



All Tasks Report

Application: Ozel

Project: Ozel

Prepared on: 06/16/2012

Table of Contents

[Section 1: Introduction](#)

[Section 2: Project Settings](#)

[Section 3: Tasks](#)

[3.1: Requirements](#)

[3.2: Architecture & Design](#)

[3.3: Development](#)

[3.4: Testing](#)

Section 1: Introduction

This report provides a complete set of prescriptive tasks to perform during the development of application Ozel Ozel. Developers and Testers can use this as a guide on how to embed security into their software. Each task solves an underlying security weakness, and tasks are sorted by their priority of the underlying weakness from highest to lowest.

Section 2: Project Settings

The following is the project's settings as configured in SD Elements. These settings produced the Tasks listed in Section 3 of this report.

Application General

- Kinds of users is Internal and external users
- HTTP-based protocols used is RESTful web services
- HTTP-based protocols used is SOAP web services
- Communication protocols used is Uses HTTP-based Protocol
- Web application include RESTful web services
- Application type is Web application

Language and Platform

- Technology/Platform is Android Application Framework
- Name of web server is Apache
- Programming language is Java
- Technology/Platform is Java EE
- Programming language is JavaScript
- Other serialization formats is JSON
- Generates or reads XML from end user or remote system
- Uses XML digital signatures
- Uses XSLTs

Features and Functions

- Authentication of end users is Directly authenticates end users
- Password Management Features is Has change existing password function
- Miscellaneous is Has file-upload function
- Input validation is Interacts with the O/S
- Encryption is Passwords stored in configuration files
- Single Sign On Tool is SiteMinder
- Input validation is Uses regular expressions on end-user input

- Session management configuration is Uses server-provided session management
- Encryption is Uses WS-Security standard
- Authentication backend is Using a Single Sign On (SSO) suite
- Uses passwords for authentication
- Authorizes users
- Has session management
- Serves files which should not be publicly viewable
- Requires remember-me (i.e. user does not have to log in next time) function
- Password Management Features is Has forgot password function
- Single Sign On (SSO) suite being used is SAML
- Password Management Features is Auto-generates passwords for new users
- Single Sign On (SSO) suite being used is OAuth
- Single Sign On (SSO) suite being used is OpenID

Compliance Requirements

- Regions you do business in is USA
- Regions you do business in is Canada

Development/Test Tools

- Open source security library use permitted

Changes Since Last Release

- Changes since last release is Changes to servers/frameworks and/or configuration
- Changes since last release is Changes to authentication
- Changes to user input/output since last release is Changes to user output
- Changes since last release is Changes to session management
- Changes to user input/output since last release is New/modified user input OR changes to how user input is used
- Changes since last release is New transactions / use cases
- Changes since last release is Changes to inbound/outbound interfaces

Section 3: Tasks

3.1: Requirements

CT160: Old Fashion Thinking

Priority: 10

Problem: The following solution is always applicable.

Solution: It is assumed the application will be secure. --- Custom always applicable rule ---

T21: Ensure password and session ID are sent over SSL

Priority: 10

Problem: 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.

Solution: Passwords and session IDs must always be sent over SSL for a security-sensitive application. Enforce this by explicitly refusing plaintext HTTP requests for the login and authenticated portions of the application.

How-To Implement:

I257: Java EE with Wicket 1.5 and above

Wicket ships with the `HttpRequestCycleProcessor` class from package `wicket.protocol.https` that handles HTTP and HTTPS protocol switching for the user based on page annotations: `@RequireHttps`.

Additionally, if a page is required to be served over HTTP, Wicket allows you to have a form on this page to submit to HTTPS.

Here is a sample of how to use the package to enable SSL:

```
class MyApplication extends WebApplication {
  @Override
  protected IRequestCycleProcessor newRequestCycleProcessor() {
    return new HttpsRequestCycleProcessor(config);
  }
}
```

I220: Rails with Devise

Description

It is up to the HTTP server to use SSL connections where necessary, and Rails does not provide its own production-grade HTTP server. However, the SSL Requirement plugin for Devise allows Rails to confirm that SSL is being used.

Code

Include the plugin in the application controller:

```
class ApplicationController < ActionController::Base
  include SslRequirement
end
```

[Download the Complete Code Example](#)

Add the following to config/application.rb (or config/environments/production.rb if SSL is only requirement in production):

```
config.to_prepare
{ Devise::SessionsController.ssl_required :new, :create }
```

[Download the Complete Code Example](#)

I176: Django

Description

Django does not include its own production-grade HTTP server, and therefore does not provide SSL. To comply with this standard, see the accompanying implementation for your HTTP server software.

I81: ASP.Net

Description

When using ASP.NET FormsAuthentication, the application should verify that the "requireSSL" attribute on the system.web.authentication.forms configuration element is set to "True." This forces the authentication cookie to specify the secure attribute. This requires the browser to provide the cookie over SSL only. In web.config:

Code

```
<configuration>
  <system.web>
    <authentication mode="Forms">
      <forms loginUrl="~/login.aspx"
        protection="All"
        requireSSL="true"/>
      <!--
        forces Browser to only provide Cookies over SSL
      -->
    </authentication>
  </system.web>
</configuration>
```

[Download the Complete Code Example](#)

I10: Java EE

Description

Use the confidential transport guarantee in the Servlet container to enforce that the entire site is sent over SSL. Note that you will need configure SSL separately from this.

Code

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>All URLs</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
```



```
<transport-guarantee>CONFIDENTIAL</transport-guarantee>  
</user-data-constraint>  
</security-constraint>
```

[Download the Complete Code Example](#)

T6: Implement account lockout or authentication throttling

Priority: 9

Problem: The software does not implement sufficient measures to prevent multiple failed authentication attempts within in a short time frame, making it more susceptible to brute force attacks.

Solution: In order to protect against brute-forcing, implement a feature to lock out users after a configurable number of failed authentication attempts. PCI DSS 8.5.13 specifies that the system need to lockout the account after six failed attempts.

Alternatively, consider a mechanism to throttle multiple authentication attempts for the same user ID or originating from the same user ID. This has the benefit of not locking out legitimate user accounts while also decreasing the likelihood of a brute-force attack. Exponentially increase the amount of time a user has to wait between authentication attempts until it reaches a rate that makes brute-forcing impractical (e.g. 24 hours).

T50: Use indirect object reference maps if accessing files

Priority: 9

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

this weakness would be if a system used sequential or otherwise easily guessable session ids that would allow one user to easily switch to another user's session and view/modify their data.

Solution: Always use indirect references for a specific file, a set of files, or all files in a particular directory or directories. Applications often allow users to access sensitive resources such as user-specific files from the application server. Direct object references use the actual file name (e.g., "file=statement1.pdf") whereas indirect object references provide an independent identifier that the application later translates into an actual filename (e.g., "file=a", where 'a' later translates to statement1.pdf). The problem with the former method is that attackers can sometimes access files that they shouldn't (e.g., "file=../config.xml"). An indirect object reference renders such an attack impossible because the application only provides access to a specified set of files (e.g., all files in a particular directory, or a predefined list of individual files).

Note that this control applies specifically to resources that require access control.

Publicly-accessible static content such as JavaScript or Cascading Style Sheet files that are normally stored on web servers do not necessarily need this protection.

T57: Do not accept user-supplied XSLTs

Priority: 9

Problem: XSLTs transform an XML document from one format to another. XSLTs are themselves XML documents with programming language constructs such as if statements and for loops. An attacker-controlled XSLT could, for example, cause a Denial of Service (DoS) condition with an infinite loop. In some cases, XSLT engines such as Xalan for Java provide extensions into the underlying programming language. This means that, in some cases, an attacker may be able to perform remote command execution using an XSLT.

Solution: XML stylesheet language transforms (XSLTs) can be used for malicious purposes, particularly if the XSLT processor provides extensions for enhanced functionality. Do not accept user-supplied XSLTs.

T58: Do not process user-supplied XSLTs in XML digital signatures

Priority: 9

Problem: XSLTs transform an XML document from one format to another. XSLTs are themselves XML documents with programming language constructs such as if statements and for loops. An attacker-controlled XSLT could, for example, cause a Denial of Service (DoS) condition with an infinite loop. In some cases, XSLT engines such as Xalan for Java provide extensions into the underlying programming language. This means that, in some cases, an attacker may be able to perform remote command execution using an XSLT.

Solution: By spec, XML Digital Signatures (part of the WS-Security standard) allow users to provide XML Stylesheet Language Transforms (XSLTs.) Default implementations of popular XSLT libraries often provide potentially dangerous extensions, such as the ability to access the runtime environment.

Where possible, avoid using XML Digital Signatures entirely. When you must use XML Digital Signatures, turn off support for processing XSLTs.

T2: Secure forgotten password

Priority: 8

Problem: It is common for an application to have a mechanism that provides a means for a user to gain access to their account in the event they forget their password. Very often the password recovery mechanism is weak, which has the effect of making it more likely that it would be possible for a person other than the legitimate system user to gain access to that user's account. This weakness may be that the security question is too easy to guess or find an answer to (e.g. because it is too common). Or there might be an implementation weakness in the password recovery mechanism code that may for instance trick the system into e-mailing the new password to an e-mail account other than that of the user. There might be no throttling done on the rate of password resets so that a legitimate user can be denied service by an attacker if an attacker tries to recover their password in a rapid succession. The system may send the original password to the user rather than generating a new temporary password. In summary, password recovery functionality, if not carefully designed and implemented can often become the system's weakest link that can be misused in a way that would allow an attacker to gain unauthorized access to the system. Weak password recovery schemes completely undermine a strong password authentication scheme.

Solution: Insecure forgotten password mechanisms are one of the easiest ways for attackers to break into an application. In particular, forgotten password questions are often easy to answer and are susceptible to brute-forcing. If the application provides password reset functionality, it should use the following sequence or one that provides a similar level of security:

1. Users visit a password-reset page where they submit their User ID or email address. The page should also include an anti-automation technique, such as a CAPTCHA image with accessibility options for visually impaired users, to prevent user enumeration
2. Upon form submission, the site displays a page indicating that an email with instructions was sent to the user
In the interim, the site should send an email with a temporary link to the user. The link should expire after a short period of time (e.g., 1 hour)
3. When the user clicks on the link, they should be taken to a page where they respond to pre-determined forgotten password questions
4. The application should enforce account lockout if the user fails to correctly answer a question after a configurable number of tries (e.g. five wrong answers)
5. Users should be informed via email that their password has been changed, but the password itself should never be sent via email

Note that in some cases, your organization may have a policy not to send emails to end users. If your organization has this policy, move directly from step 1 to step 3 above and avoid sending an email link.

T5: Minimum password standards

Priority: 7

Problem: An authentication mechanism is only as strong as its credentials. For this reason, it is important to require users to have strong passwords. Lack of password complexity significantly reduces the search space when trying to guess user's passwords, making brute-force attacks easier.

Solution: In new user registration and password reset, ensure that passwords meet minimum standards. These standards are generally outlined in an enterprise security policy. If your organization does not have such a policy, consider referring to a standard such as [the SANS Password Policy](#).

T26: Destroy sessions on logout

Priority: 7

Problem: A user's session should expire upon logging out of the application. If the session does not expire, this may provide a window of opportunity for an attacker to hijack the session. This is particularly important for applications on shared devices.

Solution: Explicitly destroy users' sessions upon logout. This protects against users on a shared machine who may access another user's session, even though the first user explicitly logged out.

How-To Implement:

I24: Java EE

Description

The following code implements a simple logout method in a Servlet:

Code

