameter Modification" issue do never considers user input as safe. So do not consider an user authenticated just because he have a parameter settled but delegate authentication controls to the server side.
For "Session ID Prediction" issue apply the remediation provided on chapter 3.1.5.1 and 3.1.5.6.
For "SQL Injection" apply the remediation provided on chapter 3.1.6.4.

3.1.3.4 Remember me functionality

Description
Implementing *remember me* functionality can be incredibly hard. Often software will just embed the username and password in headers or cookies, or a hash or crypto blob of the same. Based upon your risk profile, your application:

- High value applications must not possess *remember me* functionality

- Medium value applications should not contain *remember me* functionality. If present, the user must opt-in to *remember me*. The system should strongly warn users that *remember me* is insecure particularly on public computers
- Low value applications may include an opt-in *remember me* function. There should be a warning to the user that this option is insecure, particularly on public computers

Remediation
OWASP ESAPI reference implementation has basic *remember me* functionality based upon an AES encryption of the username and password, but this is not recommended for medium value systems, and should be used with care on low value systems.

3.1.3.5 Browser cache weakness

Description
Catching of form fields is present in most browsers. For form fields containing sensitive information (like credit card numbers) auto complete should be disabled using AUTOCOMPLETE=OFF attribute that can be used in every INPUT tag.

Remediation
Use AUTOCOMPLETE=OFF feature to disable browser catching on sensitive form fields:

```
<INPUT TYPE="password" AUTOCOMPLETE="off">
```

3.1.3.6 Weak security question / answer

Description
Questions and answers are back door credentials - they equate to the username and password for the user. Often such schemes use "Mother's Maiden Name" or other easily found information. If all systems use the same Q&As, it will be possible to break into many accounts using the same information.
They are unacceptable for the following reasons:

- Collection of information about people without their explicit consent (such as "Mother's maiden name") is illegal in most privacy regimes. Such collection is subject to privacy laws, review and correction by the subject, and so on
- IT Security Policies and standards such as ISO 27000 prohibit the clear text storage of passwords, but almost all Q&A schemes store both the question and answer in the clear
- The information in the answers is public for a goodly portion of the users of the Internet, and thus is found using public sources
- Secret Questions and Answers have been publicly abused, most notably by the attack on Sarah Palin's e-mail account, exposing her use of her Yahoo free mail account for government business

Remediation
Disable security question / answer mechanism.

3.1.3.7 Multi Factor Authentication

Description
Multi Factor Authentication is based on two or more factors. This can be accomplished through software (e.g. a software certificate), hardware (e.g. Smart Card or USB token) or any out-of-band approaches of one time passwords (e.g. via SMS or E-Mail). These methods have varying levels of security and impose different levels of inconvenience to the end user. An example is an ATM card. The card represents something you have, the PIN represents something you know, hence it is a two factor authentication. Mutual Authentication gives the user a simple way to verify that they are really connected to the intended online institution before providing sensitive information.

Techniques for authenticating a Web site are varied. The use of digital certificates coupled with encrypted communications (e.g. Secure Socket Layer, or SSL) is one; the use of shared secrets such as digital images is another. Digital certificate authentication is generally considered one of the stronger authentication technologies.

Remediation
Application must provide one o more multi-factor authentication mechanisms.

3.1.4 Authorization

3.1.4.1 Directory traversal / File Include

Description
Many web applications use and manage files as part of their daily operation. Using input validation methods that have not been well designed or deployed, an aggressor could exploit the system in order to read/write files that are not intended to be accessible. In particular situations, it could be possible to execute arbitrary code or system commands.
Traditionally, web servers and web applications implement authentication mechanisms in order to control access to files and resources. Web servers try to confine user's files inside a "root directory" or "web document root" which represent a physical directory on the file system. Users have to consider this directory as the base directory into the hierarchical structure of the web application. The definition of the privileges is made using Access Control Lists (ACL) that identify which users or groups are supposed to be able to access, modify, or execute a specific file on the server. These mechanisms are designed to prevent access to sensitive files from malicious users (for example, the common /etc/passwd file on a Unix-like platform) or to avoid the execution of system commands.
Many web applications use server-side scripts to include different kinds of files: it is quite common to use this method to manage graphics, templates, load static texts, and so on. Unfortunately, these applications expose security vulnerabilities if input parameters (i.e., form parameters, cookie values) are not correctly validated.
In web servers and web applications, this kind of problem arises in path traversal/file include attacks. By exploiting this kind of vulnerability, an attacker is able to read directories or files which he/she normally couldn't read, access data outside the web document root, or include scripts and other kinds of files from external websites.
This kind of attack is also known as the dot-dot-slash attack (../), directory traversal, directory climbing, or backtracking.

Remediation
A safer way for directory traversal protection it is to whitelist the files that are allowed to be included. Whitelisting is safer than blacklisting, so instead of trying to exclude all malicious combinations we will rather allow only a set of safe options to be used.
On a system configuration scale it's ideal to have each site running in a chroot jail. By locking down access to the user that your webserver runs under to a specific directory you can limit the impact of a traversal attack.

3.1.4.2 Bypassing authorization schema

Description
Bypassing authorization schema can allow attackers to obtains critical information or gain access to restricted areas or functions.

Remediation
For every specific role and for every function and request that the application executes during the post-authentication phase, it is necessary to verify if:

- Is it possible to access that resource even if the user is not authenticated?
- Is it possible to access that resource after the log-out?

- Is it possible to access functions and resources that should be accessible to a user that holds a different role/privilege?
- Try to access the application as an administrative user and track all the administrative functions. Is it possible to access administrative functions also if the tester is logged as a user with standard privileges?
- Is it possible to use these functionalities for a user with a different role and for whom that action should be denied?

3.1.4.3 Privilege Escalation

Description

Privilege escalation occurs when a user gets access to more resources or functionality than they are normally allowed, and such elevation/changes should have been prevented by the application. This is usually caused by a flaw in the application. The result is that the application performs actions with more privileges than those intended by the developer or system administrator.

The degree of escalation depends on which privileges the attacker is authorized to possess, and which privileges can be obtained in a successful exploit. For example, a programming error that allows a user to gain extra privilege after successful authentication limits the degree of escalation, because the user is already authorized to hold some privilege. Likewise, a remote attacker gaining superuser privilege without any authentication presents a greater degree of escalation.

Usually, we refer to vertical escalation when it is possible to access resources granted to more privileged accounts (e.g., acquiring administrative privileges for the application), and to horizontal escalation when it is possible to access resources granted to a similarly configured account (e.g., in an online banking application, accessing information related to a different user).

Remediation

Verify the user's authorization level for all requests.

For certain critical transactions provide strong authentication mechanisms through one-time password.

3.1.4.4 Direct Object References

Description

Applications frequently use the actual name or key of an object when generating web pages. Applications don't always verify the user is authorized for the target object. This results in an insecure direct object reference flaw. Attackers can easily manipulate parameter values to detect such flaws and code analysis quickly shows whether authorization is properly verified.

Such flaws can compromise all the data that can be referenced by the parameter. Unless the name space is sparse, it's easy for an attacker to access all available data of that type.

Attacker, who is an authorized system user, simply changes a parameter value that directly refers to a system object to another object the user isn't authorized for.

Remediation

Preventing insecure direct object references requires selecting an approach for protecting each user accessible object (e.g., object number, filename):

- Use per user or session indirect object references. This prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource's database key, a drop down list of six resources authorized for the current user could use the numbers 1 to 6 to indicate which value the user selected. The application has to map the per-user indirect reference back to the actual database key on the server. OWASP's ESAPI includes both sequential and random access reference maps that developers can use to eliminate direct object references

- Check access. Each use of a direct object reference from an untrusted source must include an access control check to ensure the user is authorized for the requested object

3.1.4.5 Failure to restrict access to authorized resource

Description

Applications are not always protecting page requests properly. Sometimes, URL protection is managed via configuration, and the system is misconfigured. Sometimes, developers must include the proper code checks, and they forget.

Detecting such flaws is easy. The hardest part is identifying which pages (URLs) exist to attack.

Such flaws may allow some or even all accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.

Attacker, who is an authorized system user, simply changes the URL to a privileged page. Is access granted? Anonymous users could access private pages that aren't protected.

Remediation

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 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

3.1.4.6 Failure to restrict access to authenticated resource

Description

Some applications check to see if a user is able to undertake a particular action, but then do not check if access to all resources required to complete the requested action is allowed. For example, forum software may check to see if a user is allowed to reply to a previous message, but then fails to check that the requested message is not within a protected or hidden forum or thread. Another example would be an Internet Banking application that checks to see if a user is allowed to transfer money, but does not validate that the "from account" is one of the user's accounts.

Does the application generate or allow access to static content that also contains sensitive information?

Is access to the static content controlled based on the current authenticated user?

Could an anonymous user with knowledge of resource naming retrieve that protected content?

Remediation

Use logical tier separation and patterns such as Model View Controller instead of directly accessing protected resources from the web tier.

Ensure that Model code checks to ensure that the requesting user should have access to the protected resource.

Ensure that the code requesting the resource has adequate error checking and does not assume that access will always be granted. Failure cases should be accounted for.

3.1.4.7 Failure to restrict access to static content

Description
Some applications generate static content such as a PDF transaction report and allow the underlying static web server to provide access to these files. Often this means a confidential report may be available to unauthorized access if a malicious user is able to determine a valid filename for a sensitive yet static resource.

Remediation
Ideally generate sensitive content on the fly and send directly to the browser rather than saving to the web server's file system.
If protecting static sensitive content, implement authorization checks to prevent anonymous access.
If you have to save to disk (not recommended), use random filenames (such as a GUID) and clean up temporary files regularly.
Do not store sensitive static content in web-accessible directory paths. Rather, store this content in non-web accessible directories and proxy access to this content through a handler that will implement proper authorization, logging, and other security functions. On the ASP.NET platform, the HTTPResponse.WriteFile() method can be used to implement this functionality. NOTE: Whenever accessing the file system from web-facing code be sure to guard against potential injection attacks.

3.1.5 Session Management

3.1.5.1 Bypassing Session Management Schema

Description
Cookies are used to implement session management and are described in detail in RFC 2965. In a nutshell, when a user accesses an application which needs to keep track of the actions and identity of that user across multiple requests, a cookie (or more than one) is generated by the server and sent to the client. The client will then send the cookie back to the server in all following connections until the cookie expires or is destroyed. The data stored in the cookie can provide to the server a large spectrum of information about who the user is, what actions he has performed so far, what his preferences are, etc. therefore providing a state to a stateless protocol like HTTP.
A typical example is provided by an online shopping cart. Throughout the session of a user, the application must keep track of his identity, his profile, the products that he has chosen to buy, the quantity, the individual prices, the discounts, etc. Cookies are an efficient way to store and pass this information back and forth (other methods are URL parameters and hidden fields).
Due to the importance of the data that they store, cookies are therefore vital in the overall security of the application. Being able to tamper with cookies may result in hijacking the sessions of legitimate users, gaining higher privileges in an active session, and in general influencing the operations of the application in an unauthorized way. In this test we have to check whether the cookies issued to clients can resist a wide range of attacks aimed to interfere with the sessions of legitimate users and with the application itself. The overall goal is to be able to forge a cookie that will be considered valid by the application and that will provide some kind of unauthorized access (session hijacking, privilege escalation, ...). Usually the main steps of the attack pattern are the following:

- cookie collection: collection of a sufficient number of cookie samples
- cookie reverse engineering: analysis of the cookie generation algorithm
- cookie manipulation: forging of a valid cookie in order to perform the attack. This last step might require a large number of attempts, depending on how the cookie is created (cookie brute-force attack)

Another pattern of attack consists of overflowing a cookie. Strictly speaking, this attack has a different nature, since here we are not trying to recreate a perfectly valid cookie. Instead, our

goal is to overflow a memory area, thereby interfering with the correct behavior of the application and possibly injecting (and remotely executing) malicious code.

All interaction between the client and application should be tested at least against the following criteria:

- Are all Set-Cookie directives tagged as Secure?
- Do any Cookie operations take place over unencrypted transport?
- Can the Cookie be forced over unencrypted transport?
- If so, how does the application maintain security?
- Are any Cookies persistent?
- What Expires= times are used on persistent cookies, and are they reasonable?
- Are cookies that are expected to be transient configured as such?
- What HTTP/1.1 Cache-Control settings are used to protect Cookies?
- What HTTP/1.0 Cache-Control settings are used to protect Cookies?

Cookie collection

The first step required in order to manipulate the cookie is obviously to understand how the application creates and manages cookies. For this task, we have to try to answer the following questions:

- How many cookies are used by the application?
  Surf the application. Note when cookies are created. Make a list of received cookies, the page that sets them (with the set-cookie directive), the domain for which they are valid, their value, and their characteristics.
- Which parts of the the application generate and/or modify the cookie?
  Surfing the application, find which cookies remain constant and which get modified. What events modify the cookie?
- Which parts of the application require this cookie in order to be accessed and utilized?
  Find out which parts of the application need a cookie. Access a page, then try again without the cookie, or with a modified value of it. Try to map which cookies are used where.
  A spreadsheet mapping each cookie to the corresponding application parts and the related information can be a valuable output of this phase.

Session Analysis

The session tokens (Cookie, SessionID or Hidden Field) themselves should be examined to ensure their quality from a security perspective. They should be tested against criteria such as their randomness, uniqueness, resistance to statistical and cryptographic analysis and information leakage.

Token Structure & Information Leakage

The first stage is to examine the structure and content of a Session ID provided by the application. A common mistake is to include specific data in the Token instead of issuing a generic value and referencing real data at the server side. If the Session ID is clear-text, the structure and pertinent data may be immediately obvious as the following:

```
192.168.100.1:owaspuser:password:15:58
```

If part or the entire token appears to be encoded or hashed, it should be compared to various techniques to check for obvious obfuscation. For example the string "192.168.100.1:owaspuser:password:15:58" is represented in Hex, Base64 and as an MD5 hash:

```
Hex    3139322E3136382E3130302E313A6F77617370757365723A70617373776F72643A31353A3538
Base64 MTkyLjE2OC4xMDAuMTpvd2FzcHVzZXI6cGFzc3dvcmQ6MTU6NTg=
MD5    01c2fc4f0a817afd8366689bd29dd40a
```

Having identified the type of obfuscation, it may be possible to decode back to the original data. In most cases, however, this is unlikely. Even so, it may be useful to enumerate the

encoding in place from the format of the message. Furthermore, if both the format and obfuscation technique can be deduced, automated brute-force attacks could be devised. Hybrid tokens may include information such as IP address or User ID together with an encoded portion, as the following:

```
owaspuser:192.168.100.1: a7656fafe94dae72b1e1487670148412
```

Having analyzed a single session token, the representative sample should be examined. A simple analysis of the tokens should immediately reveal any obvious patterns. For example, a 32 bit token may include 16 bits of static data and 16 bits of variable data. This may indicate that the first 16 bits represent a fixed attribute of the user – e.g. the username or IP address. If the second 16 bit chunk is incrementing at a regular rate, it may indicate a sequential or even time-based element to the token generation. See examples. If static elements to the Tokens are identified, further samples should be gathered, varying one potential input element at a time. For example, login attempts through a different user account or from a different IP address may yield a variance in the previously static portion of the session token. The following areas should be addressed during the single and multiple Session ID structure testing:

- What parts of the Session ID are static?
- What clear-text confidential information is stored in the Session ID? E.g. usernames/UID, IP addresses
- What easily decoded confidential information is stored?
- What information can be deduced from the structure of the Session ID?
- What portions of the Session ID are static for the same login conditions?
- What obvious patterns are present in the Session ID as a whole, or individual portions?

Session ID Predictability and Randomness

Analysis of the variable areas (if any) of the Session ID should be undertaken to establish the existence of any recognizable or predictable patterns. These analyses may be performed manually and with bespoke or OTS statistical or cryptanalytic tools in order to deduce any patterns in the Session ID content. Manual checks should include comparisons of Session IDs issued for the same login conditions – e.g., the same username, password, and IP address. Time is an important factor which must also be controlled. High numbers of simultaneous connections should be made in order to gather samples in the same time window and keep that variable constant. Even a quantization of 50ms or less may be too coarse and a sample taken in this way may reveal time-based components that would otherwise be missed. Variable elements should be analyzed over time to determine whether they are incremental in nature. Where they are incremental, patterns relating to absolute or elapsed time should be investigated. Many systems use time as a seed for their pseudo-random elements. Where the patterns are seemingly random, one-way hashes of time or other environmental variations should be considered as a possibility. Typically, the result of a cryptographic hash is a decimal or hexadecimal number so should be identifiable. In analyzing Session ID sequences, patterns or cycles, static elements and client dependencies should all be considered as possible contributing elements to the structure and function of the application.

Are the Session IDs provably random in nature? I.e., can the resulting values be reproduced?

Do the same input conditions produce the same ID on a subsequent run?

Are the Session IDs provably resistant to statistical or cryptanalysis?

What elements of the Session IDs are time-linked?

What portions of the Session IDs are predictable?

Can the next ID be deduced, given full knowledge of the generation algorithm and previous IDs?

Cookie reverse engineering

Now that we have enumerated the cookies and have a general idea of their use, it is time to have a deeper look at cookies that seem interesting. Which cookies are we interested in? A cookie, in order to provide a secure method of session management, must combine several

characteristics, each of which is aimed at protecting the cookie from a different class of attacks. These characteristics are summarized below:

- Unpredictability: a cookie must contain some amount of hard-to-guess data. The harder it is to forge a valid cookie, the harder is to break into legitimate user's session. If an attacker can guess the cookie used in an active session of a legitimate user, he/she will be able to fully impersonate that user (session hijacking). In order to make a cookie unpredictable, random values and/or cryptography can be used.
- Tamper resistance: a cookie must resist malicious attempts of modification. If we receive a cookie like IsAdmin=No, it is trivial to modify it to get administrative rights, unless the application performs a double check (for instance, appending to the cookie an encrypted hash of its value)
- Expiration: a critical cookie must be valid only for an appropriate period of time and must be deleted from disk/memory afterwards, in order to avoid the risk of being replayed. This does not apply to cookies that store non-critical data that needs to be remembered across sessions (e.g., site look-and-feel)
- "Secure" flag: a cookie whose value is critical for the integrity of the session should have this flag enabled in order to allow its transmission only in an encrypted channel to deter eavesdropping.

The approach here is to collect a sufficient number of instances of a cookie and start looking for patterns in their value. The exact meaning of "sufficient" can vary from a handful of samples, if the cookie generation method is very easy to break, to several thousands, if we need to proceed with some mathematical analysis (e.g., chi-squares, attractors. See later for more information).
It is important to pay particular attention to the workflow of the application, as the state of a session can have a heavy impact on collected cookies: a cookie collected before being authenticated can be very different from a cookie obtained after the authentication.
Another aspect to keep into consideration is time: always record the exact time when a cookie has been obtained, when there is the possibility that time plays a role in the value of the cookie (the server could use a timestamp as part of the cookie value). The time recorded could be the local time or the server's timestamp included in the HTTP response (or both).
Analyzing the collected values, try to figure out all variables that could have influenced the cookie value and try to vary them one at the time. Passing to the server modified versions of the same cookie can be very helpful in understanding how the application reads and processes the cookie.
Examples of checks to be performed at this stage include:

- What character set is used in the cookie? Has the cookie a numeric value? alphanumeric? hexadecimal? What happens if we insert in a cookie characters that do not belong to the expected charset?
- Is the cookie composed of different sub-parts carrying different pieces of information? How are the different parts separated? With which delimiters? Some parts of the cookie could have a higher variance, others might be constant, others could assume only a limited set of values. Breaking down the cookie to its base components is the first and fundamental step. An example of an easy-to-spot structured cookie is the following:

```
ID=5a0acfc7ffeb919:CR=1:TM=1120514521:LM=1120514521:S=j3am5KzC4v01ba3q
```

In this example we see 5 different fields, carrying different types of data:

- ID – hexadecimal
- CR – small integer
- TM and LM – large integer. (And curiously they hold the same value. Worth to see what happens modifying one of them)
- S – alphanumeric

Even when no delimiters are used, having enough samples can help. As an example, let's look at the following series:

```
0123456789abcdef
```

Brute Force Attacks
Brute force attacks inevitably lead on from questions relating to predictability and randomness. The variance within the Session IDs must be considered together with application session durations and timeouts. If the variation within the Session IDs is relatively small, and Session ID validity is long, the likelihood of a successful brute-force attack is much higher. A long Session ID (or rather one with a great deal of variance) and a shorter validity period would make it far harder to succeed in a brute force attack.
How long would a brute-force attack on all possible Session IDs take?
Is the Session ID space large enough to prevent brute forcing? For example, is the length of the key sufficient when compared to the valid life-span?
Do delays between connection attempts with different Session IDs mitigate the risk of this attack?

Remediation
If you have access to the session management schema implementation, you can check for the following:

- Random Session Token: the Session ID or Cookie issued to the client should not be easily predictable (don't use linear algorithms based on predictable variables such as the client IP address) [3.1.5.6]. The use of cryptographic algorithms with key length of 256 bits is encouraged (like AES)
- Token length: session ID will be at least 50 characters length [3.1.5.6]
- Session Time-out: session token should have a defined time-out (it depends on the criticality of the application managed data)
- Cookie configuration:
  - non-persistent: only RAM memory
  - secure (set only on HTTPS channel): Set Cookie: cookie=data; path=/; domain=.aaa.it; secure
  - HTTPOnly (not readable by a script): Set Cookie: cookie=data; path=/; domain=.aaa.it; HTTPOnly

3.1.5.2 Cookie Attributes (HTTP Only, Secure)

Description
The importance of secure use of Cookies cannot be understated, especially within dynamic web applications, which need to maintain state across a stateless protocol such as HTTP. Cookies are set by the application using the Set-Cookie directive in the application's HTTP response, and is usually in a name=value format (if cookies are enabled and if they are supported, which is the case for all modern web browsers). Once an application has told the browser to use a particular cookie, the browser will send this cookie in each subsequent request. A cookie can contain data such as personal information, user IDs, etc. Due to the sensitive nature of information in cookies, they are typically encoded or encrypted in an attempt to protect the information they contain. Often, multiple cookies will be set (separated by a semicolon) upon subsequent requests.
It is possible to set several cookie attributes:

- Secure. This attribute tells the browser to only send the cookie if the request is being sent over a secure channel such as HTTPS. This will help protect the cookie from being passed over unencrypted requests. If the application can be accessed over both HTTP and HTTPS, then there is the potential that the cookie can be sent in clear text.
- HttpOnly. This attribute is used to help prevent attacks such as cross-site scripting, since it does not allow the cookie to be accessed via a client side script such as JavaScript. Note that not all browsers support this functionality.
- Domain. This attribute is used to compare against the domain of the server in which the URL is being requested. If the domain matches or if it is a sub-domain, then the path attribute will be checked next.

Note that only hosts within the specified domain can set a cookie for that domain. Also the domain attribute cannot be a top level domain (such as .gov or .com) to prevent servers from setting arbitrary cookies for another domain. If the domain attribute is not set, then the hostname of the server which generated the cookie is used as the default value of the domain. For example, if a cookie is set by an application at app.mydomain.com with no domain attribute set, then the cookie would be resubmitted for all subsequent requests for app.mydomain.com and its subdomains (such as hacker.app.mydomain.com), but not to otherapp.mydomain.com. If a developer wanted to loosen this restriction, then he could set the domain attribute to mydomain.com. In this case the cookie would be sent to all requests for app.mydomain.com and its subdomains, such as hacker.app.mydomain.com, and even bank.mydomain.com. If there was a vulnerable server on a subdomain (for example, otherapp.mydomain.com) and the domain attribute has been set too loosely (for example, mydomain.com), then the vulnerable server could be used to harvest cookies (such as session tokens).

- Path. In addition to the domain, the URL path can be specified for which the cookie is valid. If the domain and path match, then the cookie will be sent in the request.
  Just as with the domain attribute, if the path attribute is set too loosely, then it could leave the application vulnerable to attacks by other applications on the same server. For example, if the path attribute was set to the web server root "/", then the application cookies will be sent to every application within the same domain.
- Expires. This attribute is used to set persistent cookies, since the cookie does not expire until the set date is exceeded. This persistent cookie will be used by this browser session and subsequent sessions until the cookie expires. Once the expiration date has exceeded, the browser will delete the cookie. Alternatively, if this attribute is not set, then the cookie is only valid in the current browser session and the cookie will be deleted when the session ends.

Remediation
Use cookie flags to ensure a good level of security to the application.

3.1.5.3 Session Fixation

Description
Session fixation vulnerabilities occur when:

- A web application authenticates a user without first invalidating the existing session ID, thereby continuing to use the session ID already associated with the user.
- An attacker is able to force a known session ID on a user so that, once the user authenticates, the attacker has access to the authenticated session.

In the generic exploit of session fixation vulnerabilities, an attacker creates a new session on a web application and records the associated session identifier. The attacker then causes the victim to authenticate against the server using the same session identifier, giving the attacker access to the user's account through the active session.
Furthermore, the issue described above is problematic for sites which issue a session identifier over HTTP and then redirect the user to a HTTPS login form. If the session identifier is not reissued upon authentication, the identifier may be eavesdropped and may be used by an attacker to hijack the session.

Remediation
Some platforms make it easy to protect against Session Fixation, while others make it a lot more difficult. In most cases, simply discarding any existing session is sufficient to force the framework to issue a new sessionid cookie, with a new value. Unfortunately, some platforms, notably Microsoft ASP, do not generate new values for sessionid cookies, but rather just associate the existing value with a new session. This guarantees that almost all ASP apps will be vulnerable to session fixation, unless they have taken specific measures to protect against it.

Here is some sample code to illustrate an approach to preventing session fixation attacks in ASP. The idea is that, since ASP prohibits write access to the ASPSESSIONIDxxxxx cookie, and will not allow us to change it in any way, we have to use an additional cookie that we do have control over to detect any tampering. So, we set a cookie in the user's browser to a random value, and set a session variable to the same value. If the session variable and the cookie value ever don't match, then we have a potential fixation attack, and should invalidate the session, and force the user to log on again.
Here is a sample implementation:

```
AntiFixation.asp:

<%

   ' This routine is intended to provide a degree of protection
   ' against Session Fixation attacks in classic ASP

   ' Session fixation attacks are a problem in ASP, since ASP does not
   ' allow you any access to the ASPSESSIONIDxxx cookie. Even invalidating
   ' the session does not alter the value of this cookie, preventing
   ' implementation of best practice recommendations, such as
   ' issuing new session cookies when the session is authenticated, or
   ' invalidated.

   ' The basic premise of this routine is that we create a cookie that
   ' we CAN control, e.g. ASPFIXATION, and assign a random value to this
   ' cookie when the session is authenticated. On subsequent pages, we
   ' check the value of this cookie against the same variable stored in
   ' the user's session. If they do not match, access is denied.
   ' When the user logs out, the session should be invalidated, and so
   ' by default, the cookie no longer matches the value in the session.

   Private Function RandomString(l)
       Dim value, i, r
       Randomize
       For i = 0 To l
           r = Int(Rnd * 62)
           If r<10 Then
               r = r + 48
           ElseIf r<36 Then
               r = (r - 10) + 65
           Else
               r = (r - 10 - 26) + 97
           End If
           value = value & Chr(r)
       Next
       RandomString = value
   End Function

   ' This routine should be called after the user has been authenticated.
   ' It is expected that the session has been invalidated prior to this call.
   Public Sub AntiFixationInit()
       Dim value
       value = RandomString(10)
       Response.Cookies("ASPFIXATION") = value
       Session("ASPFIXATION") = value
   End Sub

   Public Sub AntiFixationVerify(LoginPage)
       Dim cookie_value, session_value
       cookie_value = Request.Cookies("ASPFIXATION")
       session_value = Session("ASPFIXATION")
       If cookie_value <> session_value Then
           Response.redirect(LoginPage)
       End If
   End Sub
```

```
%>
```

Include the following lines in your login page:

```
<!--#include virtual="/AntiFixation.asp" -->
```

and, when your user is successfully authenticated:

```
AntiFixationInit()
```

All other private pages (i.e. only accessible by an authenticated user) should include the following lines (preferably as the first couple of lines in the file):

```
<!--#include virtual="/AntiFixation.asp" -->
<% AntiFixationVerify("login.asp") %>
```

In this case, any requests that do not contain a valid ASPFIXATION cookie will be redirected to the page indicated, in this case "login.asp". Note that we do not automatically invalidate the session, since that would allow a denial of service attack against the legitimate user. If one were concerned about brute force attacks against the fixation cookie, one could either make the random value longer, and/or use a counter in the session to detect repeated attacks, and invalidate the session if a threshold is exceeded.

In high-value applications, session tokens should be regenerated after a certain number of requests.
In high-value applications, session tokens should be regenerated after a certain period of time.
High value applications should force users to re-authenticate before viewing high-value resources or complete high-value transactions.

3.1.5.4 Session Variables

Description
If possible, the application should conduct all web traffic using HTTPS. If this is not possible, the application should, at minimum protect sensitive content and pages used to engage in sensitive transactions with HTTPS. If HTTPS is not used to protect the entire web session an additional session identifier should be incorporated that is only transferred over HTTPS. Additional application level checks for HTTPS pages should verify that this session identifier has been provided and matches the normal session being managed by the web or application server.