```
private void logout(  
    HttpServletRequest request,  
    HttpServletResponse response  
) throws ServletException, IOException {  
    //Immediately invalidate session and forward to home page  
    //Log that the user has logged out here  
    request.getSession().invalidate();  
    response.sendRedirect("/");  
}
```

[Download the Complete Code Example](#)

T7: Salt and hash stored password

Priority: 6

Problem: This makes it easier for attackers to pre-compute the hash value using dictionary attack techniques such as rainbow tables, effectively disabling the protection that an unpredictable salt would provide.

Solution: A salt value is a set of random bytes added to the original input prior to hashing. Add salt values and hash all passwords.

Appending a salt value has three major benefits:

- A rainbow table must be created for each salt value, since a standard rainbow table will hash each value directly without appending a salt. A hash of a value that has been salted will most likely not exist in the rainbow table.
- If different salt values are used for each value that is being hashed, then two identical passwords will not hash to the same value.
- Salting a value generally increases the complexity of the value (either by increasing it in length or by adding special characters), thereby increasing the possible value space and making dictionary attacks much more difficult.

Use a slow hashing algorithm, such as one of the SHA-2 algorithms, and hash the password and salt at least a thousand times. Where possible, do not write your code to perform this function. Instead, try to use a field-tested library that performs the hashing and salting for you.

T8: Consistent error handling for all authentication failures

Priority: 6

Problem: This issue frequently occurs during authentication, where a difference in failed-login messages could allow an attacker to determine if the username is valid or not. These exposures can be inadvertent (bug) or intentional (design).

Solution: Provide the same error message to users regardless of if they fail authentication because of an invalid username or an invalid password. Differing messages may result in a user-enumeration vulnerability.

How-To Implement:

I3: Java

Description

The following code demonstrates sending a consistent error message for both an invalid username and an invalid password:

Code

```
final private String FAILED_AUTH_MSG = "Invalid username and/or password";
try{
    User u = getUser(request.getParameter("username"));
    u.checkPassword(request.getParameter("password"));
} catch Exception(InvalidUserException iue) {
    throw new AuthenticationException(this.FAILED_AUTH_MSG);
} catch Exception(PasswordFailException pfe) {
    throw new AuthenticationException(this.FAILED_AUTH_MSG);
}
```

[Download the Complete Code Example](#)

T53: Virus scan all uploaded files using an inline virus scanner

Priority: 6

Problem: The software allows the attacker to upload or transfer files of dangerous types that can be automatically processed within the product's environment.

Solution: Perform a virus scan on any user-uploaded files using an inline virus scanner, such as Clam AV. This helps protect against a malicious user who may upload malware for other users to download. Alternatively, an attacker may attempt to upload malware and then use a different vulnerability to execute the file.

T55: Use XSDs instead of DTDs

Priority: 6

Problem: XML documents optionally contain a Document Type Definition (DTD), which, among other features, enables the definition of "XML entities". It is possible to define an entity locally by providing a substitution string in the form of a URL whose content is substituted for the XML entity when the DTD is processed. The attack can be launched by defining an XML entity whose content is a file URL (which, when processed by the receiving end, is mapped into a file on the server), that is embedded in the XML document, and thus, is fed to the processing application. This application may echo back the data (e.g. in an error message), thereby exposing the file contents.

Solution: User-supplied Document Type Definitions (DTDs) can be susceptible to several kinds of attacks, including arbitrary file browsing on the server and denial-of-service.

Where possible, use XSDs instead of DTDs when dealing with external and/or untrusted sources. If you must use DTDs, do not allow external users to provide the DTD or perform a strong (whitelist) validation before processing.

T3: Require old passwords when users change password

Priority: 5

Problem: This could be used by an attacker to change passwords for another user, thus gaining the privileges associated with that user.

Solution: If a user wishes to change their password outside of the "forgotten password" feature of an application, require the user to first enter their old password and have the application verify that this is the correct password. This protects against cases where a malicious agent hijacks an existing user's session and changes the password without that user's knowledge.

T9: Implement transactional authentication for high-value transactions

Priority: 5

Problem: Due to the high number of real world attacks that bypass authentication or session management, attackers are often able to perform any transaction once they compromise a user's account or session. Without sufficient controls, an attacker may be able to perform high sensitive transactions such as transfer large sums of money.

Solution: In order to approve particularly high-value transactions, require users to re-enter their password or to use a one-time password sent out-of-band (e.g. SMS). For example, an online banking system may do this for money transfers over a certain threshold (e.g. \$1,000). Define which applications are "high value" on an application-by-application basis.

T24: Enforce idle session timeouts

Priority: 5

Problem: According to WASC, "Insufficient Session Expiration is when a web site permits an attacker to reuse old session credentials or session IDs for authorization."

Solution: Idle-session timeouts decrease the window of opportunity for an attacker to steal or brute-force a session ID. They also safeguard users who forget to explicitly log out at a shared computer. Ensure that your application has an appropriate idle session timeout. Choose a timeout value commensurate with your users' needs; generally between 15 and 30 minutes.

PCI DSS 8.5.15 specifies the session timeout value should be 15 minutes.

How-To Implement:

I20: Java EE

Description

Configure a session timeout through the deployment descriptor (i.e. web.xml). The "<session-config>" element defines session attributes for the application. The "<session-timeout>" sub-element allows you to specify the idle timeout in minutes. This element is optional in web.xml, with a default value of 60. Declaratively:

Code

```
<web-app ...>
<session-config>
  <session-timeout>15</session-timeout>
</session-config>
</web-app>
```

Programmatically:

```
public void doPost(
    HttpServletRequest request,
    HttpServletResponse
    response
) throws ... {
    // Obtain existing session:
```

```
HttpSession session = request.getSession(true);
session.setMaxInactiveInterval(15*60);
}
```

[Download the Complete Code Example](#)

I21: SiteMinder 6

Description

Configure the "Idle Timeout" attribute for the User Session. This needs to be configured separately for each realm. See

https://support.ca.com/cadocs/0/CA%20SiteMinder%20r6%200%20SP6-ENU/Bookshelf_Files/HTML/index.I

T27: Turn off session rewriting

Priority: 5

Problem: Applications that allow URL rewriting for session IDs may leak the session IDs to other users on the same computer in the browser history or through links to external sites.

Solution: When the user's browser does not support session cookies, application servers may explicitly include session identifiers in a URL. Applications that allow session identifiers in the URL may leak the session ID to external sites through hyperlinks, may be more susceptible to attacks such as session fixation, and may leave the session ID in the user's browsing history or other logs.

Explicitly disallow session rewriting on your application.

How-To Implement:

I25: Java EE, Servlet Spec 2.1+

Description

Note: If you are not sure which Servlet API version you are using, you can use the following snippet to fetch the version number via a JSP file:

```
Servlet version: <%= application.getMajorVersion() %>.<%= application.getMinorVersion() %>
```

Inside of a Servlet:

Code

```
if (HttpServletRequest.isRequestedSessionIdFromURL()) {  
    //take an appropriate action, based on application specific needs, such as:  
    //invalidate the session so that it can't be reused, log the issue,  
    //lock the, user's account, redirect the user to a page describing  
    //how to use cookies, inform site admins, etc.  
    //here, we invalidate the session so that it cannot be reused  
    request.getSession().invalidate();  
}
```

[Download the Complete Code Example](#)

I26: Java EE, Servlet Spec older than version 2.1

Description

Note: If you are not sure which Servlet API version you are using, you can use the following snippet to fetch the version number via a JSP file:

```
Servlet version: <%= application.getMajorVersion() %>.<%= application.getMinorVersion() %>
```

Inside of a Servlet:

Code

```
if (HttpServletRequest.isRequestedSessionIdFromUrl()) {  
    //take an appropriate action, based on application specific needs, such as:  
    //invalidate the session so that it can't be reused, log the issue,  
    //lock the user's account, redirect the user to a page describing  
    //how to use cookies, inform site admins, etc.  
    //here, we invalidate the session so that it cannot be reused  
    request.getSession().invalidate();  
}
```

[Download the Complete Code Example](#)

I27: Java EE, Servlet Spec 2.3+

Description

This code takes advantage of the "HttpServletRequestWrapper" object, which allows you to simply disable session rewriting in a filter rather than inside the Servlet itself.

Note: If you are not sure which Servlet API version you are using, you can use the following snippet to fetch the version number via a JSP file:

```
Servlet version: <%= application.getMajorVersion() %>.<%= application.getMinorVersion() %>
```

Code

```
public void doFilter(
    HttpServletRequest request,
    HttpServletResponse response,
    FilterChain chain
) throws IOException, ServletException {

    log.debug(null, "***** doFilter *****");

    //if this is not an http request, do nothing
    if (!(request instanceof HttpServletRequest)) {
        chain.doFilter(request, response);
        return;
    }

    //cast the generic request and response objects
    //to http request and response
    HttpServletRequest httpRequest = (HttpServletRequest) request;
    HttpServletResponse httpResponse = (HttpServletResponse) response;

    //detect whether the session ID for the current request
    //comes from a rewritten URL
    if (httpRequest.isRequestedSessionIdFromURL()) {
        //take an appropriate action,
        //based on application specific needs, such as:
        //invalidate the session so that it can't be reused, log the issue,
        //lock the user's account, redirect the user to a page describing
        //how to use cookies, inform site admins, etc.