Remediation
If URL parameters must be used to store and transmit session identifiers, use POST requests for all requests to the server. This can make seemingly simple operations such as linking to pages in an application complicated so the use of browser cookies is typically preferable for session state maintenance identifiers.
If Cookies are used to store and transmit session identifiers over HTTPS they should be marked as 'Secure' so that they are not served over non-SSL tunnels.
Provide an explicit way for users to log out of the application and ensure that this log out routine explicitly expires and destroys the session [3.1.5.7].
The information here relates to how transport security applies to the transfer of sensitive Session ID data rather than data in general, and may be stricter than the caching and transport policies applied to the data served by the site. Using a personal proxy, it is possible to ascertain the following about each request and response:

- Protocol used (e.g., HTTP vs. HTTPS)
- HTTP Headers
- Message Body (e.g., POST or page content)

Each time Session ID data is passed between the client and the server, the protocol, cache, and privacy directives and body should be examined. Transport security here refers to Session IDs passed in GET or POST requests, message bodies, or other means over valid HTTP requests.

3.1.5.5 Cross Site Request Forgery (CSRF)

Description

CSRF relies on the following:

1) Web browser behavior regarding the handling of session-related information such as cookies and http authentication information
2) Knowledge by the attacker of valid web application URLs
3) Application session management relying only on information which is known by the browser
4) Existence of HTML tags whose presence cause immediate access to an http[s] resource (for example the image tag img)

Points 1, 2, and 3 are essential for the vulnerability to be present, while point 4 is accessory and facilitates the actual exploitation, but is not strictly required.
Point 1) Browsers automatically send information which is used to identify a user session. Suppose site is a site hosting a web application, and the user victim has just authenticated himself to site. In response, site sends victim a cookie which identifies requests sent by victim as belonging to victim's authenticated session. Basically, once the browser receives the cookie set by site, it will automatically send it along with any further requests directed to site.
Point 2) If the application does not make use of session-related information in URLs, then it means that the application URLs, their parameters, and legitimate values may be identified (either by code analysis or by accessing the application and taking note of forms and URLs embedded in the HTML/JavaScript).
Point 3) By "known by the browser", we mean information such as cookies, or http-based authentication information (such as Basic Authentication; NOT form-based authentication), which are stored by the browser and subsequently resent at each request directed towards an application area requesting that authentication. The vulnerabilities discussed next apply to applications which rely entirely on this kind of information to identify a user session.
Suppose, for simplicity's sake, to refer to GET-accessible URLs (though the discussion applies as well to POST requests). If victim has already authenticated himself, submitting another request causes the cookie to be automatically sent with it (see picture, where the user accesses an application on www.example.com).

The GET request could be originated in several different ways:

- by the user, who is using the actual web application
- by the user, who types the URL directly in the browser
- by the user, who follows a link (external to the application) pointing to the URL

These invocations are indistinguishable by the application. In particular, the third may be quite dangerous. There are a number of techniques (and of vulnerabilities) which can disguise the real properties of a link. The link can be embedded in an email message, or appear in a malicious web site where the user is lured, i.e., the link appears in content hosted elsewhere (another web site, an HTML email message, etc.) and points to a resource of the application. If the user clicks on the link, since it was already authenticated by the web application on site, the browser will issue a GET request to the web application, accompanied by authentication information (the session id cookie). This results in a valid operation performed on the web application. By using a tag such as img, as specified in point 4 above, it is not even necessary that the user follows a particular link. Suppose the attacker sends the user an email inducing him to visit an URL referring to a page containing the following (oversimplified) HTML:

```
<html><body>
```

```
...

<img src="https://www.company.example/action" width="0" height="0">

...

</body></html>
```

What the browser will do when it displays this page is that it will try to display the specified zero-width (i.e., invisible) image as well. This results in a request being automatically sent to the web application hosted on site. It is not important that the image URL does not refer to a proper image, its presence will trigger the request specified in the src field anyway; this happens provided that image download is not disabled in the browsers, which is a typical configuration since disabling images would cripple most web applications beyond usability.
The problem here is a consequence of the following facts:

- there are HTML tags whose appearance in a page result in automatic http request execution (img being one of those);
- the browser has no way to tell that the resource referenced by img is not actually an image and is in fact not legitimate;
- image loading happens regardless of the location of the alleged image, i.e., the form and the image itself need not be located in the same host, not even in the same domain. While this is a very handy feature, it makes difficult to compartmentalize applications

It is the fact that HTML content unrelated to the web application may refer components in the application, and the fact that the browser automatically composes a valid request towards the application, that allows such kind of attacks. As no standards are defined right now, there is no way to prohibit this behavior unless it is made impossible for the attacker to specify valid application URLs. This means that valid URLs must contain information related to the user session, which is supposedly not known to the attacker and therefore make the identification of such URLs impossible.
The problem might be even worse, since in integrated mail/browser environments simply displaying an email message containing the image would result in the execution of the request to the web application with the associated browser cookie.
Things may be obfuscated further, by referencing seemingly valid image URLs such as
<img src="https://[attacker]/picture.gif" width="0" height="0">
where [attacker] is a site controlled by the attacker, and by utilizing a redirect mechanism on http://[attacker]/picture.gif to http://[thirdparty]/action.
Cookies are not the only example involved in this kind of vulnerability. Web applications whose session information is entirely supplied by the browser are vulnerable too. This includes applications relying on HTTP authentication mechanisms alone, since the authentication information is known by the browser and is sent automatically upon each request. This does not include form-based authentication, which occurs just once and generates some form of session-related information (of course, in this case, such information is expressed simply as a cookie and can we fall back to one of the previous cases).

Sample scenario.
Let's suppose that the victim is logged on to a firewall web management application. To log in, a user has to authenticate himself; subsequently, session information is stored in a cookie.
Let's suppose our firewall web management application has a function that allows an authenticated user to delete a rule specified by its positional number, or all the rules of the configuration if the user enters '*' (quite a dangerous feature, but it will make the example more interesting). The delete page is shown next. Let's suppose that the form – for the sake of simplicity – issues a GET request, which will be of the form

```
https://[target]/fwmgt/delete?rule=1 (to delete rule number one)
https://[target]/fwmgt/delete?rule=* (to delete all rules)
```

The example is purposely quite naive, but shows in a simple way the dangers of CSRF.
Session Riding Firewall Management.gif Therefore, if we enter the value '*' and press the Delete button, the following GET request is submitted.

```
https://www.company.example/fwmgt/delete?rule=*
```

with the effect of deleting all firewall rules (and ending up in a possibly inconvenient situation).

Remediation
Preventing CSRF usually requires the inclusion of an unpredictable token in each HTTP request. Such tokens should, at a minimum, be unique per user session.
The preferred option is to include the unique token in a hidden field. This causes the value to be sent in the body of the HTTP request, avoiding its inclusion in the URL, which is more prone to exposure.
The unique token can also be included in the URL itself, or a URL parameter. However, such placement runs a greater risk that the URL will be exposed to an attacker, thus compromising the secret token.
OWASP's CSRF Guard can automatically include such tokens in Java EE, .NET, or PHP apps. OWASP's ESAPI includes methods developers can use to prevent CSRF vulnerabilities.
Requiring the user to reauthenticate, or prove they are a user (e.g., via a CAPTCHA) can also protect against CSRF.

Add session-related information to the URL. What makes the attack possible is the fact that the session is uniquely identified by the cookie, which is automatically sent by the browser. Having other session-specific information being generated at the URL level makes it difficult to the attacker to know the structure of URLs to attack.
Other countermeasures, while they do not resolve the issue, contribute to make it harder to exploit.
Use POST instead of GET. While POST requests may be simulated by means of JavaScript, they make it more complex to mount an attack. The same is true with intermediate confirmation pages (such as: "Are you sure you really want to do this?" type of pages). They can be bypassed by an attacker, although they will make their work a bit more complex. Therefore, do not rely solely on these measures to protect your application. Automatic logout mechanisms somewhat mitigate the exposure to these vulnerabilities, though it ultimately depends on the context (a user who works all day long on a vulnerable web banking application is obviously more at risk than a user who uses the same application occasionally).

3.1.5.6 Token Strength

Description
The session identifier should use the largest character set available to it. If a session identifier is made up of say 8 characters of 7 bits the effective key length is 56 bits. However if the character set is made up of only integers that can be represented in 4 bits giving a key space of only 32 bits.

Remediation
A good session identifier should use as many characters as possible. Exceptions to this can be made, however, for special control characters that would require escaping and thus complicate development. Most application frameworks use the characters A-Z and 0-9 and some add case sensitivity by including a-z.

3.1.5.7 Logout functionality

Description
The end of a web session is usually triggered by one of the following two events:

- The user logs out

- The user remains idle for a certain amount of time and the application automatically logs him/her out

Both cases must be implemented carefully, in order to avoid introducing weaknesses that could be exploited by an attacker to gain unauthorized access (timeout issues are described on 3.1.5.8). More specifically, the logout function must ensure that all session tokens (e.g., cookies) are properly destroyed or made unusable, and that proper controls are enforced at the server side to prevent the reuse of session tokens.

Remediation
All pages should have an effective logout button on every single page in a common location.
Upon logout the session ID should be invalidated on the server side and deleted on the client via expiring and overwriting the value.

3.1.5.8 Session timeout

Description
Session timeout represents the event occuring when a user do not perform any action on a web site during a interval (defined by web server). The event, on server side, change the status of the user session to 'invalid' (ie. "not used anymore") and instruct the web server to destroy it (deleting all data contained into it).

Remediation
Set session timeout to the minimal value possible depending on the context of the application.
Avoid "infinite" session timeout.
Sessions should timeout after:

- 5 minutes for high-value applications
- 10 minutes for medium value application
- 20 minutes for low risk applications

On JEE web application, there 2 ways to define session timeout,

- Declaratively in web deployment descriptor (file "web.xml"): this definition is applied to all session created for the application
- Programmatically on session object: this definition apply only on current session

Prefer declarative definition of the session timeout in order to apply global timeout for all application sessions.
Trace session creation/destroy in order to analyse creation trend and try to detect anormal session number creation (application profiling phase in a attack).

3.1.5.9 Multiple concurrent sessions

Description
It is the web application design decision to determine if multiple simultaneous logons from the same user are allowed from the same or from different client IP addresses

Remediation
If the web application does not want to allow simultaneous session logons, it must take effective actions after each new authentication event, implicitly terminating the previously available session, or asking the user (through the old, new or both sessions) about the session that must remain active.
It is recommended for web applications to add user capabilities that check the details of active sessions at any time, monitor and alert the user about concurrent logons, provide user features to remotely terminate sessions manually, and track account activity history (logbook) by recording multiple client details such as IP address, User-Agent, login date and time, idle time, etc.

https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

3.1.6 Data Validation

3.1.6.1 Reflected cross site scripting

Description
Reflected XSS attacks are also known as non-persistent XSS attacks and, since the attack payload is delivered and executed via a single request and response, they are also called first-order XSS or type 1. These are the most frequent type of XSS attacks found nowadays.
When a web application is vulnerable to this type of attack, it will pass unvalidated input sent through requests to the client. The common modus operandi of the attack includes a design step, in which the attacker creates and tests an offending URI, a social engineering step, in which she convinces her victims to load this URI on their browsers, and the eventual execution of the offending code — using the victim's credentials.
Commonly the attacker's code is written in the Javascript language, but other scripting languages are also used, e.g., ActionScript and VBScript.
Attackers typically leverage these vulnerabilities to install key loggers, steal victim cookies, perform clipboard theft, and change the content of the page (e.g., download links).
One of the important matters about exploiting XSS vulnerabilities is character encoding. In some cases, the web server or the web application could not be filtering some encodings of characters, so, for example, the web application might filter out "<script>", but might not filter %3cscript%3e which simply includes another encoding of tags.

Remediation
Preventing XSS requires separation of untrusted data from active browser content.
The preferred option is to properly escape all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. See the OWASP XSS Prevention Cheat Sheet for details on the required data escaping techniques.
Positive or “whitelist” input validation is also recommended as it helps protect against XSS, but is not a complete defense as many applications require special characters in their input. Such validation should, as much as possible, validate the length, characters, format, and business rules on that data before accepting the input.
For rich content, consider auto-sanitization libraries like OWASP’s AntiSamy or the Java HTML Sanitizer Project.
Consider Content Security Policy (CSP) to defend against XSS across your entire site.

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

3.1.6.2 Stored cross site scripting

Description
Stored XSS occurs when a web application gathers input from a user which might be malicious, and then stores that input in a data store for later use. The input that is stored is not correctly filtered. As a consequence, the malicious data will appear to be part of the web site and run within the user’s browser under the privileges of the web application. Since this vulnerability typically involves at least two requests to the application, this may also called second-order XSS.
This vulnerability can be used to conduct a number of browser-based attacks including:

- Hijacking another user's browser
- Capturing sensitive information viewed by application users
- Pseudo defacement of the application
- Port scanning of internal hosts ("internal" in relation to the users of the web application)
- Directed delivery of browser-based exploits
- Other malicious activities

Stored XSS does not need a malicious link to be exploited. A successful exploitation occurs when a user visits a page with a stored XSS. The following phases relate to a typical stored XSS attack scenario:

- Attacker stores malicious code into the vulnerable page
- User authenticates in the application
- User visits vulnerable page
- Malicious code is executed by the user's browser

This type of attack can also be exploited with browser exploitation frameworks such as BeEF, XSS Proxy and Backframe. These frameworks allow for complex JavaScript exploit development.
Stored XSS is particularly dangerous in application areas where users with high privileges have access. When the administrator visits the vulnerable page, the attack is automatically executed by their browser. This might expose sensitive information such as session authorization tokens.

Remediation
Preventing XSS requires separation of untrusted data from active browser content.
The preferred option is to properly escape all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. See the OWASP XSS Prevention Cheat Sheet for details on the required data escaping techniques.
Positive or "whitelist" input validation is also recommended as it helps protect against XSS, but is not a complete defense as many applications require special characters in their input. Such validation should, as much as possible, validate the length, characters, format, and business rules on that data before accepting the input.
For rich content, consider auto-sanitization libraries like OWASP's AntiSamy or the Java HTML Sanitizer Project.
Consider Content Security Policy (CSP) to defend against XSS across your entire site.

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

3.1.6.3 Unvalidated redirects and forwards

Description
Applications frequently redirect users to other pages, or use internal forwards in a similar manner. Sometimes the target page is specified in an unvalidated parameter, allowing attackers to choose the destination page.
Such redirects may attempt to install malware or trick victims into disclosing passwords or other sensitive information. Unsafe forwards may allow access control bypass.
Attacker links to unvalidated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site. Attacker targets unsafe forward to bypass security checks.

Remediation
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
4. 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

Applications can use ESAPI to override the sendRedirect() method to make sure all redirect destinations are safe.

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

3.1.6.4 SQL injection

Description

A SQL injection attack consists of insertion or "injection" of either a partial or complete SQL query via the data input or transmitted from the client (browser) to the web application. A successful SQL injection attack can read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file existing on the DBMS file system or write files into the file system, and, in some cases, issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.
In general the way web applications construct SQL statements involving SQL syntax written by the programmers is mixed with user-supplied data. Example:

```
select title, text from news where id=$id
```

In the example above the variable $id contains user-supplied data, while the remainder is the SQL static part supplied by the programmer; making the SQL statement dynamic.
Because the way it was constructed, the user can supply crafted input trying to make the original SQL statement execute further actions of the user's choice. The example below illustrates the user-supplied data "10 or 1=1", changing the logic of the SQL statement, modifying the WHERE clause adding a condition "or 1=1".

```
select title, text from news where id=10 or 1=1
```

SQL Injection attacks can be divided into the following three classes:

- Inband: data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.
- Out-of-band: data is retrieved using a different channel (e.g., an email with the results of the query is generated and sent to the tester).
- Inferential or Blind: there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the DB Server.