        //here, we invalidate the session so that it cannot be reused
        httpRequest.getSession().invalidate();
    }
}
```

```

}

//next override the existing HttpServletResponse to
//disable common session rewriting methods in our output
HttpServletResponseWrapper wrappedResponse =
    new HttpServletResponseWrapper(httpResponse) {

    public String encodeRedirectUrl(String url) {
        return url;
    }

    public String encodeRedirectURL(String url) {
        return url;
    }

    public String encodeUrl(String url) {
        return url;
    }

    public String encodeURL(String url) {
        return url;
    }
};

//pass the original request and our wrapped response
//to the next filter in the chain
chain.doFilter(request, wrappedResponse);
}

```

[Download the Complete Code Example](#)

I28: Java EE, Servlet Spec 3.x

Description

This code uses the "<tracking-mode>" element in web.xml in ServletSpec 3, where we can force the Servlet container to use a particular mechanism to track sessions.

Note: If you are not sure which Servlet API version you are using, you can use the following snippet to fetch the version number via a JSP file:

Servlet version: <%= application.getMajorVersion() %>.<%= application.getMinorVersion() %>

Code

```
<session-config>
  <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

[Download the Complete Code Example](#)

I29: SiteMinder 6

Description

SiteMinder provides support for cookie-based sessions only; session rewriting is not an issue.

T186: Maintain the latest security patch level for third party libraries and software

Priority: 5

Problem: Insufficient patch maintenance for third party libraries or software can result in the whole system to become vulnerable to third party security weaknesses.

Solution: Either use the latest version or apply the latest security patches for any third party libraries or software being used in the system. Regularly reviewing and addressing the security vulnerabilities reported for the software the system relies on will prevent the third party vulnerabilities compromising the security of the whole system.

Note that it is essential to review all the HowTos for this task as each HowTo applies to a different platform or library being used.

Most third party library/framework vendors publish security bulletins for their products via their website directly. Additionally, the following sources can be used to locate security advisories and details about required patch levels for all commonly available products/libraries:

- [Security Focus Vulnerability Database](#) advisories categorized by vendor->Product->Version
- [National Vulnerability Database \(NVD\)](#)
- [Common Vulnerability Enumerator \(by MITRE Foundation\)](#)

T185: Follow best practices to secure SAML implementations

Priority: 5

Problem: In a system that implements an integrated authentication for the purposes of providing a single sign on experience, a weakness in the protocol, or the implementation of it can allow an attacker to impersonate a legitimate user and gain unauthorized access to the service provider and its resources.

Solution: Follow these requirements in your SAML implementation to ensure that the integrated login via SAML does not result in a weakness/vulnerability in your authentication schema:

For in-band metadata and/or X509 certificates, i.e. when they are provided through a URL, use SSL/HTTPS along with a certificate to be validated by the remote party to ensure authenticity. Follow this both when providing metadata and accepting metadata. Alternatively, metadata and/or X509 certificates can be provided to remote parties out-of-band (manually). More specifically:

a. Reject any attempt to retrieve metadata through non-HTTPS URLs. b. Reject any metadata URL that does not start with "https://" c. Ensure that the certificate chain is fully validated when using https URLs.

Reject any attempt to use unencrypted channels for SAML communication. Restrict SAML communication/bindings to SSL-enabled channels of communication, such as HTTPS, for confidentiality reasons.

Do not skip on any of the protocol-specified verifications and avoid any simplifications of the protocol specifications.

Use HTTP Post binding in favor of HTTP Redirect binding to avoid data being cached/observed on proxy nodes along the way.

Log events of failed validations.

T4: Configurable password policies

Priority: 4

Problem: An authentication mechanism is only as strong as its credentials. For this reason, it is important to require users to have strong passwords. Lack of password complexity significantly reduces the search space when trying to guess user's passwords, making brute-force attacks easier.

Solution: Application administrators should be able to configure password requirements, including:

- Password complexity requirements
- Password age / expiry (minimum and maximum)
- Password history
- Optional dictionary of words that cannot be used as passwords

T20: Invalidate old session ID after authentication

Priority: 4

Problem: Such a scenario is commonly observed when: 1. A web application authenticates a user without first invalidating the existing session, thereby continuing to use the session already associated with the user 2. An attacker is able to force a known session identifier on a user so that, once the user authenticates, the attacker has access to the authenticated session 3. The application or container uses predictable session identifiers. 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 associate, and possibly authenticate, against the server using that session identifier, giving the attacker access to the user's account through the active session.

Solution: After a user successfully authenticates into a web application, reset the session ID in order to protect against session-fixation attacks. Take care to copy the server-side state associated with the old session to the new one when you do this.

How-To Implement:

I9: Java EE

Description

After validating that the user exists, obtain the session object that was created when the user visited login.jsp without authenticating. Kill this session by making a call to the "invalidate()"

method, and then create a new session object by making a call to "getSession()," passing a boolean "true" parameter, and creating a new session ID. Now we can bind attributes to the session and then authenticate with this new session. Note that this code assumes there are no attributes currently bound to the session.

Code

```
protected void doPost(  
    HttpServletRequest request,  
    HttpServletResponse response  
) throws ServletException, IOException {  
    String username = request.getParameter("username");  
    String password = request.getParameter("password");  
    String ipAddress = request.getRemoteAddr();  
  
    if (AuthenticationService.login(username, password, ipAddress)){  
        // Get the user's session created on login.jsp page  
        HttpSession session = request.getSession(false);  
  
        // Kill the initial session  
        if (session != null) {  
            session.invalidate();  
  
            // Create new session  
            HttpSession newSession = request.getSession(true);  
        }  
  
        response.sendRedirect("success.html");  
    }  
    else {  
        response.sendRedirect("failure.html");  
    }  
}
```

[Download the Complete Code Example](#)

T22: Set secure flag on session cookies

Priority: 4

Problem: The Secure attribute for sensitive cookies in HTTPS sessions is not set, which could cause the user agent to send those cookies in plaintext over an HTTP session.

Solution: Always ensure that the user's session cookie has the "secure" flag set to true. This forces browsers to send the cookie only over an encrypted channel such as SSL.

How-To Implement:

I14: Java EE , Servlet Spec 3+

Description

The following example shows how to explicitly set the "secure" flag value for Java EE when the servlet container supports version 3 or over in web.xml:

Code

```
<session-config>
  <cookie-config>
    <http-only>true</http-only>
    <secure>true</secure>
  </cookie-config>
</session-config>
```

[Download the Complete Code Example](#)

I15: SiteMinder 6

Description

In siteminder, set the "UseSecureCookies" parameter to true. Note that if you are using secure cookies across multiple domains, you may need to set the "UseSecureCPCookies" parameter to true instead. For more information, please see <https://support.ca.com/cadocs/0/h006171e.pdf>, pg.s 89 & 90

T23: Set HttpOnly flag on session cookies

Priority: 4

Problem: The 'HttpOnly' cookie flag reduces the impact of cross-site scripting. Cookies with the 'HttpOnly' cookie flag cannot be accessed by JavaScript. Session IDs in cookie without the

'HttpOnly' cookie flag may be read and transmitted by malicious JavaScript, allowing an adversary to gain access to the user's session.

Solution: Add the "; HttpOnly" flag to the end of a cookie value. Most modern browsers will enforce that the cookie cannot be accessed by JavaScript, thereby counteracting some types of session hijacking which use Cross -Site Scripting (XSS).

How-To Implement:

I16: Java EE , Servlet Spec 3+

Description

The following example shows how to explicitly set the HttpOnly flag value for Java EE when the servlet container supports version 3 or over in web.xml:

Code

```
<session-config>
  <cookie-config>
    <http-only>true</http-only>
    <secure>true</secure>
  </cookie-config>
</session-config>
```

[Download the Complete Code Example](#)

T180: Provide privacy preferences to the browser via P3P

Priority: 4

Problem: Not communicating a privacy policy might result in a lack of trust for the application/service. In some cases, like browsers, this might result in blocking functionality or preventing users partially or completely from accessing the service.

Solution: - Include P3P headers in HTTP response headers - The Compact Policy Field of P3P headers should match the privacy policy available to the users in human readable text form.

About P3P

As an organization, you need to establish a clear privacy policy to communicate to users how their data is being handled. Platform for Privacy Policy, commonly called P3P, provides a structured mean for applications to communicate this to users and the applications on user's end that deal with the service called user agents.

Specifically, user agents such as browsers look for the P3P headers in HTTP responses and use it to determine how the cookies must be handled.

Note that you might consider adding P3P header to your HTTP responses even if you do not handle personal/private data. Browsers use the compact policy field of P3P headers to determine how to handle cookies based on the privacy preferences specified by the user.

An example of P3P header in HTTP response is:

```
HTTP/1.0 200 OK
...
P3P: policyref="/w3c/p3p.xml", CP="ALL DSP COR CURa OUR IND COM NAV CNT"
...
```

For details about what each token means in the compact policy, [refer to P3P specification here](#).

How-To Implement:

I287: Apache

Description

To configure Apache to send the P3P header for all pages, the `Header` directive can be used in `http.conf` as shown in the example below:

Code

```
<IfModule mod_headers.c>
Header set P3P "policyref=\"/w3c/p3p.xml\", CP=\"ALL DSP COR CURa OUR IND COM NA
</IfModule>
```

T13: Reassign new password if password is automatically generated

Priority: 4

Problem: If a system automatically generates a password, a malicious administrator may be able to retrieve the automatically-generated password. If an automatically generated password is distributed in plain text a malicious user may intercept the password.

Solution: If the application automatically generates a password, require users to change their password upon the first login.

T54: Validate file contents

Priority: 4

Problem: An application might use the file name or extension of a user-supplied file to determine the proper course of action, such as selecting the correct process to which control should be passed, deciding what data should be made available, or what resources should be allocated. If the attacker can cause the code to misclassify the supplied file, then the wrong action could occur. For example, an attacker could supply a file that ends in a ".php.gif" extension that appears to be a GIF image, but would be processed as PHP code. In extreme cases, code execution is possible, but the attacker could also cause exhaustion of resources, denial of service, information disclosure of debug or system data (including application source code), or being bound to a particular server side process. This weakness may be due to a vulnerability in any of the technologies used by the web and application servers, due to misconfiguration, or resultant from another flaw in the application itself.

Solution: Use tools to validate the type of a particular user-uploaded file, rather than relying on the extension. Attackers may try to upload malicious executable files with a valid file extension (e.g., .jpg) and then exploit a different vulnerability to execute the file. Moreover, certain attacks take advantage of the fact that some file formats use headers at the start of the file while other formats use headers anywhere in the file. The best approach is to use a tool that validates the entire contents of the file. For example, the ImageMagik library can help determine whether a file is a valid image.