A successful SQL Injection attack requires the attacker to craft a syntactically correct SQL Query. If the application returns an error message generated by an incorrect query, then it may be easier for an attacker to reconstruct the logic of the original query and, therefore, understand how to perform the injection correctly. However, if the application hides the error details, then the tester must be able to reverse engineer the logic of the original query.
About the techniques to exploit SQL injection flaws there are five commons techniques. Also those techniques sometimes can be used in a combined way (e.g. union operator and out-of-band):

- Union Operator: can be used when the SQL injection flaw happens in a SELECT statement, making it possible to combine two queries into a single result or result set
- Boolean: use Boolean condition(s) to verify whether certain conditions are true or false
- Error based: this technique forces the database to generate an error, giving the attacker or tester information upon which to refine their injection
- Out-of-band: technique used to retrieve data using a different channel (e.g., make a HTTP connection to send the results to a web server)
- Time delay: use database commands (e.g. sleep) to delay answers in conditional queries. It useful when attacker doesn't have some kind of answer (result, output, or error) from the application

Remediation
Preventing injection requires keeping untrusted data separate from commands and queries:

1) The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful with APIs, such as stored procedures, that are parameterized, but can still introduce injection under the hood.
2) If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI provides many of these escaping routines.
3) Positive or "white list" input validation is also recommended, but is not a complete defense as many applications require special characters in their input. If special characters are required, only approaches 1. and 2. above will make their use safe. OWASP's ESAPI has an extensible library of white list input validation routines.

3.1.6.5 LDAP Injection

Description
A web application could use LDAP in order to let users authenticate or search other user's information inside a corporate structure.
The goal of LDAP injection attacks is to inject LDAP search filters metacharacters in a query which will be executed by the application.
Rfc2254 defines a grammar on how to build a search filter on LDAPv3 and extends Rfc1960 (LDAPv2). An LDAP search filter is constructed in Polish notation, also known as prefix notation. This means that a pseudo code condition on a search filter like this:

```
find("cn=John & userPassword=mypass")
```

will be represented as:

```
find("(&(cn=John)(userPassword=mypass))")
```

Boolean conditions and group aggregations on an LDAP search filter could be applied by using the following metacharacters:

- & [Boolean AND]
- | [Boolean OR]
- ! [Boolean NOT]
- = [Equals]
- ~= [Approx]
- >= [Greater than]
- <= [Less than]
- * [Any character]
- () [Grouping parenthesis]

More complete examples on how to build a search filter can be found in the related RFC.
A successful exploitation of an LDAP injection vulnerability could allow the attacker to:

- Access unauthorized content
- Evade application restrictions
- Gather unauthorized informations
- Add or modify Objects inside LDAP tree structure

Remediation
Protecting LDAP-enabled web applications demands the effort of developers as well as the LDAP administrators. Though effective at reducing the risk of such an attack, the approaches discussed in the next section are not complete solutions. It is best to remember that web application security, by its own definition, must be a continually evolving process. As hackers change their methodologies, so must those who want to implement a secure Web application.

Incoming Data Validation

All client-supplied data needs to be cleaned of any characters or strings that could possibly be used maliciously. This should be done for all applications, not just those that use LDAP queries. Stripping quotes or putting backslashes in front of them is nowhere near enough. The best way to filter data is with a default-deny regular expression that includes only the type of characters that you want. For instance, the following regular expression will return only letters and numbers:

```
s/[^0-9a-zA-Z]//g
```

Make your filter as specific as possible. Whenever possible use only numbers. After that, numbers and letters only. If you need to include symbols or punctuation of any kind, make absolutely sure to convert them to HTML substitutes (such as " "e; " or " > "). For instance, if the user is submitting an email address, allow only the "at" sign, underscore, period, and hyphen in addition to numbers and letters, and only after those characters have been converted to their HTML substitutes.

Outgoing Data Validation
All data returned to the user should be validated and the amount of data returned by the queries should be restricted as an added layer of security.

LDAP Configuration
Implementing tight access control on the data in the LDAP directory is imperative, especially when configuring the permissions on user objects, and even more importantly if the directory is used for single sign-on solution. You must fully understand how each objectclass is used and decide if the user should be allowed to modify it. Allowing users to modify their uidNumber attribute, for example, may let the user change access levels when accessing systems. The access level used by the Web application to connect to the LDAP server should be restricted to the absolute minimum required. That way, even if an attacker manages to find a way to break the application, the damage would be limited. In addition, the LDAP server should not be directly accessible on the Internet, thereby eliminating direct attacks to the server itself.

3.1.6.6 Command Injection

Description
OS command injection is a technique used via a web interface in order to execute OS commands on a web server.
The user supplies operating system commands through a web interface in order to execute OS commands. Any web interface that is not properly sanitized is subject to this exploit. With the ability to execute OS commands, the user can upload malicious programs or even obtain passwords. OS command injection is preventable when security is emphasized during the design and development of applications.

Remediation
Avoid sending user controlled data to the OS as much as possible.
Ensure that a robust escaping routine is in place to prevent the user from adding additional characters that can be executed by the OS (e.g. user appends | to the malicious data and then executes another OS command). Remember to use a positive approach when constructing escaping routinges.

3.1.6.7 Unrestricted File Upload

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, forwarding attacks to backend systems, and simple defacement. It depends on what the application does with the uploaded file, including where it is stored.
There are really two different classes of problems here. The first is with the file metadata, like the path and filename. These are generally provided by the transport, such as HTTP multipart encoding. This data may trick the application into overwriting a critical file or storing the file in a bad location. You must validate the metadata extremely carefully before using it.
The other class of problem is with the file content. The range of problems here depends entirely on what the file is used for. Examples are:

- 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 trojaned executable)
- Upload virus infected file (victims' machines infected)
- Upload .html file containing script (victim experiences Cross-site Scripting)

Remediation
To protect against this type of attack, you should analyze everything your application does with files and think carefully about what processing and interpreters are involved.
Never accept a filename and its extension directly without having a white-list filter.
It is necessary to have a list of only permitted extensions on the web application. And, file extension can be selected from the list. For instance, it can be a “select case” syntax (in case of having VBScript) to choose the file extension in regard to the real file extension.
All the control characters and Unicode ones should be removed from the filenames and their extensions without any exception. Also, the special characters such as “;”, “:”, “>”, “<”, “/” ,”\”, additional “.”, “*”, “%”, “$”, and so on should be discarded as well. If it is applicable and there is no need to have Unicode characters, it is highly recommended to only accept Alpha-Numeric characters and only 1 dot as an input for the file name and the extension; in which the file name and also the extension should not be empty at all (regular expression: [a-zA-Z0-9]{1,200}\.[a-zA-Z0-9]{1,10}).
Limit the filename length. For instance, the maximum length of the name of a file plus its extension should be less than 255 characters (without any directory) in an NTFS partition.
It is recommended to use an algorithm to determine the filenames. For instance, a filename can be a MD5 hash of the name of file plus the date of the day.
Uploaded directory should not have any “execute” permission.
Limit the file size to a maximum value in order to prevent denial of service attacks (on file space or other web application’s functions such as the image resizer).
Restrict small size files as they can lead to denial of service attacks. So, the minimum size of files should be considered.
Use Cross Site Request Forgery protection methods.
Prevent from overwriting a file in case of having the same hash for both.
Use a virus scanner on the server (if it is applicable). Or, if the contents of files are not confidential, a free virus scanner website can be used. In this case, file should be stored with a random name and without any extension on the server first, and after the virus checking (uploading to a free virus scanner website and getting back the result), it can be renamed to its specific name and extension.
Try to use POST method instead of PUT (or GET)
Log users’ activities. However, the logging mechanism should be secured against log forgery and code injection itself.
In case of having compressed file extract functions, contents of the compressed file should be checked one by one as a new file.

3.1.7 Cryptography

3.1.7.1 Cache directives

Description
The TLS protocol provides confidentiality only for data in transit but it does not help with potential data leakage issues at the client or intermediary proxies. As a result, it is frequently prudent to instruct these nodes not to cache or persist sensitive data.

Remediation
One option is to add a suitable Cache-Control header to relevant HTTP responses, for example "Cache-Control: no-cache, no store, must-revalidate". For compatibility with HTTP/1.0 the response should include header "Pragma: no-cache". More information is available in HTTP 1.1 RFC 2616, section 14.9.

https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

3.1.7.2 Insecure Cryptographic storage

Description
Most web applications have a need to store sensitive information, either in a database or on a file system somewhere. The information might be passwords, credit card numbers, account records, or proprietary information. Frequently, encryption techniques are used to protect this sensitive information. While encryption has become relatively easy to implement and use, developers still frequently make mistakes while integrating it into a web application. Developers may overestimate the protection gained by using encryption and not be as careful in securing other aspects of the site.
A few areas where mistakes are commonly made include:

- Failure to encrypt critical data
- Insecure storage of keys, certificates, and passwords
- Improper storage of secrets in memory
- Poor sources of randomness
- Poor choice of algorithm
- Attempting to invent a new encryption algorithm
- Failure to include support for encryption key changes and other required maintenance procedures

The impact of these weaknesses can be devastating to the security of a website. Encryption is generally used to protect a site's most sensitive assets, which may be totally compromised by a weakness.

Remediation
The easiest way to protect against cryptographic flaws is to minimize the use of encryption and only keep information that is absolutely necessary. For example, rather than encrypting credit card numbers and storing them, simply require users to re-enter the numbers. Also, instead of storing encrypted passwords, use a one-way function, such as SHA-1, to hash the passwords.
If cryptography must be used, choose a library that has been exposed to public scrutiny and make sure that there are no open vulnerabilities. Encapsulate the cryptographic functions that are used and review the code carefully. Be sure that secrets, such as keys, certificates, and passwords, are stored securely. To make it difficult for an attacker, the master secret should be split into at least two locations and assembled at runtime. Such locations might include a configuration file, an external server, or within the code itself.

https://www.owasp.org/index.php/Insecure_Storage

3.1.7.3 Sensitive information sent via unencrypted channels

Description
Many communication channels can be "sniffed" by attackers during data transmission. For example, network traffic can often be sniffed by any attacker who has access to a network interface. This significantly lowers the difficulty of exploitation by attackers.
Though a connection is successfully made, the connection is unencrypted and it is possible that all sensitive data (SSN, credit card codes, personal datas, etc.) sent to or received from the server will be read by unintended actors.

Remediation
Sensitive information should only traverse encrypted links.
All pages which are available over TLS must not be available over a non-TLS connection. A user may inadvertently bookmark or manually type a URL to a HTTP page (e.g. http://example.com/myaccount) within the authenticated portion of the application. If this request is processed by the application then the response, and any sensitive data, would be returned to the user over the clear text HTTP.

https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

3.1.7.4 Sensitive Information sent on URL arguments

Description
Sometimes developers use URLs for pass arguments or sensitive data.

Remediation
Sensitive data must not be transmitted via URL arguments. A more appropriate place is to store sensitive data in a server side repository or within the user's session.
When using TLS the URL arguments and values are encrypted during transit. However, there are two methods that the URL arguments and values could be exposed:

- The entire URL is cached within the local user's browser history. This may expose sensitive data to any other user of the workstation
- The entire URL is exposed if the user clicks on a link to another HTTPS site. This may expose sensitive data within the referral field to the third party site. This exposure occurs in most browsers and will only occur on transitions between two TLS sites.

For example, a user following a link on https://example.com which leads to https://someOtherexample.com would expose the full URL of https://example.com (including URL arguments) in the referral header (within most browsers). This would not be the case if the user followed a link on https://example.com to http://someHTTPexample.com

https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

3.1.8 Logging and Audit

3.1.8.1 Time Synchronization

Description
The log files are useful, in particular, in the event of faults or intrusions in computer systems, either because they reveal these events and because they allow to follow the evolution of the problem so you can get a consistent temporal sequence of events.

Remediation
Log must be synchronized with an Coordinated Universal Time (UTC).

3.1.8.2 Logging

Description
Web applications should increase their logging capabilities by including information regarding the full life cycle of sessions.

Remediation
It is recommended to record session related events, such as the creation, renewal, and destruction of session IDs, as well as details about its usage within login and logout operations, privilege level changes within the session, timeout expiration, invalid session activities (when detected), and critical business operations during the session.

The log details might include a timestamp, source IP address, web target resource requested (and involved in a session operation), HTTP headers (including the User-Agent and Referer), GET and POST parameters, error codes and messages, username (or user ID), plus the session ID (cookies, URL, GET, POST…). Sensitive data like the session ID should not be included in the logs in order to protect the session logs against session ID local or remote disclosure or unauthorized access. However, some kind of session-specific information must be logged into order to correlate log entries to specific sessions. It is recommended to log a salted-hash of the session ID instead of the session ID itself in order to allow for session-specific log correlation without exposing the session ID.

In particular, web applications must thoroughly protect administrative interfaces that allow to manage all the current active sessions. Frequently these are used by support personnel to solve session related issues, or even general issues, by impersonating the user and looking at the web application as the user does.

The session logs become one of the main web application intrusion detection data sources, and can also be used by intrusion protection systems to automatically terminate sessions and/or disable user accounts when (one or many) attacks are detected. If active protections are implemented, these defensive actions must be logged too.

3.1.8.3 Audit

Description
All logins to application administration interface, successful or unsuccessful, must be logged.

Remediation
All login logs are retained for at least six months.
Audit logs are regularly reviewed by knowledgeable and independent individuals appointed by the data proprietor to meet the data proprietor's requirements. These requirements and the review process are documented.
Accounts that are locked due to maximum database login failures trigger an automatic notification of the security administrator(s) responsible for this system.

3.1.8.4 Protection of log information

Description
Logging facilities and log information should be protected against tampering and unauthorized access.

Remediation
Controls should aim to protect against unauthorized changes and operational problems with the logging facility including:

- alterations to the message types that are recorded;
- log files being edited or deleted;

- storage capacity of the log file media being exceeded, resulting in either the failure to record events or over-writing of past recorded events.

Some audit logs may be required to be archived as part of the record retention policy or because of requirements to collect and retain evidence.

Set all logs read-only and sign them with a digital signature or a hash function (e.g. MD5).

3.1.9 Client Side

3.1.9.1 DOM Based XSS

Description

DOM-based Cross-Site Scripting is the de-facto name for XSS bugs which are the result of active browser-side content on a page, typically JavaScript, obtaining user input and then doing something unsafe with it which leads to execution of injected code. This document only discusses JavaScript bugs which lead to XSS.

The DOM, or Document Object Model, is the structural format used to represent documents in a browser. The DOM enables dynamic scripts such as JavaScript to reference components of the document such as a form field or a session cookie. The DOM is also used by the browser for security (for example to limit scripts on different domains from obtaining session cookies for other domains).