T12: Mask user passwords by default

Priority: 3

Problem: The software fails to mask passwords during entry, increasing the potential for attackers to observe and capture passwords.

Solution: Do not display passwords to end-users. Use masking features to hide the passwords. If business requirements necessitate it, allow users the option of displaying the password, but mask the password by default.

How-To Implement:

I274: HTML, Web Applications

You can achieve this by setting the type of the input to "password" in HTML.

For example:

```
<input name="Password" id="Password" type="password" autocomplete="off" />
```

T47: Implement a global error handler and generic default error page for end users

Priority: 3

Problem: The software fails to return custom error pages to the user, possibly exposing sensitive information.

Solution: Use a global error handler to catch errors or exceptions not explicitly handled by the application. Always provide end-users with a generic error page that does not reveal internal system details such as a stack trace or core dump.

In some cases, the application or web server may provide a facility to catch all errors before forwarding to the end-user. In other cases, the application will need to explicitly develop a handler which catches any errors before redirecting them to the end-user. The advantage of the latter approach is that the application can log the error with a unique ID, create a generic message to the end-user, and include the unique ID in the page. This allows application support personnel to troubleshoot an issue without revealing system details to an end user.

How-To Implement:

I64: Java EE

Description

In a typical Java EE environment, set up a default Servlet or JSP to handle all incoming exceptions and to define a transaction ID for troubleshooting.

Code

In web.xml:

```
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/errorPage.jsp</location>
</error-page>
```

Then call a method from errorPage.jsp in another class that does something similar to the following:

```
public synchronized static int logException (Throwable t){
    setTransactionId(getTransactionId() + 1);
    log.error("Transaction ID: " + getTransactionId(), t);
    return getTransactionId();
}
```

[Download the Complete Code Example](#)

T51: Throw an error page for unknown extensions on the web server

Priority: 3

Problem: The software fails to return custom error pages to the user, possibly exposing sensitive information.

Solution: When the web server encounters a file type that it does not know how to handle, it should provide an error message or simply not serve the file. This helps protect against scenarios where an attacker tries to download or execute unauthorized content (e.g., a password file) from the application server as the result of a different attack, such as directory traversal.

How-To Implement:

I67: Apache 2.0

Description

Use the "Files" directive with a regular expression in your httpd.conf. First, define a generic directive that denies all file type, then add another handler that explicitly allows a specific set of file types. Note: This example is for illustrative purposes only. Your implementation will need to make provisions for all extensions used within your application.

Code

```
<Files ~ ">
order allow,deny
allow from none
deny from all
</Files>
<Files ~
    "\.(css|js|jsp|asp|aspx|do|faces|html|htm|gif|jpeg|jpg|png|swf|txt)$">
order allow,deny
Allow from env=let_me_in
</Files>
```

[Download the Complete Code Example](#)

T65: Restrict accepted HTTP verbs

Priority: 3

Problem: HTTP specifies many different methods (or verbs), most prominently GET, POST, HEAD, PUT, etc. Web applications generally use GET and POST methods exclusively for requests.

Many web applications will limit which HTTP verbs can be used to access a resource (most likely GET or POST) by defining a security constraint within web.xml and explicitly listing GET and POST within separate <http-method> tags.

The HTTP HEAD method works exactly like a GET method, with the exception that the server must not return a message body in the response [1]. Note also that the HTTP specification

intended that GET requests (along with HEAD) “should not have the significance of taking an action other than retrieval.” In other words, GET requests are meant to be idempotent – they should not change the state of the application.

Unfortunately, many applications define non-idempotent GET requests, that is, they treat GET requests just like POST requests in that they can execute transactions that change the state of the application. An example of such a URL would be:

`www.myapp.org/mgmt/accts.jsp?q=delAcct.`

Therefore, an attacker could attempt to trap the above GET request in an HTTP proxy, modify it to a HEAD request, and submit the request. Since HEAD is not one of the HTTP methods listed in `web.xml`, the authorization check would be bypassed and the code that processes GET requests may be executed. This is known as an HTTP Verb Tampering attack.

Furthermore, some frameworks such as Java EE allows the use of arbitrary HTTP verbs. An attacker can send a “BOB” requests rather than HEAD, and since it is not listed in the security constraint, the constraint will be bypassed. Arbitrary verbs will not work when accessing a Servlet, but is acceptable if the request targets a JSP directly. This is because when a JSP is compiled its components are sent to a `service()` method, rather than the `doGet()` or `doPost()` methods of a Servlet. Therefore, any arbitrary verb that is submitted will be directed to the `service()` method.

[1] <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>

Solution: Restrict the HTTP verbs which your application or web server accepts to the ones required, such as GET, POST, and HEAD.

How-To Implement:

I75: Apache 2.0

Description

Use “`SetEnvIf`” in your `httpd.conf` to limit the supported methods. Note that you can also use the “`<LimitExcept>`” directive, but that this may not be compatible with the whitelist restriction of file types shown below.

Code

```
SetEnvIf Request_Method "GET" let_me_in
SetEnvIf Request_Method "HEAD" let_me_in
```

```
SetEnvIf Request_Method "POST" let_me_in
```

```
<Files ~  
    "\.(css|js|jsp|asp|aspx|do|faces|html|htm|gif|jpeg|jpg|png|swf|txt)$">  
order allow,deny  
Allow from env=let_me_in  
</Files>
```

```
Deny from all
```

[Download the Complete Code Example](#)

T135: Provide unique user IDs for each user

Priority: 3

Problem: The product uses multiple resources that can have the same identifier, in a context in which unique identifiers are required. This could lead to operations on the wrong resource, or inconsistent operations.

Solution: Ensure that every user of the system has a unique user ID. Note: this task is generally derived from compliance requirements, such as the Payment Card Industry Data Security Standards (PCI DSS) or the Health Insurance Portability and Accountability Act (HIPAA).

T62: Lockdown passwords in property and configuration files (e.g. database connection strings)

Priority: 3

Problem: This can result in compromise of the system for which the password is used. An attacker could gain access to this file and learn the stored password or worse yet, change the password to one of their choosing.

Solution: Web applications often store plaintext system passwords and keys in configuration files. For example, several frameworks use plaintext configuration files for database connection strings, database encryption keys, Lightweight Directory Access Protocol (LDAP) connection strings, keystore passwords, and other values. Attackers who are able to exploit other

vulnerabilities are sometimes able to view the contents of files.

Always encrypt credentials in property files.

Unfortunately, the problem of providing a password or key to decrypt encrypted credentials still exists. While no solution is perfect, you may wish to employ one of several password / key storage options:

- Store a private key unique to each machine as a binary file that can only be accessed by the application server. While this control succeeds in preventing attackers from viewing plaintext passwords in configuration files, it does not prevent attackers from first accessing the binary key and then the configuration file using the same exploit. This should be the minimum security option.
- Store the decryption key / password in a file, similar to the preceding option. Use operating system controls to ensure that file is only accessible by a separate launching process – not by the application server. The launching process can then pass the key / password as a command-line argument when launching the application server. This way, a user who exploits the application server may not necessarily have access to the decryption key itself.
- Support passphrases from an environment variable and/or web-form. This solution takes more work and may necessitate manual intervention, but also greatly reduces the risk of an attacker finding plaintext passwords in configuration files.

T25: Enforce absolute session timeouts

Priority: 2

Problem: Absolute or hard session timeouts impose a maximum age on the life of a session, regardless of activity level. This control limits the potential harm an attacker can inflict upon session hijacking to a finite period. Without this control, an attacker who successfully hijacks a session can keep the session alive for as long as the server keeps track of sessions.

Solution: Enforce an absolute timeout on user sessions. An absolute timeout value of 10 hours will invalidate the session token 10 hours after its creation, regardless of activity or inactivity. Note that most frameworks do not provide this as a configuration option; developers will need to implement this control themselves.

Absolute session timeout values require a tradeoff between usability and security. Some applications, such as stock quote system for brokers, may require sessions for more than twenty-four hours. Aim to set the absolute value low enough to minimize the damage in the

event of a session hijacking, but high enough that legitimate users will not encounter it. In the event of an absolute session timeout, notify the user through an appropriate error message.

How-To Implement:

I22: Java EE

Description

Inside of a Servlet filter:

Code

```
public void doFilter(  
    ServletRequest request,  
    ServletResponse response,  
    FilterChain chain  
) throws IOException, ServletException {  
    //First check to see if this is a valid HTTP request.  
    //Throw exception if not  
    if (!(request instanceof HttpServletRequest)){  
        throw new ServletException("Invalid Request Type");  
    }  
  
    //Get Session from object.  
    //We don't want to create one if it doesn't exist yet either  
    HttpSession session = ((HttpServletRequest)request).getSession(false);  
  
    //We're not interested if the session hasn't been set yet  
    //or if the client hasn't had the session set yet  
    if (!(session==null) && !(session.isNew())) {  
        //Get the difference between time of session creation and now  
        //get the maximum length of active session  
        //according to configuration  
        long diff = System.currentTimeMillis() - session.getCreationTime();  
        long maxLength = this.getMaxActiveSessionLength() * 1000 * 60;  
  
        //If the difference in time between in creation is greater than  
        //the maximum active length then invalidate the session.  
        if (diff >= maxLength){
```

```
        session.invalidate();
    }
}

chain.doFilter(request, response);
}
```

[Download the Complete Code Example](#)

I23: SiteMinder 6

Description

Configure the "Maximum Timeout" attribute for the User Session. This needs to be configured separately for each realm. See

https://support.ca.com/cadocs/0/CA%20SiteMinder%20r6%200%20SP6-ENU/Bookshelf_Files/HTML/index.I

T44: Specify standard encoding format for all HTML content

Priority: 2

Problem: The software fails to properly handle encoding or decoding of the data, resulting in unexpected values.

Solution: Unless there is a specific reason to use a different format, specify a consistent character-encoding format such as UTF-8 to all HTTP response pages for HTML content from the application server.

How-To Implement:

I60: Java EE

Description

The following instructions demonstrate how to do this programmatically:

Code

Add the following to the top of all JSPs:

```
<%@page contentType="text/html; charset=UTF-8" %>
```

Add the following to Servlet-generated responses:

```
response.setContentType("text/html; charset=UTF-8");
```

[Download the Complete Code Example](#)

T73: Use random delays in authentication failures

Priority: 2

Problem: Two separate operations in a product require different amounts of time to complete, in a way that is observable to an actor and reveals security-relevant information about the state of the product, such as whether a particular operation was successful or not.

Solution: Malicious users may be able to guess the reason for authentication failure based on the length of time it takes to generate an authentication error message. To prevent this, use a random delay upon each authentication failure such that end users cannot ascertain the reason for failure.

T56: Restrict access to WSDL

Priority: 2

Problem: The Web services architecture may require exposing a WSDL file that contains information on the publicly accessible services and how callers of these services should interact with them (e.g. what parameters they expect and what types they return).

Solution: Web services frameworks sometimes generate a WSDL automatically and publish that WSDL to a browseable URL. Attackers can use the WSDL to gain an understanding of the web services in order to craft their attacks.

Where possible, disable automatic publication of WSDLs unless there is a legitimate business reason not to. Unfortunately, some frameworks do not provide a programmatic or declarative mechanism to suppress WSDL publishing. In those cases, use your web server's configuration to avoid serving .wsdl files

If business requirements dictate that the WSDL should be published, protect against disclosure to unauthorized users by ensuring that the file has appropriate authorization controls.

How-To Implement:

I78: Apache 2.0

Description

If your framework doesn't support the suppression of WSDL publishing, add the following directive to your httpd.conf file:

Code

```
<FilesMatch "^\.wsdl">  
    Order allow,deny  
    Deny from all  
    Satisfy All  
</FilesMatch>
```

3.2: Architecture & Design

T14: Principles of least privilege

Priority: 7

Problem: When access control checks are not applied consistently - or not at all - users are able to access data or perform actions that they should not be allowed to perform. This can lead to a wide range of problems, including information exposures, denial of service, and arbitrary code execution.

Solution: The following three principles must be followed:

1. Principle of least privilege: The users of application must be given the least amount of privilege that is required to perform the actions needed based on their role.
2. Principle of late privilege: Grant elevated privileges as late as possible in processing
3. Principle of minimum duration privilege: Drop elevated privileges as soon as possible

For example, when the currently logged in user needs to perform a high value transaction via an additional pass-phrase check:

- Re #1: The pass-phrase check should only allow for said transaction (or a preset authorized amount) and not any other privileged actions that is not in the scope (such as reversal of transaction)
- Re #2: The pass-phrase check should be made right before the transaction is submitted
- Re #3: There should be a short enough timeout such that an inactive session could not be misused

T15: Centralize authorization

Priority: 7

Problem: When access control checks are not applied consistently - or not at all - users are able to access data or perform actions that they should not be allowed to perform. This can lead to a wide range of problems, including information exposures, denial of service, and arbitrary code execution.

Solution: Centralize authorization functionality into a single module. Hard-coding authorization logic into an application complicates maintainability. For example, checking to see if a user is in a particular role within a business logic method introduces coupling between the authorization code and business logic. A change in security policy may mean a change in several business logic methods.

T18: Apply domain specific authorization checks in business logic

Priority: 7

Problem: Applications that do not perform sufficient business logic level authorization may allow malicious users to perform actions that they should not have access to. For example, a regular user may be able to access a different user's data without sufficient authorization.

Solution: For business logic transactions, explicitly ensure that users have sufficient privileges to perform a particular action. For example, an online banking application may have a constraint such as "Users cannot transfer more than \$500 from savings accounts."

While developers often perform this sort of transaction within the Model-View-Controller, business logic may later be consumed by other interfaces/channels that do not perform this sort of testing. Furthermore, embedding the check on the middle tier provides a measure of defense-in-depth, in case an attacker can circumvent MVC security controls. Note that the authorization check can still be implemented by a centralized authorization library or function.

How-To Implement:

I7: Java EE with ESAPI

Description

The ESAPI AccessController can be used for this.

Code

```
public static String doAction() throws AccessControlException{
    ESAPI.accessController().assertAuthorizedForFunction(
        "privelegedFunction"
    );
}
```

```
    return "This user can do a priveleged action!";  
}
```

[Download the Complete Code Example](#)

3rd Party Libraries

OWASP ESAPI for Java 2.0, RC7

T32: Always perform input validation on the server

Priority: 7

Problem: When software fails to validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.

Solution: Always perform input validation on the server; never rely exclusively on the client. Remember that malicious users can always bypass input validation on the client, such as JavaScript validation.

3.3: Development

T43: Avoid unsafe operating system interaction

Priority: 10

Problem: The software constructs all or part of a command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended command when it is sent to a downstream component.

Solution: To protect against command injection, avoid interacting dynamically with an operating system. When user-supplied input is necessary, determine what kind of input you would like to accept from the user and ensure you allow only those characters.

For example, if your application dynamically starts a user-specified process through shell interaction, ensure that only specific process names are allowed; reject all others character combinations.

T17: Avoid client-side authorization

Priority: 8

Problem: When the server relies on protection mechanisms placed on the client side, an attacker can modify the client-side behavior to bypass the protection mechanisms resulting in potentially unexpected interactions between the client and server. The consequences will vary, depending on what the mechanisms are trying to protect.

Solution: Never rely on client-side code, such as a JavaScript library, to authorize users. Users can always bypass client-side security controls.

Remember, users may be able to forcibly browse or otherwise guess the URL of a page with a privileged action. Always check whether a user is allowed to perform an action on the server itself.

T33: Verify integrity of client-supplied read-only data

Priority: 8

Problem: If a web product does not properly protect assumed-immutable values from modification in hidden form fields, parameters, cookies, or URLs, this can lead to modification of critical data. Web and client-server applications often mistakenly make the assumption that data passed to the client in hidden fields or cookies is not susceptible to tampering. Failure to validate portions of data that are user-controllable can lead to the application processing incorrect, and often malicious, input. For example, custom cookies commonly store session data or persistent data across sessions. This kind of session data is normally involved in security related decisions on the server side, such as user authentication and access control. Thus, the cookies might contain sensitive data such as user credentials and privileges. This is a dangerous practice, as it can often lead to improper reliance on the value of the client-provided cookie by the server side application.

Solution: In some cases, end-users should not be able to modify certain parameters, such as some hidden form fields, in web applications. For example, several web-based shopping cart applications use a client-supplied price parameter to calculate total cost. Avoid trusting client-supplied data for these cases.

If you must rely on client-supplied data for business or architectural reasons, use a server-side mechanism that detects tampering. One example is to use a Hashed Message Authentication Code (HMAC), which does the following:

- Takes a hash of read-only fields in a form prior to sending them to the client
- Encrypts that hash with a secret key stored on the server
- Adds the hashed and encrypted value as an additional hidden field in the form
- Upon form submission, rehashes and re-encrypts the read-only client-supplied parameters and compares the hash with the client-supplied HMAC parameter. Any difference indicates that one or more of the read-only parameters were tampered.

T36: Escape untrusted data in HTML, HTML attributes, Cascading Style Sheets and JavaScript

Priority: 8

Problem: Cross-site scripting (XSS) vulnerabilities occur when the software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output.

Once the malicious script is injected, the attacker can perform a variety of malicious activities. The attacker could transfer private information, such as cookies that may include session information, from the victim's machine to the attacker.

The attacker could send malicious requests to a web site on behalf of the victim, which could be especially dangerous to the site if the victim has administrator privileges to manage that site. Phishing attacks could be used to emulate trusted web sites and trick the victim into entering a password, allowing the attacker to compromise the victim's account on that web site.

Finally, the script could exploit a vulnerability in the web browser itself possibly taking over the victim's machine, sometimes referred to as "drive-by hacking." In many cases, the attack can be launched without the victim even being aware of it. Even with careful users, attackers frequently use a variety of methods to encode the malicious portion of the attack, such as URL encoding or Unicode, so the request looks less suspicious.

Solution: Always escape untrusted data such as user-supplied HTTP parameters before printing them onto a web page. For example, HTML entities encode dangerous characters that are included inside of HTML. Doing so will decrease the risk of a Cross-Site Scripting (XSS) vulnerability.

Note that any form of output on a web page may be vulnerable to script injection. The rules for encoding the data will differ by the form of output. In most cases, web applications use untrusted data within HTML; however, your application may dynamically generate JavaScript, Cascading Style Sheets (CSS), or Adobe Flash data based on untrusted data. The rules on how to escape characters differ by format.

The best approach to escaping is to use a white-list: escape all characters unless you explicitly know them to be safe (such as Unicode letters and numbers). However, in some cases this approach may be infeasible. At a minimum, encode known special characters for each output format.

You can review existing encoding implementations as a reference. For example, the OWASP Enterprise Security API provides implementation of the encoder interface, which enables encoding data in several formats, including, HTML, HTML attributes, CSS and JavaScript.

How-To Implement:

I43: Java EE with Tag Libraries

Description

Tag library output may be vulnerable to XSS, depending on several factors: 1) If HTML/XML escaping for tags that output data to web context is not turned on by default 2) Even if HTML escaping is turned on, it may be insufficient to protect against XSS when the tag output is within the context of an HTML attribute, JavaScript, or CSS. In these cases, proper contextual escaping using the appropriate contextual encoding is required. ESAPI provides Expression Language (EL) contextual escaping functions for tag libraries. Use the function depending on the context of your output.

Code

```
<c:set
    var="dangerous"
    scope="request"
    value="<%= request.getParameter(\"test\") %>" />
<p></p>
<p>
    Test using ESAPI tag libraries with JSTL in HTML:
    ${esapi:encodeForHTML(dangerous)}
</p>
<p>
    Test using ESAPI tag libraries with JSTL in HTML Attribute:
    <input type="text" value="${esapi:encodeForHTMLAttribute(dangerous)}" />
</p>
<p>
    Test using ESAPI tag libraries with JSTL in JavaScript:
    <script language="javascript">
        var str=${esapi:encodeForJavaScript(dangerous)};
    </script>
</p>
```

[Download the Complete Code Example](#)

3rd Party Libraries

OWASP ESAPI for Java 2.0, RC7

T37: Avoid DOM-based cross site scripting (XSS)

Priority: 8

Problem: Cross-site scripting (XSS) vulnerabilities occur when the software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output.

Once the malicious script is injected, the attacker can perform a variety of malicious activities. The attacker could transfer private information, such as cookies that may include session information, from the victim's machine to the attacker.

The attacker could send malicious requests to a web site on behalf of the victim, which could be especially dangerous to the site if the victim has administrator privileges to manage that site. Phishing attacks could be used to emulate trusted web sites and trick the victim into entering a password, allowing the attacker to compromise the victim's account on that web site.