A DOM-based XSS vulnerability may occur when active content, such as a JavaScript function, is modified by a specially crafted request such that a DOM element that can be controlled by an attacker.

Not all XSS bugs require the attacker to control the content returned from the server, but can instead abuse poor JavaScript coding practices to achieve the same results. The consequences are the same as a typical XSS flaw, only the means of delivery is different.

In comparison to other cross site scripting vulnerabilities (reflected and stored XSS), where an unsanitized parameter is passed by the server, returned to the user and executed in the context of the user's browser, a DOM-based XSS vulnerability controls the flow of the code by using elements of the Document Object Model (DOM) along with code crafted by the attacker to change the flow.

Due to their nature, DOM-based XSS vulnerabilities can be executed in many instances without the server being able to determine what is actually being executed. This may make many of the general XSS filtering and detection techniques impotent to such attacks.

The first hypothetical example uses the following client side code:

```
<script>
document.write("Site is at: " + document.location.href + ".");
</script>
```

An attacker may append #<script>alert('xss')</script> to the affected page URL which would, when executed, display the alert box. In this instance, the appended code would not be sent to the server as everything after the # character is not treated as part of the query by the browser but as a fragment. In this example, the code is immediately executed and an alert of "xss" is displayed by the page. Unlike the more common types of cross site scripting (Stored and Reflected) in which the code is sent to the server and then back to the browser, this is executed directly in the user's browser without server contact.

The consequences of DOM-based XSS flaws are as wide ranging as those seen in more well known forms of XSS, including cookie retrieval, further malicious script injection, etc. and should therefore be treated with the same severity.

JavaScript applications differ significantly from other types of applications because they are often dynamically generated by the server, and to understand what code is being executed, the website being tested needs to be crawled to determine all the instances of JavaScript being executed and where user input is accepted. Many websites rely on large libraries of functions, which often stretch into the hundreds of thousands of lines of code and have not been

developed in-house. In these cases, top-down testing often becomes the only really viable option, since many bottom level functions are never used, and analyzing them to determine which are sinks will use up more time than is often available. The same can also be said for top-down testing if the inputs or lack thereof is not identified to begin with.
User input comes in two main forms:

- Input written to the page by the server in a way that does not allow direct XSS
- Input obtained from client-side JavaScript objects

Here are two examples of how the server may insert data into JavaScript:

```
var data = "<escaped data from the server>";
var result = someFunction("<escaped data from the server>");
```

And here are two examples of input from client-side JavaScript objects:

```
var data = window.location;
var result = someFunction(window.referer);
```

While there is little difference to the JavaScript code in how they are retrieved, it is important to note that when input is received via the server, the server can apply any permutations to the data that it desires, whereas the permutations performed by JavaScript objects are fairly well understood and documented, and so if someFunction in the above example were a sink, then the exploitability of the former would depend on the filtering done by the server, whereas the latter would depend on the encoding done by the browser on the window.referer object. Stefano Di Paulo has written an excellent article on what browsers return when asked for the various elements of a URL using the document. and location. attributes.
Additionally, JavaScript is often executed outside of <script> blocks, as evidenced by the many vectors which have led to XSS filter bypasses in the past, and so, when crawling the application, it is important to note the use of scripts in places such as event handlers and CSS blocks with expression attributes. Also, note that any off-site CSS or script objects will need to be assessed to determine what code is being executed.
Automated testing has only very limited success at identifying and validating DOM-based XSS as it usually identifies XSS by sending a specific payload and attempts to observe it in the server response. This may work fine for the simple example provided below, where the message parameter is reflected back to the user:

```
<script>
var pos=document.URL.indexOf("message=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

but may not be detected in the following contrived case:

```
<script>
var navAgt = navigator.userAgent;

if (navAgt.indexOf("MSIE")!=-1) {
     document.write("You are using IE as a browser and visiting site: " +
document.location.href + ".");
}
else
{
    document.write("You are using an unknown browser.");
}
</script>
```

For this reason, automated testing will not detect areas that may be susceptible to DOM-based XSS unless the testing tool can perform addition analysis of the client side code.
Manual testing should therefore be undertaken and can be done by examining areas in the code where parameters are referred to that may be useful to an attacker. Examples of such

areas include places where code is dynamically written to the page and elsewhere where the DOM is modified or even where scripts are directly executed. Further examples are described in the excellent DOM XSS article by Amit Klein, referenced at the end of this section.

Remediation

For prevent to DOM Based XSS there are several rules:

1) HTML Escape then JavaScript Escape Before Inserting Untrusted Data into HTML Subcontext within the Execution Context. There are several methods and attributes which can be used to directly render HTML content within JavaScript. These methods constitute the HTML Subcontext within the Execution Context. If these methods are provided with untrusted input, then an XSS vulnerability could result. Examples of dangerous HTML Methods are:
   - Attributes

```
element.innerHTML = "<HTML> Tags and markup";
element.outerHTML = "<HTML> Tags and markup";
```

   - Methods

```
document.write("<HTML> Tags and markup");
document.writeln("<HTML> Tags and markup");
```

   - Guideline. To make dynamic updates to HTML in the DOM safe, we recommend a) HTML encoding, and then b) JavaScript encoding all untrusted input, as shown in these examples:

```
element.innerHTML =
"<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrustedData))%>";
element.outerHTML =
"<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrustedData))%>";

document.write("<%=Encoder.encodeForJS(Encoder.encodeForHTML(untruste
dData))%>");
document.writeln("<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrus
tedData))%>");
```

   Note: The Encoder.encodeForHTML() and Encoder.encodeForJS() are just notional encoders. Various options for actual encoders are listed later in this document.

2) JavaScript Escape Before Inserting Untrusted Data into HTML Attribute Subcontext within the Execution Context. The HTML attribute *subcontext* within the *execution* context is divergent from the standard encoding rules. This is because the rule to HTML attribute encode in an HTML attribute rendering context is necessary in order to mitigate attacks which try to exit out of an HTML attributes or try to add additional attributes which could lead to XSS. When you are in a DOM execution context you only need to JavaScript encode HTML attributes which do not execute code (attributes other than event handler, CSS, and URL attributes).
   For example, the general rule is to HTML Attribute encode untrusted data (data from the database, HTTP request, user, back-end system, etc.) placed in an HTML Attribute. This is the appropriate step to take when outputting data in a rendering context, however using HTML Attribute encoding in an execution context will break the application display of data.
3) Be Careful when Inserting Untrusted Data into the Event Handler and JavaScript code Subcontexts within an Execution Context. Putting dynamic data within JavaScript code is especially dangerous because JavaScript encoding has different semantics for JavaScript encoded data when compared to other encodings. In many cases, JavaScript encoding does not stop attacks within an execution context. For example, a JavaScript encoded string will execute even though it is JavaScript encoded.

4) JavaScript Escape Before Inserting Untrusted Data into the CSS Attribute Subcontext within the Execution Context. Normally executing JavaScript from a CSS context required either passing javascript:attackCode() to the CSS url() method or invoking the CSS expression() method passing JavaScript code to be directly executed. From my experience, calling the expression() function from an execution context (JavaScript) has been disabled. In order to mitigate against the CSS url() method, ensure that you are URL encoding the data passed to the CSS url() method.

```
 document.body.style.backgroundImage                                    =
"url(<%=Encoder.encodeForJS(Encoder.encodeForURL(companyName))%>)";
```

5) URL Escape then JavaScript Escape Before Inserting Untrusted Data into URL Attribute Subcontext within the Execution Context. The logic which parses URLs in both execution and rendering contexts looks to be the same. Therefore there is little change in the encoding rules for URL attributes in an execution (DOM) context.

```
var x = document.createElement("a");
x.setAttribute("href",
'<%=Encoder.encodeForJS(Encoder.encodeForURL(userRelativePath))%>');
var y = document.createTextElement("Click Me To Test");
x.appendChild(y);
document.body.appendChild(x);
```

If you utilize fully qualified URLs then this will break the links as the colon in the protocol identifier ("http:" or "javascript:") will be URL encoded preventing the "http" and "javascript" protocols from being invoked.

3.1.9.2 Clickjacking

Description

"Clickjacking" (that is a subset of the "UI redressing") is a malicious technique that consists of deceiving a web user into interact (in most cases by clicking) on something different to what the user believes he is interacting on. This type of attack, that can be used alone or in combination with other attacks, could potentially send unauthorized commands or reveal confidential information while the victim is interacting on seemingly harmless web pages. The term "Clickjacking" was coined by Jeremiah Grossman and Robert Hansen in 2008.

A Clickjacking attack uses seemingly innocuous features of HTML and Javascript to force the victim to perform undesired actions, such as clicking on a button that appears to perform another operation. This is a "client side" security issue that affects a variety of browsers and platforms. To carry out this type of technique the attacker has to create a seemingly harmless web page that loads the target application throught the use of an iframe (suitably concealed through the use of CSS code). Once this is done, the attacker could induce the victim to interact with his fictitious web page by other means (like for example social engineering). Like others attacks, an usual prerequisite is that the victim is authenticated against the attacker's target website.

The power of this method is due to the fact that the actions performed by the victim are originated from the authentic target web page (hidden but authentic). Consequently some of the anti-CSRF protections, that are deployed by the developers to protect the web page from CSRF attacks, could be bypassed.

Remediation

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.

There are three possible values for the X-Frame-Options headers:

- DENY, which prevents any domain from framing the content
- SAMEORIGIN, which only allows the current site to frame the content

- ALLOW-FROM uri, which permits the specified 'uri' to frame this page. (e.g., ALLOW-FROM http://www.example.com) The ALLOW-FROM option is a relatively recent addition (2012) and may not be supported by all browsers yet

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.

One way to defend against clickjacking is to include a "frame-breaker" script in each page that should not be framed. The following methodology will prevent a webpage from being framed even in legacy browsers, that do not support the X-Frame-Options-Header.
In the document HEAD element:

1. apply an ID to the style element itself:

```
<style id="antiClickjack">body{display:none !important;}</style>
```

2. And then delete that style by its ID immediately after in the script:

```
<script type="text/javascript">
   if (self === top) {
       var antiClickjack = document.getElementById("antiClickjack");
       antiClickjack.parentNode.removeChild(antiClickjack);
   } else {
       top.location = self.location;
   }
</script>
```

This way, everything can be in the document HEAD and you only need one method/taglib in your API.
The use of x-frame-options or a frame-breaking script is a more fail-safe method of clickjacking protection. However, in scenarios where content must be frameable, then a window.confirm() can be used to help mitigate Clickjacking by informing the user of the action they are about to perform.

Invoking window.confirm() will display a popup that cannot be framed. If the window.confirm() originates from within an iframe with a different domain than the parent, then the dialog box will display what domain the window.confirm() originated from. In this scenario the browser is displaying the origin of the dialog box to help mitigate Clickjacking attacks. It should be noted that Internet Explorer is the only known browser that does not display the domain that the window.confirm() dialog box originated from, to address this issue with Internet Explorer insure that the message within the dialog box contains contextual information about the type of action being performed. For example:

```
<script type="text/javascript">
   var action_confirm = window.confirm("Are you sure you want to delete your
youtube account?")
   if (action_confirm) {
       //... perform action
   } else {
       //...  The user does not want to perform the requested action.
   }
</script>
```

3.1.10 Web Services

3.1.10.1 Transport Confidenciality

Description
Transport confidentiality protects against eavesdropping and man-in-the-middle attacks against web service communications to/from the server.

Remediation
All communication with and between web services containing sensitive features, an authenticated session, or transfer of sensitive data must be encrypted using well configured TLS. This is recommended even if the messages themselves are encrypted because SSL/TLS provides numerous benefits beyond traffic confidentiality including integrity protection, replay defenses, and server authentication. For more information on how to do this properly see 3.4.3.

3.1.10.2 Server Authentication

Description
Server Authentication provides to the client a proof of server's identity authenticity.

Remediation
SSL/TLS must be used to authenticate the service provider to the service consumer. The service consumer should verify the server certificate is issued by a trusted provider, is not expired, is not revoked, matches the domain name of the service, and that the server has proven that it has the private key associated with the public key certificate (by properly signing something or successfully decrypting something encrypted with the associated public key).

3.1.10.3 User Authentication

Description
User authentication verifies the identity of the user or the system trying to connect to the service. Such authentication is usually a function of the container of the web service.

Remediation
If used, Basic Authentication must be conducted over SSL, but Basic Authentication is not recommended.
Client Certificate Authentication using SSL is a strong form of authentication that is recommended.

3.1.10.4 Transport Encoding

Description
SOAP encoding styles are meant to move data between software objects into XML format and back again.

Remediation
Enforce the same encoding style between the client and the server.

3.1.10.5 Message Integrity

Description
Integrity of data in transit can easily be provided by SSL/TLS.
When using public key cryptography, encryption does guarantee confidentiality but it does not guarantee integrity since the receiver's public key is public. For the same reason, encryption does not ensure the identity of the sender.

Remediation
For XML data, use XML digital signatures to provide message integrity using the sender's private key. This signature can be validated by the recipient using the sender's digital certificate (public key).

3.1.10.6 Message Confidenciality

Description
Data elements meant to be kept confidential must be encrypted using a strong encryption cipher with an adequate key length to deter brute forcing.

Remediation
Messages containing sensitive data must be encrypted using a strong encryption cipher. This could be transport encryption or message encryption.
Messages containing sensitive data that must remain encrypted at rest after receipt must be encrypted with strong data encryption, not just transport encryption.

3.1.10.7 Authorization

Description
Web services need to authorize web service clients the same way web applications authorize users. A web service needs to make sure a web service client is authorized to: perform a certain action (coarse-grained); on the requested data (fine-grained).

Remediation
A web service should authorize its clients whether they have access to the method in question. Following authentication, the web service should check the privileges of the requesting entity whether they have access to the requested resource. This should be done on every request.
Ensure access to administration and management functions within the Web Service Application is limited to web service administrators. Ideally, any administrative capabilities would be in an application that is completely separate from the web services being managed by these capabilities, thus completely separating normal users from these sensitive functions.

3.1.10.8 Schema Validation

Description
Schema validation enforces constraints and syntax defined by the schema.

Remediation
Web services must validate SOAP payloads against their associated XML schema definition (XSD).
The XSD defined for a SOAP web service should, at a minimum, define the maximum length and character set of every parameter allowed to pass into and out of the web service.
The XSD defined for a SOAP web service should define strong (ideally white list) validation patterns for all fixed format parameters (e.g., zip codes, phone numbers, list values, etc.).

3.1.10.9 Content Validation

Description
Like any web application, web services need to validate input before consuming it.

Remediation
Content validation for XML input should include:
- Validation against malformed XML entities
- Validation against XML Bomb attacks
- Validating inputs using a strong white list

- Validating against external entity attacks

3.1.10.10 Output Encoding

Description
Web services need to ensure that output sent to clients is encoded to be consumed as data and not as scripts. This gets pretty important when web service clients use the output to render HTML pages either directly or indirectly using AJAX objects

Remediation
All the rules of output encoding applies as per XSS (Cross Site Scripting) Prevention Cheat Sheet
[https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet]

3.1.10.11 Virus Protection

Description
SOAP provides the ability to attach files and document to SOAP messages. This gives the opportunity for hackers to attach viruses and malware to these SOAP messages.

Remediation
Ensure Virus Scanning technology is installed and preferably inline so files and attachments could be checked before being saved on disk.
Ensure Virus Scanning technology is regularly updated with the latest virus definitions / rules.

3.1.10.12 Message Size

Description
Web services like web applications could be a target for DOS attacks by automatically sending the web services thousands of large size SOAP messages. This either cripples the application making it unable to respond to legitimate messages or it could take it down entirely.

Remediation
SOAP Messages size should be limited to an appropriate size limit. Larger size limit (or no limit at all) increases the chances of a successful DoS attack.

3.1.10.13 Message Throughput

Description
Throughput represents the number of web service requests served during a specific amount of time.

Remediation
Configuration should be optimized for maximum message throughput to avoid running into DoS-like situations.

3.1.10.14 XML Denial Of Service

Description
XML Denial of Service is probably the most serious attack against web services.

Remediation
The web service must provide the following validation:
- validation against recursive payloads
- Validation against oversized payloads
- Protection against XML entity expansion

- Validating against overlong element names. If you are working with SOAP-based Web Services, the element names are those SOAP Actions

This protection should be provided by your XML parser/schema validator. To verify, build test cases to make sure your parser to resistant to these types of attacks.

3.2 Network Controls

3.2.1 Configuration & Deploy

3.2.1.1 Network Configuration

Description
Network deployment must take security in mind with strong segregation particulary for core and exposed services.

Remediation
The network must not be flat but make use of VLAN technology and inter-vlan routing.
Servers must be placed in a different VLAN divided by server functions. The communication between VLANs must be possible only trough a network firewall. At least one dedicated VLAN must be created for Web server another one for application sever and another for database server.
If the use of a virtualization platform is in place, the VLAN segregation can be done at Hypervisor level using proprietary tools of the specific solution (I.E VMware virtual switches and port groups). If the total network troughput of the virtualized machines equals the firewall troughput, according to specifications/datasheets, the use of a virtual firewall appliance must be taken in consideration to offload the physical firewall and prevent bandwidth hogs.
The workstation machines, must be placed in separate VLANs matching the employees different departments and geographical location. Network firewall must regulate the communication of this VLANs with servers VLANS and internet. On the firewall policy the workstations must communicate with servers only on the ports needed to access business services on the respective VLANs. The communication of the workstations to internet must be allowed only to specific ports (usually 80 HTTP amd 443 HTTPS) and must employ URL filtering technology that can be done on the firewall itself or on a dedicated proxy server.

3.2.1.2 Firewall for Web Server and Database

Description
Web Servers and Databases must be placed properly for ensure an high security.

Remediation
The database server is located behind a firewall with default rules to deny all traffic.
The database server firewall is opened only to specific application or web servers, and firewall rules do not allow direct client access.
If the development environment cannot meet this requirement, then restricted data is not stored in the development database server and mock data is made up for development. Data obfuscation of production data is not sufficient.
Firewall rule change control procedures are in place and notification of rule changes are distributed to System Administrators (SAs) and Database Administrators (DBAs).
Firewall rules for database servers are maintained and reviewed on a regular basis by SAs and DBAs. If using the IST provided firewall service, the rules are also regularly reviewed by SNS.
Regularly test machine hardening and firewall rules via network scans, or by allowing SNS scans through the firewall.

3.3 Operative System Controls

3.3.1 Configuration & Deploy

3.3.1.1 OS Hardening

Description
It is important to configure properly the server OS for guarantee an acceptable security level.

Remediation
NSA has developed and distributed configuration guidance for operating systems (http://www.nsa.gov/ia/mitigation_guidance/security_configuration_guides/operating_systems.shtml). These guides are currently being used throughout the government and by numerous entities as a security baseline for their systems.
Covered OS are:

- Apple iOS 5 Devices
- Apple Mac OS X 10.6 'Snow Leopard'
- Red Hat Enterprise Linux 5
- Microsoft Windows Applications
- Microsoft Windows 7
- Microsoft Windows Vista
- Microsoft Windows Server 2008/ Windows Server 2008 R2
- Microsoft Windows Server 2003
- Microsoft Windows XP
- Sun Solaris 10
- Sun Solaris 9

Static content generated by web application (for example pdf created in response to a particular request) shall be encrypted from the operating systems.

3.3.1.2 Patching and updates

Description
Computers are typically attacked within seconds of being connected to the U of A network – systems that are not "patched" to protect against these attacks will be compromised immediately. This is very common with brand new or reinstalled systems, which are often connected to the network "just to patch software", and are compromised during this process.
All software requires regular maintenance to remain secure; this maintenance comes in the form of patches or updates. As new ways to exploit weaknesses are discovered, software vendors release patches to correct problems in the software. Patches can usually be downloaded from the vendor's homepage, while some vendors provide a means to automatically download and install them. It is important that patches be installed as soon as possible after they are released; attackers often develop exploits shortly after the vulnerabilities are discovered.

Remediation
Follow these guideline for maintains patched and updated OS.
For Microsoft Windows patch updates

- Use Automatic Updates
- Check Windows Update: http://windowsupdate.microsoft.com
- Check Office Updates: http://office.microsoft.com/en-us/officeupdate/default.aspx
- Subscribe to Microsoft Security Notification Service

For Apple Macintosh patch updates:

- Use Automatic Updates
- Subscribe to Apple Security Notification Service

For Linux OS patch updates subscribe to mailing list for your distribution.

For Unix OS patch updates
- Check vendor's website for available options
- Subscribe to applicable mailing list(s)

http://security.arizona.edu/isg602b

3.3.2 Logging and Audit

3.3.2.1 **System administrators logs**

Description
System administrators activities should be logged.

Remediation
Logs should include:
- the time at which an event (log-in; log-out; log-failed) occurred;
- information about the event (e.g. files handled) or failure (e.g. error occurred and corrective action taken);
- which account and which administrator was involved;
- which processes were involved;
- All login logs are retained for at least six months.

System administrators logs should be reviewed on a regular basis.

3.3.2.2 Protection of log information

Description
Logging facilities and log information should be protected against tampering and unauthorized access.

Remediation
Controls should aim to protect against unauthorized changes and operational problems with the logging facility including:
- alterations to the message types that are recorded;
- log files being edited or deleted;
- storage capacity of the log file media being exceeded, resulting in either the failure to record events or over-writing of past recorded events.

Set all logs read-only and sign them with a digital signature or a hash function (e.g. MD5).

3.4 Database and Middleware Controls

3.4.1 Configuration & Deploy

3.4.1.1 Server privileges

Description
Far too often, web and application servers run at too great a permission level. They execute using privileged accounts such as root in UNIX environment or LOCALSYSTEM in Windows environments. When web and application servers run as root or LOCALSYSTEM, the processes and the code on top of these processes run with all of the rights of these users. Malicious code will execute with the authority of the privileged account, thus increasing the possible damage from an exploit. Web and application servers should be executed under accounts with minimal permissions.

The database accounts used by web applications often have privileges beyond those actually required or advisable. Allowing web applications to use "sa" or other privileged database accounts destroys the database server's ability to defend against access to or modification of unauthorized resources. Accounts with db_owner equivalent privileges such as schema modification or unlimited data access typically have far more access to the database than is required to implement application functionality. Web applications should use one or more lesser-privileged accounts that are prevented from making schema changes or sweeping changes to or requests for data.

Remediation
Development, test, and staging environments must be set up to function with the lowest possible privilege so that production will also work with lowest possible privileges.
Ensure that system level accounts (those that run the environment) have privileges limited to the greatest degree possible. Web and application servers should never execute as "Administrator", "root", "sa", "sysman", "Supervisor", or any other privileged account.
User accounts should possess just enough privileges within the application to perform their assigned tasks. These user accounts should be created unprivileged and be given permissions incrementally until they have the full capabilities required. They should not be created with high privileges and then arbitrarily limited.
Business user accounts should not be administrator accounts and vice versa. Separate accounts should be used to perform these different sets of tasks even if the same user needs to be able to perform tasks in both realms.
Web applications should access the database through one or more limited accounts that do not have schema-modification privileges unless required. If the web application requires the ability to modify the database schema, then the design should be analyzed to determine if and why functionality must be implemented in such a potentially hazardous manner.
Database access should be performed through parameterized stored procedures (or similar) to allow all table access to be revoked (i.e. select, drop, update, insert, etc). This should be done using a low privilege database account. This account should not hold any SQL roles above "user" (or similar).
Code access security should be evaluated and implemented if possible. If a component only needs the ability to perform DNS queries, it should only be granted the code access permissions to permit this. That way if the code attempts to read from the file system or make arbitrary network connections, this will not be allowed and an error will be raised.

3.4.1.2 Default credentials

Description
Nowadays web applications often make use of popular open source or commercial software that can be installed on servers with minimal configuration or customization by the server administrator. Moreover, a lot of hardware appliances (i.e. network routers and database servers), offer web-based configuration or administrative interfaces.
Often these applications, once installed, are not properly configured and the default credentials provided for initial authentication and configuration are never changed.
These default credentials are well known by penetration testers and, unfortunately, also by malicious attackers, who can use them to gain access to various types of applications.
Furthermore, in many situations, when a new account is created on an application, a default password (with some standard characteristics) is generated. If this password is predictable and the user does not change it on the first access, this can lead an attacker to gain unauthorized access to the application.

The root cause of this problem can be identified as:

- Inexperienced IT personnel, who are unaware of the importance of changing default passwords on installed infrastructure components;
- Programmers who leave backdoors to easily access and test their application and later forget to remove them.

- Applications with built-in, non-removable default accounts with a pre-set username and password.
- Applications that do not force the user to change the default credential after the first login operation.

Remediation
Change all default credentials and use a strong password policy (8 characters minimum length contains uppercase, lowercase, numbers and special characters).

3.4.1.3 Patching and updates

Description
Computers are typically attacked within seconds of being connected to the U of A network – systems that are not "patched" to protect against these attacks will be compromised immediately. This is very common with brand new or reinstalled systems, which are often connected to the network "just to patch software", and are compromised during this process.
All software requires regular maintenance to remain secure; this maintenance comes in the form of patches or updates. As new ways to exploit weaknesses are discovered, software vendors release patches to correct problems in the software. Patches can usually be downloaded from the vendor's homepage, while some vendors provide a means to automatically download and install them. It is important that patches be installed as soon as possible after they are released; attackers often develop exploits shortly after the vulnerabilities are discovered.

Remediation
The database software version is currently supported by the vendor or open source project, as required by the campus minimum security standards.
Database software is patched to include all current security patches. Provisions are made to maintain security patch levels in a timely fashion.

Follow these guideline for maintains patched and updated middleware:
- Visit vendor's website regularly to check for updates
- Register to receive email notification of updates

http://security.arizona.edu/isg602b

3.4.1.4 Hardening

Description
It is important to comprehend any parts of the security process about back-end components that directly communicate with the web applications as well as databases, wen server, etc.

Remediation
For web server follows the guidelines provided by NIST (http://csrc.nist.gov/publications/nistpubs/800-44-ver2/SP800-44v2.pdf):
- Install the Web server software on a dedicated host or a dedicated virtualized guest OS Apply any patches or upgrades to correct for known vulnerabilities
- Create a dedicated physical disk or logical partition (separate from OS and Web server application) for Web content
- Remove or disable all services installed by the Web server application but not required (e.g., gopher, FTP, remote administration)
- Remove or disable all unneeded default login accounts created by the Web server installation
- Remove all manufacturer documentation from server
- Remove any example or test files from server, including scripts and executable code
- Apply appropriate security template or hardening script to the server

- Reconfigure HTTP service banner (and others as required) NOT to report Web server and OS type and version
- Configure OS and Web server access controls
- Configure the Web server process to run as a user with a strictly limited set of privileges
- Configure the Web server so that Web content files can be read but not written by service processes
- Configure the Web server so that service processes cannot write to the directories where public Web content is stored
- Configure the Web server so that only processes authorized for Web server administration can write Web content files
- Configure the host OS so that the Web server can write log files but not read them
- Configure the host OS so that temporary files created by the Web server application are restricted to a specified and appropriately protected subdirectory
- Configure the host OS so that access to any temporary files created by the Web server application is limited to the service processes that created the files
- Install Web content on a different hard drive or logical partition than the OS and Web server application
- If uploads are allowed to the Web server, configure it so that a limit is placed on the amount of hard drive space that is dedicated for this purpose; uploads should be placed on a separate partition
- Ensure that log files are stored in a location that is sized appropriately; log files should be placed on a separate partition
- Configure the maximum number of Web server processes and/or network connections that the Web server should allow
- Ensure that any virtualized guest OSs follow this checklist
- Ensure users and administrators are able to change passwords
- Disable users after a specified period of inactivity
- Ensure each user and administrator has a unique ID
- Configure a secure Web content directory
- Dedicate a single hard drive or logical partition for Web content and establish related subdirectories exclusively for Web server content files, including graphics but excluding scripts and other programs
- Define a single directory exclusively for all external scripts or programs executed as part of Web server content (e.g., CGI, ASP)
- Disable the execution of scripts that are not exclusively under the control of administrative accounts. This action is accomplished by creating and controlling access to a separate directory intended to contain authorized scripts
  Disable the use of hard or symbolic links (e.g., shortcuts for Windows)
- Define a complete Web content access matrix. Identify which folders and files within the Web server document should be restricted and which should be accessible (and by whom)
- Check the organization's password policy and set account passwords appropriately (e.g., length, complexity)
- Use the robots.txt file, if appropriate
- Configure anti-spambot protection, if appropriate (e.g., CAPTCHAs, nofollow, or keyword filtering)

For databases use the following guidelines (https://security.berkeley.edu/node/138):

- DBAs understand their responsibility for reviewing all requested script and database changes to ensure the security of the system is not compromised
- Accounts with system administration capabilities are provided to as few individuals as is practical, and only as needed to support the application
- All Developers, Vendors, SAs, DBAs & Contractors have signed a non-disclosure agreement

- All developers, SAs, DBAs and contractors have passed a criminal background check if required by the background check policy. The background check policy may be found at http://campuspol.chance.berkeley.edu/Policies/BackgroundChecks.htm
- Operating system accounts used by DBA staff to login to dataserver machines for administrative duties are individual accounts, and not a shared group account.
- When possible, the daemon OS account that is required to run the dataserver process does not allow a direct login
- Instead, individual OS accounts are used to login, then sudo or su to the daemon account (for UNIX) or disallow desktop login (Windows)
- Database accounts used by DBA staff for administrative duties are individual accounts, and not a shared group account
- A group account is permitted for running automated DBA maintenance and monitoring jobs, such as backups
- This group account is not used for daily interactive tasks by the DBA group, except when required to troubleshoot maintenance and monitoring jobs
- Passwords for all DBA operating system accounts and database accounts are strong passwords, and are changed when administrators/contractors leave positions. See: Password complexity guidelines
- If the DBA and developer roles are being filled by a single person, changes are approved by the Data Proprietor
- Secure authentication to the database is used
- The procedure for provisioning and reviewing access to the database is documented. The data proprietor has signed the procedures document
- Only authorized users have access to the database
- Users are granted the minimal permissions necessary for their job function in the database. Permissions are managed through roles or groups, and not by direct grants to user IDs where possible
- Strong passwords in the database are enforced when technically possible, and database passwords are encrypted when stored in the database or transmitted over the network.
- Data input required by the authentication form must be stored in separate tables from those containing passwords
- Applications require individual database login/password and roles/grants when possible. When not possible, application accounts may be utilized. However, the login ID and password must be secured in this case, and this information does not exist on the client workstation
- Applications should manage user permissions and auditing to meet the Data Proprietors requirements
- User database objects with restricted data do not have public grants when possible. Document any public grants if needed in databases with restricted data
- Non-DBA accounts do not allow the granting of roles or permissions in any environment with restricted data (QA, Production, Dev)
- Database accounts are locked after at most six failed logins
- Procedure to address inactive users are documented and approved by the Data Proprietor
- A report of elevated database permissions is provided to the data proprietor by the DBAs on a quarterly basis
- A report of all access rights for users is provided to the data proprietor by the DBAs on a regular basis. Twice a year is the recommended interval

Furthermore follow the OWASP hardening guidelines:

- Oracle (https://www.owasp.org/index.php/OWASP_Backend_Security_Project_Oracle_Hardening)
- MS SQL Server (https://www.owasp.org/index.php/OWASP_Backend_Security_Project_SQLServer_Hardening)

- IBM DB2 (https://www.owasp.org/index.php/OWASP_Backend_Security_Project_DB2_Hardening)
- MySQL (https://www.owasp.org/index.php/OWASP_Backend_Security_Project_MySQL_Hardening)
- PostgreSQL (https://www.owasp.org/index.php/OWASP_Backend_Security_Project_PostgreSQL_Hardening)

https://www.owasp.org/index.php/OWASP_Backend_Security_Project

3.4.1.5 Sensitive Information on file extension and Old, Backup and Unreferenced Files

Description

Old, backup and unreferenced files present various threats to the security of a web application:

- Unreferenced files may disclose sensitive information that can facilitate a focused attack against the application; for example include files containing database credentials, configuration files containing references to other hidden content, absolute file paths, etc.
- Unreferenced pages may contain powerful functionality that can be used to attack the application; for example an administration page that is not linked from published content but can be accessed by any user who knows where to find it
- Old and backup files may contain vulnerabilities that have been fixed in more recent versions; for example *viewdoc.old.jsp*may contain a directory traversal vulnerability that has been fixed in *viewdoc.jsp* but can still be exploited by anyone who finds the old version
- Backup files may disclose the source code for pages designed to execute on the server; for example requesting *viewdoc.bak* may return the source code for *viewdoc.jsp*, which can be reviewed for vulnerabilities that may be difficult to find by making blind requests to the executable page. While this threat obviously applies to scripted languages, such as Perl, PHP, ASP, shell scripts, JSP, etc., it is not limited to them, as shown in the example provided in the next bullet
- Backup archives may contain copies of all files within (or even outside) the webroot. This allows an attacker to quickly enumerate the entire application, including unreferenced pages, source code, include files, etc. For example, if you forget a file named *myservlets.jar.old* file containing (a backup copy of) your servlet implementation classes, you are exposing a lot of sensitive information which is susceptible to decompilation and reverse engineering
- In some cases copying or editing a file does not modify the file extension, but modifies the filename. This happens for example in Windows environments, where file copying operations generate filenames prefixed with "Copy of " or localized versions of this string. Since the file extension is left unchanged, this is not a case where an executable file is returned as plain text by the web server, and therefore not a case of source code disclosure. However, these files too are dangerous because there is a chance that they include obsolete and incorrect logic that, when invoked, could trigger application errors, which might yield valuable information to an attacker, if diagnostic message display is enabled
- Log files may contain sensitive information about the activities of application users, for example sensitive data passed in URL parameters, session IDs, URLs visited (which may disclose additional unreferenced content), etc. Other log files (e.g. ftp logs) may contain sensitive information about the maintenance of the application by system administrators
- Filesystem snapshots may contain copies of your code that contain vulnerabilities that have been fixed in more recent versions, for example */.snapshot/monthly.1/view.php* may contain a directory traversal vulnerability

that has been fixed in*/view.php* but can still be exploited by anyone who finds the old version

Remediation
To guarantee an effective protection strategy it is necessary a security policy which clearly forbids dangerous practices, such as:

- Editing files in-place on the web server / application server filesystems. This is a particular bad habit, since it is likely to unwillingly generate backup files by the editors. It is amazing to see how often this is done, even in large organizations. If you absolutely need to edit files on a production system, do ensure that you don't leave behind anything which is not explicitly intended, and consider that you are doing it at your own risk
- Check carefully any other activity performed on filesystems exposed by the web server, such as spot administration activities. For example, if you occasionally need to take a snapshot of a couple of directories (which you shouldn't, on a production system...), you may be tempted to zip/tar them first. Be careful not to forget behind those archive files
- Appropriate configuration management policies should help not to leave around obsolete and unreferenced files
- Applications should be designed not to create (or rely on) files stored under the web directory trees served by the web server. Data files, log files, configuration files, etc. should be stored in directories not accessible by the web server, to counter the possibility of information disclosure (not to mention data modification if web directory permissions allow writing...)
- Filesystem snapshots should not be accessible via the web if your document root is on a filesystem using this technology. Configure your web server to deny access to such directories, for example under apache a location directive such this should be used:

```
<Location ~ ".snapshot">
    Order deny,allow
    Deny from all
</Location>
```

A good strategy is that of periodically scheduling a background job checking for files with extensions likely to identify them as copy/backup files, and performing manual checks as well on a longer time basis.

3.4.1.6 Infrastructure admin interfaces

Description
Databases and middleware may require an administrator interface to enable a privileged user to access administrative functionality.

Remediation
Enable external administrative interface access only to authorized IPs or use authorized internal accounts through VPN connection.

3.4.1.7 Insecure HTTP Methods

Description
While GET and POST are by far the most common methods that are used to access information provided by a web server, the Hypertext Transfer Protocol (HTTP) allows several other (and somewhat less known) methods. RFC 2616 (which describes HTTP version 1.1 which is the today standard) defines the following eight methods:

- HEAD
- GET
- POST
- PUT

- DELETE
- TRACE
- OPTIONS
- CONNECT

Some of these methods can potentially pose a security risk for a web application, as they allow an attacker to modify the files stored on the web server and, in some scenarios, steal the credentials of legitimate users

Remediation
The HTTP Methods that should be disabled are the following:

- Put method allows a client to upload new files on the web server. An attacker can exploit it by uploading malicious files (e.g.: an asp file that executes commands by invoking cmd.exe), or by simply using the victim's server as a file repository
- Delete method allows a client to delete a file on the web server. An attacker can exploit it as a very simple and direct way to deface a web site or to mount a DoS attack
- Connect method could allow a client to use the web server as a proxy
- Trace method simply echoes back to the client whatever string has been sent to the server, and is used mainly for debugging purposes. This method, originally assumed harmless, can be used to mount an attack known as Cross Site Tracing, which has been discovered by Jeremiah Grossman (see links at the bottom of the page)

If an application needs one or more of these methods, such as REST Web Services (which may require PUT or DELETE), it is important to check that their usage is properly limited to trusted users and safe conditions.

3.4.1.8 Management of account permissions

Description
In security, the Principle of Least Privilege encourages system designers and implementers to allow running code only the permissions needed to complete the required tasks and no more. When designing web applications the business logic of web applications must be written with authorization controls in mind.

Remediation
Once a user has authenticated to the running system, their access to resources should be limited based on their identity and roles. In addition, users attempts to perform actions should also be authorized. Both the J2EE and ASP.NET web application platforms provide the ability to declaratively limit a user's access to web resources by their identity and roles (as configured in web.xml and web.config respectively). The J2EE platform provides controls down to the method-level for limiting user access to the capabilities of EJB components. By designing file resource layouts and components APIs with authorization in mind, these powerful capabilities of the J2EE and .NET platforms can be used to enhance security.

3.4.1.9 Management of application permissions

Description
In security, the Principle of Least Privilege encourages system designers and implementers to allow running code only the permissions needed to complete the required tasks and no more. When designing web applications, the capabilities attached to running code should be limited in this manner.

Remediation
The J2EE and .NET platforms provide developers the ability to limit the capabilities of code running inside of their virtual machines. Often web applications run in environments with AllPermission (Java) or FullTrust (.NET) turned on. This limits the ability of the virtual machine to control the actions of code running under its control. Implementing code access security

measures is not only useful for mitigating risk when running untrusted code – it can also be used to limit the damage caused by compromises to otherwise trusted code.

3.4.2 Error Handling

3.4.2.1 Error Codes

Description
Often, during a penetration test on web applications, we come up against many error codes generated from applications or web servers. It's possible to cause these errors to be displayed by using a particular requests, either specially crafted with tools or created manually. These codes are very useful to penetration testers during their activities, because they reveal a lot of information about databases, bugs, and other technological components directly linked with web applications. Within this section we'll analyse the more common codes (error messages) and bring into focus the steps of vulnerability assessment. The most important aspect for this activity is to focus one's attention on these errors, seeing them as a collection of information that will aid in the next steps of our analysis. A good collection can facilitate assessment efficiency by decreasing the overall time taken to perform the penetration test.

Web Server Errors
A common error that we can see during our search is the HTTP 404 Not Found. Often this error code provides useful details about the underlying web server and associated components. For example:

```
Not Found
The requested URL /page.html was not found on this server.
Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g DAV/2 PHP/5.1.2 Server at
localhost Port 80
```

This error message can be generated by requesting a non-existant URL. After the common message that shows a page not found, there is information about web server version, OS, modules and other products used. This information can be very important from an OS and application type and version identification point of view.
Application Server Errors

Database Errors
Web server errors aren't the only useful output returned requiring security analysis. Consider the next example error message:

```
Microsoft OLE DB Provider for ODBC Drivers (0x80004005)
[DBNETLIB][ConnectionOpen(Connect())] - SQL server does not exist or access denied
```

What happened? We will explain step-by-step below.
In this example, the 80004005 is a generic IIS error code which indicates that it could not establish a connection to its associated database. In many cases, the error message will detail the type of the database. This will often indicate the underlying operating system by association. With this information, the penetration tester can plan an appropriate strategy for the security test.
By manipulating the variables that are passed to the database connect string, we can invoke more detailed errors.

```
Microsoft OLE DB Provider for ODBC Drivers error '80004005'
[Microsoft][ODBC Access 97 ODBC driver Driver]General error Unable to open
registry key 'DriverId'
```

In this example, we can see a generic error in the same situation which reveals the type and version of the associated database system and a dependence on Windows operating system registry key values.

Now we will look at a practical example with a security test against a web application that loses its link to its database server and does not handle the exception in a controlled manner. This could be caused by a database name resolution issue, processing of unexpected variable values, or other network problems.
Consider the scenario where we have a database administration web portal, which can be used as a front end GUI to issue database queries, create tables, and modify database fields. During the POST of the logon credentials, the following error message is presented to the penetration tester. The message indicates the presence of a MySQL database server:

```
Microsoft OLE DB Provider for ODBC Drivers (0x80004005)
[MySQL][ODBC 3.51 Driver]Unknown MySQL server host
```

If we see in the HTML code of the logon page the presence of a hidden field with a database IP, we can try to change this value in the URL with the address of database server under the penetration tester's control in an attempt to fool the application into thinking that the logon was successful.
Another example: knowing the database server that services a web application, we can take advantage of this information to carry out a SQL Injection for that kind of database or a persistent XSS test.

To test for ASP.net and IIS Error Handling attackers can fire up your browser and type a random page name:

```
http:\\www.mywebserver.com\anyrandomname.asp
```

If the server returns:

```
The page cannot be found

HTTP 404 - File not found
Internet Information Services
```

it means that IIS custom errors are not configured. Please note the .asp extension.
Also test for .net custom errors. Type a random page name with aspx extension in your browser

```
http:\\www.mywebserver.com\anyrandomname.aspx
```

If the server returns

```
Server Error in '/' Application.
--------------------------------------------------------------------------------

The resource cannot be found.
Description: HTTP 404. The resource you are looking for (or one of its
dependencies) could have been removed, had its name changed, or is temporarily
unavailable. Please review the following URL and make sure that it is spelled
correctly.
```

custom errors for .net are not configured.

Remediation
Error Handling in IIS and ASP .net
ASP .net is a common framework from Microsoft used for developing web applications. IIS is one of the commonly used web server. Errors occur in all applications, we try to trap most errors but it is almost impossible to cover each and every exception.
IIS uses a set of custom error pages generally found in c:\winnt\help\iishelp\common to display errors like '404 page not found' to the user. These default pages can be changed and

custom errors can be configured for IIS server. When IIS receives a request for an aspx page, the request is passed on to the dot net framework.
There are various ways by which errors can be handled in dot net framework. Errors are handled at three places in ASP .net:

- Inside Web.config customErrors section
- Inside global.asax Application_Error Sub
- At the the aspx or associated codebehind page in the Page_Error sub

Handling errors using web.config

```
<customErrors defaultRedirect="myerrorpagedefault.aspx" mode="On|Off|RemoteOnly">
   <error statusCode="404" redirect="myerrorpagefor404.aspx"/>
   <error statusCode="500" redirect="myerrorpagefor500.aspx"/>
</customErrors>
```

mode="On" will turn on custom errors. mode=RemoteOnly will show custom errors to the remote web application users. A user accessing the server locally will be presented with the complete stack trace and custom errors will not be shown to him.
All the errors, except those explicitly specified, will cause a redirection to the resource specified by defaultRedirect, i.e., myerrorpagedefault.aspx. A status code 404 will be handled by myerrorpagefor404.aspx.

Handling errors in Global.asax
When an error occurs, the Application_Error sub is called. A developer can write code for error handling/page redirection in this sub.

```
Private Sub Application_Error (ByVal sender As Object, ByVal e As System.EventArgs)
     Handles MyBase.Error
End Sub
```

Handling errors in Page_Error sub
This is similar to application error.