Finally, the script could exploit a vulnerability in the web browser itself possibly taking over the victim's machine, sometimes referred to as "drive-by hacking." In many cases, the attack can be launched without the victim even being aware of it. Even with careful users, attackers frequently use a variety of methods to encode the malicious portion of the attack, such as URL encoding or Unicode, so the request looks less suspicious.

Solution: You need to be aware that some of the DOM properties can be controlled by the user and might include unsafe values. An example is `document.location`. If this property is used in your dynamic JavaScript code to create HTML content in an unsafe manner (e.g. assigned to `element.innerHTML`), it can allow attackers to inject script by creating malicious links and essentially leading to a cross-site scripting vulnerability.

To prevent a DOM-based XSS, you need to:

- Treat the user-controlled DOM values as unsafe.

Some of the DOM properties that may be manipulated for XSS include (note that these properties might have sub-attributes such as `location.hash` that are considered unsafe as well):

```
document.URL
document.URLUnencoded
document.location (and many of its properties)
document.referrer
window.location (and many of its properties)
location (note that window.location can be access as just location)
```

Where possible use server side to validate and/or properly encode these values and avoid using them directly

Validate these values against a stringent whitelist before using

Use dynamic build interfaces and avoid using rendering these rendering methods along with the unsafe variables.

Some examples of safe dynamic build interfaces are:

```
document.createElement("...")
element.setAttributes("...", "value")
element.appendChild(...)
```

Some examples of unsafe rendering methods are:

```
element.innerHTML = "...";
element.outerHTML = "...";
document.write(...);
document.writeln(...);
```

Use client side encoding helper functions (the function name might vary based on the library/framework being used. See the How-Tos for more info) to encode the unsafe values for display purposes.

Be aware of the numerous methods which implicitly eval() data passed to it and make sure the data is properly sanitized before passing to these functions.

How-To Implement:

I50: JavaScript

Description

Some of the JavaScript functions that may result in writing Script into the page include:

```
document.write(...)
document.writeln(...)
document.body.innerHTML=...
document.forms[0].action=... (and various other collections)
document.attachEvent(...)
document.create...(...)
document.execCommand(...)
document.body. ... (accessing the DOM through the body object)
```



```
window.attachEvent(...)
document.location=... (and assigning to location's href, host and hostname)
document.location.hostname=...
document.location.replace(...)
document.location.assign(...)
document.URL=...
window.navigate(...)
document.open(...)
window.open(...)
window.location.href=...(and assigning to location's href, host and hostname)
eval(...)
window.execScript(...)
window.setInterval(...)
window.setTimeout(...)
```

If you are not using any platforms or libraries that provide HTML escaping functionality, you can use this helper function:

```
function escapeHTML(str) {
    str = str + "";
    var out = "";
    for(var i=0; i<str.length; i++) {
        if(str[i] === '<') {
            out += '&lt;';
        } else if(str[i] === '>') {
            out += '&gt;';
        } else if(str[i] === '"') {
            out += '&#39;';
        } else if(str[i] === "'") {
            out += '&quot;';
        } else {
            out += str[i];
        }
    }
    return out;
}
```

I252: JavaScript with ESAPI

Description

You need to safely encode all the unsafe values derived from user controllable DOM variables. Using ESAPI JavaScript library, you can use either the `encodeURIComponent` method for when the target is treated as a URL or `encodeForHTML` method when the target string is used for display/rendering purposes.

The Following demonstrates the usages of the ESAPI library encoding functionality for dynamic content.

Code

```
var input = window.location.pathname + '#test';
//URL encoding is happening in JavaScript
window.location = ESAPI4JS.encodeForURL(input);
//HTML encoding is happening in JavaScript
document.writeln(ESAPI4JS.encodeForHTML(input));
```

T48: HTML entity encode validation error messages

Priority: 8

Problem: Cross-site scripting (XSS) vulnerabilities occur when the software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output.

Once the malicious script is injected, the attacker can perform a variety of malicious activities. The attacker could transfer private information, such as cookies that may include session information, from the victim's machine to the attacker.

The attacker could send malicious requests to a web site on behalf of the victim, which could be especially dangerous to the site if the victim has administrator privileges to manage that site. Phishing attacks could be used to emulate trusted web sites and trick the victim into entering a password, allowing the attacker to compromise the victim's account on that web site.

Finally, the script could exploit a vulnerability in the web browser itself possibly taking over the victim's machine, sometimes referred to as "drive-by hacking." In many cases, the attack can be launched without the victim even being aware of it. Even with careful users, attackers frequently use a variety of methods to encode the malicious portion of the attack, such as URL encoding or Unicode, so the request looks less suspicious.

Solution: If it is possible for user-controllable data to be concatenated to an error message, encode the HTML message before storing the message in a log file or displaying the exception

on a webpage. This will prevent the application from exposing Cross-Site Scripting vulnerabilities.

Note that exception messages should never be displayed to an end-user. However, exception messages can be displayed to an administrator or developer through a web-based administrative console.

T72: Use safe arithmetic to avoid integer overflow

Priority: 8

Problem: An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may wrap to become a very small or negative number. While this may be intended behavior in circumstances that rely on wrapping, it can have security consequences if the wrap is unexpected. This is especially the case if the integer overflow can be triggered using user-supplied inputs. This becomes security-critical when the result is used to control looping, make a security decision, or determine the offset or size in behaviors such as memory allocation, copying, concatenation, etc.

Solution: Arithmetic operations on numeric primitives such as integers, shorts, and longs may be vulnerable to overflow. For example, if you add two large positive integers the result may be negative. The inverse is true for negative numbers. Other operations such as subtraction, multiplication, division, and exponents can be vulnerable as well. This is of particular concern if you use at least one un-trusted number in an operation, such as a user-supplied number.

In order to avoid overflow, you can cast one of the operands or the result to a primitive with more bits. For example, you can cast a short to an int or an int to a long in most languages. Alternatively, you can use logical checks to test for overflow. Assuming o1 is operand 1, o2 is operand 2, and r is result, then you use the following checks:

- for $o1 * o2 = r$, check that $r / o2 = o1$
- for $o1 ^ o2 = r$, check that $\log r \text{ base } o2 = o1$
- for $o1 + o2 = r$, check that $r > 0$
- for $o1 + o2 = r$, if o1 & o2 are both positive then check that $r > 0$, if o1 & o2 are both negative then check that $r < 0$

How-To Implement:

I271: Java

Java primitive types, including `int` and `long` are susceptible to integer wrapping (underflow/overflow). Arithmetic operations performed on the primitive types will not raise an exception in case of integer wrapping.

There are several approaches available to mitigate the risk of integer wrapping. The best choice depends on the usage in each case and might vary in different modules/apps:

Precondition testing: Check the inputs to each arithmetic operator to ensure that overflow cannot occur. Throw an `ArithmeticException` when the operation would overflow if it were performed; otherwise, perform the operation. For example, for an application that input values are supposed to be in percentage form, validate that x and y are both between 0 to 100 so the maximum result of a multiplication will be in range of 0 to 10000 and will fit an `int`, resulting in safe multiplication.

Upcasting: Cast the inputs to the next larger primitive integer type and perform the arithmetic in the larger size. Check each intermediate result for overflow of the original smaller type and throw an `ArithmeticException` if the range check fails. Note that the range check must be performed after each arithmetic operation; larger expressions without per-operation bounds checking can overflow the larger type. Downcast the final result to the original smaller type before assigning to a variable of the original smaller type. This approach cannot be used for type `long` because `long` is already the largest primitive integer type.

BigInteger: Convert the inputs into objects of type `BigInteger` and perform all arithmetic using `BigInteger` methods. Type `BigInteger` is the standard arbitrary-precision integer type provided by the Java standard libraries. The arithmetic operations implemented as methods of this type cannot overflow; instead, they produce the numerically correct result. Consequently, compliant code performs only a single range check just before converting the final result to the original smaller type and throws an `ArithmeticException` if the final result is outside the range of the original smaller type.

Reverse Arithmetic: Verify the sanity of the result by performing the reverse of the arithmetic procedure. For example, if we are multiplying x by y , to get the result z , then we can check if z divided by y is equal to x .

The *precondition testing* technique requires different precondition tests for each arithmetic operation. This can be somewhat more difficult to implement and to audit than either of the other two approaches.

The *upcast* technique is the preferred approach when applicable. The checks it requires are simpler than those of the previous technique; it is substantially more efficient than using `BigInteger`. Unfortunately, it cannot be applied to operations involving type `long`, as there is no bigger type to upcast to.

The *BigInteger* technique is conceptually the simplest of the three techniques because arithmetic operations on `BigInteger` cannot overflow. However, it requires the use of method calls for each operation in place of primitive arithmetic operators, which can obscure the intended meaning of the code. Operations on objects of type `BigInteger` can also be significantly less efficient than operations on the original primitive integer type.

T184: Perform authorization checks on RESTful web services

Priority: 8

Problem: RESTful web service requests can support several HTTP methods, such as GET, POST, HEAD, Put and DELETE. The specific method use can alter the behavior of the web service. For example, a call to DELETE `/exampleurl/users/bobsmith` might result in deleting the user Bob Smith. Without sufficient authorization controls, a malicious user may be able to perform an action that normally requires elevated privileges (e.g. deleting a user).

Solution: Ensure that the user accessing the web service has sufficient permission to access the URL AND use the HTTP verb. For example, a user may have access to GET a resource but may not have permission to DELETE or PUT the same resource.

T11: Disallow external redirects

Priority: 7

Problem: An http parameter may contain a URL value and could cause the web application to redirect the request to the specified URL. By modifying the URL value to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials. Because the server name in the modified link is identical to the original site, phishing attempts have a more trustworthy appearance.

Solution: Do not allow users to specify URLs which redirect to external sites. In some cases, there may be a legitimate reason to allow this functionality for specific sites. For example, an online payment processor may wish to redirect users back to the site that requested payment. In these cases, validate the URL against a whitelist of known, trusted URLs.

T16: Authorize every page

Priority: 7

Problem: Web applications susceptible to direct request attacks often make the false assumption that such resources can only be reached through a given navigation path and so only apply authorization at certain points in the path.

Solution: Implement an explicit authorization check on every non-public dynamic server page to ensure that only authorized users can access that page.

T29: Use anti cross site request forgery (CSRF) tokens

Priority: 7

Problem: When a web server is designed to receive a request from a client without any mechanism for verifying that it was intentionally sent, then it might be possible for an attacker to trick a client into making an unintentional request to the web server which will be treated as an authentic request. This can be done via a URL, image load, XMLHttpRequest, etc. and can result in data disclosure or unintended code execution.

Solution: An anti-CSRF token is a session-specific or even transaction-specific random string appended as a parameter to important transactions (e.g., purchasing a stock on a brokerage site). Upon handling the client's request, the server ensures that the CSRF token is the value expected for that session/transaction. If the token is not correct, then the application denies the transaction. This helps protect against CSRF because each request will have at least one unique parameter that an attacker cannot know ahead of time.

Note that you may be able to mitigate the risk of CSRF by using an alternative user-specific token, such as the `userId`, rather than a specific anti-CSRF token.

How-To Implement:

I30: Java EE

Description

The following code demonstrates the use of a JSP tag to generate a simple CSRF token in a form, as well as the use of another tag to check for the existence of the CSRF token upon form submission.

Code

If you've already created a reusable library for CSRF tags then you simply have to enter the `csrftoken` tag in a JSP with a form:

```
<form method="POST" action="success.jsp">
<p>Name: <input type="text" name="name"></p>
<p>Title: <input type="text" name="title"></p>
<csrf:csrftoken/>
<input type="submit" value="Submit" name="button" />
</form>
```

Then you can check for the presence of the token in the page that handles form submission:

```
<body>
<csrf:csrfcheck/>
<h1>Success!</h1>
</body>
```

If you haven't already defined the tags then you can add the following:

First, define a new `bodytagssupport` class to handle the generate CSRF token tag:

```
public class AntiCSRFTokenTag extends BodyTagSupport {
public String getToken () throws JspTagException {
    try {
        HttpSession session = ((HttpServletRequest)this.pageContext.
            getRequest()).getSession(false);
        String token = CSRFTokenUtil.getToken(session);
        return token;
    } catch (Exception e) {
        throw new JspTagException(e);
    }
}
```

```

}

public int doStartTag() throws JspTagException
{
    JspWriter out;
    out = this.pageContext.getOut();
    try
    {
        out.print("<input type='hidden' name='"
            + CSRFTokenUtil.SESSION_ATTR_KEY +
            "' value='" + getToken() + "' />");
    }
    catch (Exception e)
    {
        throw new JspTagException(
            "Error writing to body's enclosing JspWriter",e
        );
    }
    return SKIP_BODY;
}
}

```

Next, create a class that implements checking for the CSRF Token:

```

public class AntiCSRFTokenTagCheck extends BodyTagSupport {
    public int doStartTag() throws JspTagException
    {
        try {
            if (!CSRFTokenUtil.isValid((HttpServletRequest)
                (pageContext.getRequest()))){
                throw new CSRFTokenException("CSRF Attempt!");
            }
        } catch (NoSuchAlgorithmException e) {
            throw new JspTagException(e);
        } catch (ServletException e) {
            throw new JspTagException(e);
        }

        return SKIP_BODY;
    }
}

```


Next you'll need to define the tag library in a TLD:

```
<taglib
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
    version="2.0">
    <description>
        Library used for generating and checking Anti-CSRF
        tokens
    </description>
    <display-name>Anti CSRF Tags</display-name>
    <tlib-version>2.0</tlib-version>
    <short-name>csrf</short-name>
    <uri>
        http://labs.securitycompass.com/code/anti_csrf
    </uri>
    <tag>
        <name>csrftoken</name>
    <tag-class>
        com.securitycompass.examples.sessionManagement.anticsrf.AntiCSRFTokenTag
    </tag-class>
        <body-content>empty</body-content>
    </tag>

    <tag>
        <description>
            Checks the csrf token in the session against
            the one in the request. If the two
            don't match then generates a JspTag exception
        </description>
        <display-name>Anti CSRF Token Check</display-name>
        <name>csrfcheck</name>
    <tag-class>
        com.securitycompass.examples.sessionManagement.anticsrf.AntiCSRFTokenTagCheck
    </tag-class>
        <body-content>empty</body-content>
```

```
        </tag>
</taglib>
```

Here's a simple CSRFTokenUtil class based on OWASP's earlier version of the CSRFGuard:

```
public final class CSRFTokenUtil
{
    //algorithm to generate key
    private final static String DEFAULT_PRNG = "SHA1PRNG";

    public final static String SESSION_ATTR_KEY = "CSRF_TOKEN";
    private final static String NO_SESSION_ERROR = "No valid session found";

    private static String getToken() throws NoSuchAlgorithmException
    {
        return getToken(DEFAULT_PRNG);
    }

    private static String getToken(String prng) throws NoSuchAlgorithmException
    {
        SecureRandom sr = SecureRandom.getInstance(prng);
        return "" + sr.nextLong();
    }

    public static String getToken (HttpSession session)
        throws ServletException, NoSuchAlgorithmException {
        //throw exception if session is null
        if (session == null) {
            throw new ServletException(NO_SESSION_ERROR);
        }

        //Now attempt to retrieve existing token from session.
        //If it doesn't exist then add it
        String token_val = (String)session.getAttribute(SESSION_ATTR_KEY);
        if (token_val == null){
            token_val = getToken();
            session.setAttribute(SESSION_ATTR_KEY, token_val);
        }
        return token_val;
    }
}
```

```

}

public static boolean isValid (HttpServletRequest request)
throws ServletException, NoSuchAlgorithmException {
    //throw exception if session is null
    if (request.getSession(false)== null) {
        throw new ServletException(NO_SESSION_ERROR);
    }
    return getToken(request.getSession(false)).equals(
        request.getParameter(SESSION_ATTR_KEY));
}
}

```

[Download the Complete Code Example](#)

T31: Perform input validation on all forms of input

Priority: 7

Problem: When software fails to validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.

Solution: Consistent input validation decreases the risk of many web application attacks, including Cross-Site Scripting. Explicitly validate the characters for all user input from end-users, including HTTP parameter names and values, cookies, or any other exposed interface (e.g., remote method invocation). Where possible, use whitelist validation (i.e., only allow a specific set of characters and disallow all others). Where creating a whitelist is impossible or infeasible, use a blacklist approach by filtering out known dangerous characters. In either case, ensure you also perform output encoding (covered in separate standards) to reduce the risk of injection attacks.

How-To Implement:

I32: Java

Description

In any Java application, you can use Java Regular Expressions to validate input, such as the following:

Code

```
private static String user_parameter_regex = "^[a-zA-Z0-9\\-_]{1,50}$";

public static void main(String[] args) {
    if (args.length < 1){
        System.out.println(
            "Usage: java -jar java.input_validation.jar user_parameter"
        );

        return;
    }
    if (OSInteraction.runExternalCommand(args[0])) {
        System.out.println(args[0]);
        System.out.println("Worked!");
    }
    else {
        System.out.println("Didn't work");
    }
}

public static boolean runExternalCommand (String user_parameter){

    try {
        Runtime rt = Runtime.getRuntime();
        if(Pattern.matches(user_parameter_regex, user_parameter)){
            rt.exec("c:\\Python26\\python fake_script " + user_parameter);
            return true;
        }
        else {
            return false;
        }
    }catch (IOException e){
        return false;
    }
}
```

```
}  
}
```

[Download the Complete Code Example](#)

T34: Disallow overly-long, malformed, and non-printable characters unless specifically required

Priority: 7

Problem: When software fails to validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.

Solution: Overly long and malformed characters in variable-length encoding formats such as UTF-8 can be used to bypass filters. These characters may sometimes be translated to the proper format after sanitization by a different component or application. Accept legal character sequences only.

The W3C provides a Perl regular expression to validate printable UTF-8 characters:

```
$field =~  
m/\A(  
    [\x09\x0A\x0D\x20-\x7E]          # ASCII  
    | [\xC2-\xDF][\x80-\xBF]        # non-overlong 2-byte  
    |  \xE0[\xA0-\xBF][\x80-\xBF]   # excluding overlongs  
    | [\xE1-\xEC\xEE\xEF][\x80-\xBF]{2} # straight 3-byte  
    |  \xED[\x80-\x9F][\x80-\xBF]   # excluding surrogates  
    |  \xF0[\x90-\xBF][\x80-\xBF]{2} # planes 1-3  
    | [\xF1-\xF3][\x80-\xBF]{3}     # planes 4-15  
    |  \xF4[\x80-\x8F][\x80-\xBF]{2} # plane 16  
)*\z/x;
```

This check will return true if \$field is UTF-8 and false otherwise.

How-To Implement:

I39: Java

Description

Depending on the vendor, the treatment of illegal byte sequences in Java Virtual Machines (JVMs) can vary. The Sun JVM converts illegal UTF-8 sequences into the generic \uFFFD Unicode character. However, other JVMs may ignore illegal byte sequences or convert the sequences into legal code points. See, for example, the following ticket for IBM JVMs: <http://www-01.ibm.com/support/docview.wss?uid=swg1IZ80870>. If you do not use a Sun JVM, check with your vendor to ensure that illegal byte sequences are processed the same way as the Sun JVM.

T166: Use secure JSON format for AJAX end points

Priority: 7

Problem: JSON Hijacking takes advantage of a feature in some browsers that allow script to override the core language's object setter routines, which allows malicious JavaScript to interject logic that allows it to monitor JSON messages returned from a server.

In a JSON Hijacking attack, the attacker overrides the JavaScript array constructor to steal information passed to it. The attacker then embeds the JSON endpoint of the domain being hijacked in an HTML script tag. The browser will treat the cross domain resource as a JavaScript file and tries to render it, resulting in a call to the attacker's version of the array constructor, leading to disclosure of the information across domains.

Solution: To prevent JSON Hijacking:

- Use **application/json** MIME type for all returned JSON responses. (Avoid using text/html)
- Do not use arrays as the highest level JSON structure. Use objects(dictionaries) instead. For example, if you need to return '[1,2,3]', return '{"result":[1,2,3]}'
- If your client side library allows it (e.g. Dojo), use Comment Filtered JSON format and use text/json-comment-filtered for the mime-type. For example, the '[1,2,3]' json return gets converted to '/*[1,2,3]*/' in this form

T61: Change all default passwords in third party libraries

Priority: 6

Problem: Third party systems such as servers and libraries may have default passwords associated with them. Users may be able to use these default passwords to authenticate into systems.