```
Private Sub Page_Error (ByVal sender As Object, ByVal e As System.EventArgs)
     Handles MyBase.Error
End Sub
```

Error hierarchy in ASP .net
Page_Error sub will be processed first, followed by global.asax Application_Error sub, and, finally, customErrors section in web.config file.
Information Gathering on web applications with server-side technology is quite difficult, but the information discovered can be useful for the correct execution of an attempted exploit (for example, SQL injection or Cross Site Scripting (XSS) attacks) and can reduce false positives.

3.4.2.2 Stack Traces

Description
Stack traces are not vulnerabilities by themselves, but they often reveal information that is interesting to an attacker. Attackers attempt to generate these stack traces by tampering with the input to the web application with malformed HTTP requests and other input data.
Search the code for the calls that cause an exception to be rendered to a String or output stream. For example, in Java this might be code in a JSP that looks like:

```
<% e.printStackTrace( new PrintWriter( out ) ) %>
```

In some cases, the stack trace will be specifically formatted into HTML, so be careful of accesses to stack trace elements.

Remediation
Search the configuration to verify error handling configuration and the use of default error pages. For example, in Java this configuration can be found in web.xml.

3.4.3 Cryptography

3.4.3.1 Weak SSL Protocol

Description
SSL/TLS is a collection of protocols. Weaknesses have been identified with earlier SSL protocols, including SSLv2 and SSLv3. The best practice for transport layer protection is to only provide support for the TLS protocols - TLS1.0, TLS 1.1 and TLS 1.2. This configuration will provide maximum protection against skilled and determined attackers and is appropriate for applications handling sensitive data or performing critical operations.
Nearly all modern browsers support at least TLS 1.0. As of February 2013, contemporary browsers (Chrome v20+, IE v8+, Opera v10+, and Safari v5+) support TLS 1.1 and TLS 1.2.

Remediation
You should provide support for TLS 1.1 and TLS 1.2 to accommodate clients which support the protocols.
In situations where lesser security requirements are necessary, it may be acceptable to also provide support for SSL 3.0 and TLS 1.0. SSLv3 has known weaknesses which severely compromise the channel's security. TLS 1.0 suffers CBC Chaining attacks and Padding Oracle attacks. SSLv3 and TLSv1.0 should only be used only after risk analysis and acceptance.
Under no circumstances should SSLv2 be enabled as a protocol selection. The SSLv2 protocol is broken and does not provide adequate transport layer protection.

3.4.3.2 Weak SSL/TSL Ciphers

Description
A cipher suite is specified by an encryption protocol (DES, RC4, AES), the encryption key length (such as 40, 56, or 128 bits), and a hash algorithm (SHA, MD5) used for integrity checking. Each protocol (SSLv3, TLSv1.0, etc) provides cipher suites. As of TLS 1.2, there is support for over 300 suites (320+ and counting), including national vanity cipher suites. The strength of the encryption used within a TLS session is determined by the encryption cipher negotiated between the server and the browser.
Briefly, the key points for the cipher suite determination are the following:

1. The client sends to the server a ClientHello message specifying, among other information, the protocol and the cipher suites that it is able to handle. Note that a client is usually a web browser (most popular SSL client nowadays), but not necessarily, since it can be any SSL-enabled application; the same holds for the server, which needs not to be a web server, though this is the most common case. (For example, a noteworthy class of SSL clients is that of SSL proxies such as stunnel (www.stunnel.org) which can be used to allow non-SSL enabled tools to talk to SSL services);
2. The server responds with a ServerHello message, containing the chosen protocol and cipher suite that will be used for that session (in general the server selects the strongest protocol and cipher suite supported by both the client and server).
   It is possible (for example, by means of configuration directives) to specify which cipher suites the server will honor. In this way you may control, for example, whether or not conversations with clients will support 40-bit encryption only.

Remediation
In order to ensure that only strong cryptographic ciphers are selected the server must be modified to disable the use of weak ciphers. It is recommended to configure the server to only

support strong ciphers and to use sufficiently large key sizes. In general, the following should be observed when selecting CipherSuites:

- Disable cipher suites that do not offer authentication (NULL ciphersuites, aNULL or eNULL)
- Disable anonymous Diffie-Hellman key exchange (ADH)
- Disable export level ciphers (EXP, eg. ciphers containing DES)
- Disable key sizes smaller than 128 bits for encrypting payload traffic
- Disable the use of MD5 as a hashing mechanism for payload traffic
- Use AES, 3-key 3DES for encryption operated in CBC mode
- Stream Ciphers which XOR the key stream with plaintext (such as AES/CTR mode)
- Use SHA1 or above for digests, prefer SHA2 (or equivalent)
- Support ephemeral Diffie-Hellman key exchange

https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

3.4.3.3 SSL certificate validity – client and server

Description

When accessing a web application via the HTTPS protocol, a secure channel is established between the client and the server. The identity of one (the server) or both parties (client and server) is then established by means of digital certificates. So, once the cipher suite is determined, the "SSL Handshake" continues with the exchange of the certificates, like follow:
The server sends its Certificate message and, if client authentication is required, also sends a CertificateRequest message to the client.
The server sends a ServerHelloDone message and waits for a client response.
Upon receipt of the ServerHelloDone message, the client verifies the validity of the server's digital certificate.
In order for the communication to be set up, a number of checks on the certificates must be passed. While discussing SSL and certificate based authentication is beyond the scope of this Guide, we will focus on the main criteria involved in ascertaining certificate validity:

a) checking if the Certificate Authority (CA) is a known one (meaning one considered trusted)
b) checking that the certificate is currently valid
c) checking that the name of the site and the name reported in the certificate match

Let's examine each check more in detail:

a) Each browser comes with a preloaded list of trusted CAs, against which the certificate signing CA is compared (this list can be customized and expanded at will). During the initial negotiations with an HTTPS server, if the server certificate relates to a CA unknown to the browser, a warning is usually raised. This happens most often because a web application relies on a certificate signed by a self-established CA. Whether this is to be considered a concern depends on several factors. For example, this may be fine for an Intranet environment (think of corporate web email being provided via HTTPS; here, obviously all users recognize the internal CA as a trusted CA). When a service is provided to the general public via the Internet, however (i.e. when it is important to positively verify the identity of the server we are talking to), it is usually imperative to rely on a trusted CA, one which is recognized by all the user base (and here we stop with our considerations; we won't delve deeper in the implications of the trust model being used by digital certificates).
b) Certificates have an associated period of validity, therefore they may expire. Again, we are warned by the browser about this. A public service needs a temporally valid certificate; otherwise, it means we are talking with a server whose certificate was issued by someone we trust, but has expired without being renewed.
c) What if the name on the certificate and the name of the server do not match? If this happens, it might sound suspicious. For a number of reasons, this is not so rare to see. A system may host a number of name-based virtual hosts, which share the same IP address and are identified by means of the HTTP 1.1 Host: header information. In this

case, since the SSL handshake checks the server certificate before the HTTP request is processed, it is not possible to assign different certificates to each virtual server. Therefore, if the name of the site and the name reported in the certificate do not match, we have a condition which is typically signaled by the browser. To avoid this, IP-based virtual servers must be used.

Remediation
Examine the validity of the certificates used by the application at both server and client levels. The usage of certificates is primarily at the web server level; however, there may be additional communication paths protected by SSL (for example, towards the DBMS). You should check the application architecture to identify all SSL protected channels.
Certificate must be signed with a strong hashing algorithm (SHA-256 or greater).
Ensure that RSA key of certificates are equal or greater than 2048 bits.

3.4.3.4 Insecure Renegotiation

Description
A design weakness in TLS, identified as CVE-2009-3555, allows an attacker to inject a plaintext of his choice into a TLS session of a victim. In the HTTPS context the attacker might be able to inject his own HTTP requests on behalf of the victim.

Remediation
Insecure Renegotiation (CVE-2009-3555) issue can be mitigated either by disabling support for TLS renegotiations or by supporting only renegotiations compliant with RFC 5746. All modern browsers have been updated to comply with this RFC.

https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

3.4.3.5 Compression and CRIME attack

Description
Compression Ratio Info-leak Made Easy (CRIME) is an exploit against the data compression scheme used by the TLS and SPDY protocols. The exploit allows an adversary to recover user authentication cookies from HTTPS. The recovered cookie can be subsequently used for session hijacking attacks.

Remediation
Disable compression for avoid CRIME attack.

https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

3.4.3.6 Credentials storage

Description
Media covers the theft of large collections of passwords on an almost daily basis. Media coverage of password theft discloses the password storage scheme, the weakness of that scheme, and often discloses a large population of compromised credentials that can affect multiple web sites or other applications. Proper storage helps prevent theft, compromise, and malicious use of credentials. Information systems store passwords and other credentials in a variety of protected forms. Common vulnerabilities allow the theft of protected passwords through attack vectors such as SQL Injection. Protected passwords can also be stolen from artifacts such as logs, dumps, and backups.
Specific guidance herein protects against stored credential theft but the bulk of guidance aims to prevent credential compromise. That is, this guidance helps designs resist revealing users' credentials or allowing system access in the event threats steal protected credential information

Remediation

For solve this issue is necessary:

- do not limit the character set or length of credentials. Some organizations restrict the 1) types of special characters and 2) length of credentials accepted by systems because of their inability to prevent SQL Injection, Cross-site scripting, command-injection and other forms of injection attacks. These restrictions, while well-intentioned, facilitate certain simple attacks such as brute force
  Do not apply length, character set, or encoding restrictions on the entry or storage of credentials. Continue applying encoding, escaping, masking, outright omission, and other best practices to eliminate injection risks
- use a cryptographically strong credential-specific salt. A salt is fixed-length cryptographically-strong random value. Append credential data to the salt and use this as input to a protective function. Store the protected form appended to the salt as follows:

  ```
  [protected form] = [salt] + protect([protection func], [salt] + [credential]);
  ```

  Follow these practices to properly implement credential-specific salts:
  - Generate a unique salt upon creation of each stored credential (not just per user or system wide)
  - Use cryptographically-strong random data
  - As storage permits, use a 32bit or 64b salt (actual size dependent on protection function)
  - Scheme security does not depend on hiding, splitting, or otherwise obscuring the salt
  - Salts serve two purposes: 1) prevent the protected form from revealing two identical credentials and 2) augment entropy fed to protecting function without relying on credential complexity. The second aims to make pre-computed lookup attacks on an individual credential and time-based attacks on a population intractable

- impose infeasible verification on attacker. The function used to protect stored credentials should balance attacker and defender verification. The defender needs an acceptable response time for verification of users' credentials during peak use. However, the time required to map <credential> → <protected form> must remain beyond threats' hardware (GPU, FPGA) and technique (dictionary-based, brute force, etc) capabilities. Two approaches facilitate this, each imperfectly.
  - Leverage an adaptive one-way function Adaptive one-way functions compute a one-way (irreversible) transform. Each function allows configuration of 'work factor'. Underlying mechanisms used to achieve irreversibility and govern work factors (such as time, space, and parallelism) vary between functions and remain unimportant to this discussion.
    Select:
    - PBKDF2 when FIPS certification or enterprise support on many platforms is required;
    - Scrypt where resisting any/all hardware accelerated attacks is necessary but support isn't.

    Example protect() pseudo-code follows:

    ```
    return [salt] + pbkdf2([salt], [credential], c=10000);
    ```

    Designers select one-way adaptive functions to implement protect() because these functions can be configured to cost (linearly or exponentially) more than a hash function to execute. Defenders adjust work factor to keep pace with threats' increasing hardware capabilities. Those implementing adaptive one-way

functions must tune work factors so as to impede attackers while providing acceptable user experience and scale.
Additionally, adaptive one-way functions do not effectively prevent reversal of common dictionary-based credentials (users with password 'password') regardless of user population size or salt usage

- Leverage Keyed functions. Keyed functions, such as HMACs, compute a one-way (irreversible) transform using a private key and given input. For example, HMACs inherit properties of hash functions including their speed, allowing for near instant verification. Key size imposes infeasible size- and/or space-requirements on compromise--even for common credentials (aka password = 'password'). Designers protecting stored credentials with keyed functions:
  - Use a single "site-wide" key
  - Protect this key as any private key using best practices
  - Store the key outside the credential store (aka: not in the database)
  - Generate the key using cryptographically-strong pseudo-random data
  - Do not worry about output block size (i.e. SHA-256 vs. SHA-512)

Example protect() pseudo-code follows:

```
return [salt] + HMAC-SHA-256([key], [salt] + [credential]);
```

Upholding security improvement over (solely) salted schemes relies on proper key management

- Design password storage assuming eventual compromise. The frequency and ease with which threats steal protected credentials demands "design for failure". Having detected theft, a credential storage scheme must support continued operation by marking credential data compromised and engaging alternative credential validation workflows as follows:
  - Protect the users account
    1. Invalidate authentication 'shortcuts' disallowing login without 2nd factors or secret questions.
    2. Disallow changes to user accounts such as editing secret questions and changing account multi-factor configuration settings.
  - Load and use new protection scheme
    1. Load a new (stronger) protect(credential) function
    2. Includ e version information stored with form
    3. Set 'tainted'/'compromised' bit until user resets credentials
    4. Rotate any keys and/or adjust protection function parameters (iter count)
    5. Increment scheme version number
  - When user logs in:
    1. Validate credentials based on stored version (old or new); if old demand 2nd factor or secret answers
    2. Prompt user for credential change, apologize, & conduct out-of-band confirmation
    3. Convert stored credentials to new scheme as user successfully log in

https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet

3.4.3.7 Session data storage

Description
Application servers should use private temporary file areas per client/application to store session data.

Remediation
Storage areas must be encrypted with a strong enctyption algorithm (AES al least).

3.4.3.8 Database ecryption and key management

Description
Restricted data is encrypted during transmission over the network using encryption measures strong enough to minimize the risk of the data's exposure if intercepted or misrouted from database to client workstation.
If database-level encryption for restricted data is implemented, procedures for secure key management are documented. (Check National Institute of Standards and Technology (NIST) for current recommendations.) Note: It is recommended that all application layers (network, application, client workstation) are already encrypted before encrypting the database. Database encryption is not a substitute for any of the above requirements. Database encryption of restricted data is not mandatory to meet this standards document.
For data subject to disclosure that is encrypted at storage, the means to decrypt must be available to more than one person and approved by the data proprietor.

Remediation
Backup tapes store backups of the database in an encrypted format, and the tapes do not store the plain text encryption keys necessary to decrypt the backups.
Key management procedures for decrypting backups are documented, available to more than one person and approved by the data proprietor.

3.4.4 Logging and Audit

3.4.4.1 Database administrators logs

Description
Database administrators activities should be logged.

Remediation
Logs should include:
- the time at which an event (log-in; log-out; log-failed) occurred;
- information about the event (e.g. files handled) or failure (e.g. error occurred and corrective action taken);
- which account and which database administrator was involved;
- which processes were involved;
- All login logs are retained for at least six months.

Database administrators logs should be reviewed on a regular basis.

3.4.4.2 Protection of log information

Description
Logging facilities and log information should be protected against tampering and unauthorized access.

Remediation
Controls should aim to protect against unauthorized changes and operational problems with the logging facility including:
- alterations to the message types that are recorded;
- log files being edited or deleted;
- storage capacity of the log file media being exceeded, resulting in either the failure to record events or over-writing of past recorded events.

Set all logs read-only and sign them with a digital signature or a hash function (e.g. MD5).

3.4.5 Backup & Recovery

Description
The backup and recovery procedures are documented and meet data proprietor's requirements.

Remediation
Backup and recovery procedures are periodically tested.
Backup retention intervals are documented and sufficient to meet the business resumption requirements and expectations of the data proprietor.