Solution: Some application frameworks and other third-party code contains configuration files with default passwords. As these files aren't covered in a typical hardening process, administrators may miss changing those passwords prior to deployment. Ensure that all of your libraries have all default passwords changed to strong alternatives.

T66: Frame busting

Priority: 6

Problem: Clickjacking occurs when an attacker lures a victim to a malicious site. The site overlays a transparent frame of a different site on top without the attacker's knowledge. The victim clicks on something which appears to be part of the attacker controlled site but is actually part of the other site. For example, an attacker has a site with a game in which the user has to click on a button. When the user clicks on the button, they're actually clicking on "transfer funds" from their online banking site without realizing it.

Solution: Unless you must have your application as an iframe to meet a requirement, there are two general solutions to prevent your website to be loaded as an iframe from within a third-party web-site:

1. Ensure that your application has JavaScript that automatically detects and moves out of a frame.
2. Use the `X-Frame-Options` HTTP header in responses to communicate to the browser that you do not wish the web pages to be loaded inside a third-party iframe.

There are two possible values for X-Frame-Options:

- `DENY`: The page cannot be displayed in a frame, regardless of the site attempting to do so
- `SAMEORIGIN`: The page can only be displayed in a frame on the same origin as the page itself.

Either of these values will achieve the desired result. Note that this header is supported in newer versions of the browser only. See below for a list of versions that support this tag:

Browser	Lowest version
Firefox (Gecko)	3.6.9 (1.9.2.9)

Opera	10.50
Safari	4.0
Chrome	4.1.249.1042

How-To Implement:

I76: Javascript

Description

Use the following JavaScript to remove the application from an iframe:

Code

```
<!-- Just below you <body> tag in HTML -->
<script>
if (self == top) {
    var theBody = document.getElementsByTagName('body')[0];
    theBody.style.display = "block";
} else {
    top.location = self.location;
}
</script>
```

[Download the Complete Code Example](#)

I286: Apache

Description

To configure Apache to send the X-Frame-Options header for all pages, the Header directive can be used in http.conf as shown in the example below:

Code

```
<IfModule mod_headers.c>
Header always append X-Frame-Options SAMEORIGIN
</IfModule>
```


T75: Use regular expressions that are not vulnerable to Denial of Service

Priority: 6

Problem: Improperly crafted regular expressions may lead to denial of service conditions. In particular, large payloads may lead to exponential complexity which can allow an attacker to cause a denial of service with relatively little input. See [OWASP Regular Expression DoS](#) for more details.

Solution: Some regular expressions may be vulnerable to Denial of Service. In particular, avoid the following:

- Grouping with repetition
- Inside the repeated group:
- Repetition
- Alternation with overlapping

Examples of Malicious Patterns:

- (a+)+
- ([a-zA-Z]+)*
- (a|aa)+
- (a|a?)+
- (.*a){x} | for x > 10

These examples are from [OWASP Regular Expression DoS](#).

T76: Do not hard code passwords

Priority: 5

Problem: A hard-coded password typically leads to a significant authentication failure that can be difficult for the system administrator to detect. Once detected, it can be difficult to fix, so the administrator may be forced into disabling the product entirely. There are two main variations: In the Inbound variant, a default administration account is created, and a simple password is hard-coded into the product and associated with that account. This hard-coded password is the same for each installation of the product, and it usually cannot be changed or disabled by

system administrators without manually modifying the program, or otherwise patching the software. If the password is ever discovered or published (a common occurrence on the Internet), then anybody with knowledge of this password can access the product. Finally, since all installations of the software will have the same password, even across different organizations, this enables massive attacks such as worms to take place. The Outbound variant applies to front-end systems that authenticate with a back-end service. The back-end service may require a fixed password which can be easily discovered. The programmer may simply hard-code those back-end credentials into the front-end software. Any user of that program may be able to extract the password. Client-side systems with hard-coded passwords pose even more of a threat, since the extraction of a password from a binary is usually very simple.

Solution: Avoid hard-coding passwords anywhere in the application. Use configuration files with encrypted passwords instead.

T63: Disable auto-complete on confidential fields

Priority: 4

Problem: The autocomplete function of most browsers may cache sensitive information, such as credit card numbers. The cached data from autocomplete may be accessed by other users on the same computer.

Solution: The auto-complete feature on web browsers may reveal confidential information on shared systems, such as usernames or credit card numbers. Explicitly disable this feature on confidential form fields.

Use `autocomplete=off` to do this on HTML forms:

```
<form method="POST" action="authenticate">
  <input type="text" id="val" autocomplete="off" />
</form>
```

Note that alternatively `autocomplete=off` can be specified in the `<form>` tag. All child `<input>` tags, will inherit this attribute.

```
<form method="POST" action="authenticate" autocomplete="off">
  <input type="text" id="val" />
</form>
```

T40: Use XML encoding when interacting with XML data

Priority: 4

Problem: Within XML, special elements could include reserved words or characters such as "<", ">", "'", and "&", which could then be used to add new data or modify XML syntax.

Solution: To mitigate the risk of injection, XML encodes untrusted data when dynamically creating XML. Perform explicit encoding for other XML formats such as:

- XPath
- Document Type Definition (DTD)
- XML Stylesheets
- Preprocessing tags

As with HTML, use a whitelist of characters which you know to be safe (e.g. alphanumeric) and XML-encode all other characters. If this approach is not feasible, consider XML-encoding known special characters.

T39: Disallow carriage returns and line feeds when adding data to HTTP response headers

Priority: 3

Problem: The software uses CRLF (carriage return line feeds) as a special element, e.g. to separate lines or records, but it does not neutralize or incorrectly neutralizes CRLF sequences from inputs.

Solution: Allowing carriage returns or line feeds inside the content of the HTTP response headers, such as in the HTTP header value, without proper encoding may result in HTTP response-splitting. Remember to disallow or properly encode carriage returns or line feeds (i.e. %0A and %0D in URL encoding) when dynamically creating HTTP response headers such as:

- URLs HTTP redirects
- Cookie names or values

Note that when stripping a potentially malicious character, ensure the resulting string is also free from dangerous characters. For example %0A0A would still result in a URL-encoded newline

character if the %0A was stripped out once.

T45: Log potential application security events

Priority: 3

Problem: When security-critical events are not logged properly, such as a failed login attempt, this can make malicious behavior more difficult to detect and may hinder forensic analysis after an attack succeeds.

Solution: Create a security-specific log to detect malicious activity. The OWASP AppSensor project defines a list of detection points. At a minimum, consider logging the following high-risk events defined by AppSensor:

- AE2 Multiple Failed Passwords
- IE2 Violations Of Implemented White Lists
- ACE2 Modifying Parameters Within A POST For Direct Object Access Attempts
- ACE3 Force Browsing Attempts
- ACE4 Evading Presentation Access Control Through Custom Posts

If possible, consider logging the following detection points:

- RE1 Unexpected HTTP Commands
- RE2 Attempts To Invoke Unsupported HTTP Methods
- RE3 GET When Expecting POST
- RE4 POST When Expecting GET
- AE1 Use Of Multiple Usernames
- AE3 High Rate Of Login Attempts
- AE4 Unexpected Quantity Of Characters In Username
- AE5 Unexpected Quantity Of Characters In Password
- AE6 Unexpected Types Of Characters In Username
- AE7 Unexpected Types Of Characters In Password
- AE8 Providing Only The Username
- AE9 Providing Only The Password
- AE10 Adding Additional POST Variables
- AE11 Removing POST Variables
- SE1 Modifying Existing Cookies

- SE2 Adding New Cookies
- SE3 Deleting Existing Cookies
- SE4 Substituting Another User's Valid Session ID Or Cookie
- SE5 Source IP Address Changes During Session
- SE6 Change Of User Agent Mid Session
- ACE1 Modifying URL Arguments Within A GET For Direct Object Access Attempts
- IE1 Cross Site Scripting Attempt
- EE1 Double Encoded Characters
- EE2 Unexpected Encoding Used
- CIE1 Blacklist Inspection For Common SQL Injection Values
- CIE2 Detect Abnormal Quantity Of Returned Records.
- CIE3 Null Byte Character In File Request
- CIE4 Carriage Return Or Line Feed Character In File Request
- FIO1 Detect Large Individual Files
- FIO2 Detect Large Number Of File Uploads
- UT1 Irregular Use Of Application
- UT2 Speed Of Application Use
- UT3 Frequency Of Site Use
- UT4 Frequency Of Feature Use
- STE1 High Number Of Logouts Across The Site
- STE2 High Number Of Logins Across The Site
- STE3 High Number Of Same Transaction Across The Site

T64: No-cache for confidential pages

Priority: 3

Problem: For each web page, the application should have an appropriate caching policy specifying the extent to which the page and its form fields should be cached.

Solution: Set the "no-store" and "no-cache" provisions for HTTP headers to be enabled for all confidential pages. This prevents users' browsers from caching confidential data, which another user on the same computer may view later.

Note that for HTTP 1.1, the important keyword is "no-store". A common misuse is to use only the "no-cache" directive which according to the RFC, tells the browser that it should revalidate

with the server before serving the page from the cache. Hence, if the "no-store" directive is missing and the "no-cache" directive is used, the browser, can still store the page in its cache, but would display it only after revalidating with the server, resulting in unwanted storage and leakage of confidential data.

Although in practice, newer versions of IE and Firefox have started treating the no-cache directive as if it instructs the browser not to even cache the page (due to widespread misuse of no-cache for no-store), it is strongly advised that the "no-store" directive be specified.

How-To Implement:

I74: Java EE

Description

Call the following function from your JSP or Servlet:

Code

```
public void setNoCacheHeaders(HttpServletResponse response) {  
    // HTTP 1.1  
    response.setHeader(  
        "Cache-Control",  
        "no-store, no-cache, must-revalidate"  
    );  
    // HTTP 1.0  
    response.setHeader("Pragma", "no-cache");  
    response.setDateHeader("Expires", -1);  
}
```

[Download the Complete Code Example](#)

T71: Capture sufficient information for transactional audit logging

Priority: 3

Problem: When security-critical events are not logged properly, such as a failed login attempt, this can make malicious behavior more difficult to detect and may hinder forensic analysis after

an attack succeeds.

Solution: Maintain audit logs for each full end-user transaction in the system. At a minimum, include the following:

- User ID
- Timestamp - including milliseconds
- Source IP
- Description of event
- Error codes (if applicable)

Note: Use internal system clock to generate timestamps.

3.4: Testing

T87: Test password and session ID are sent over SSL

Priority: 10

Problem: 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.

Solution: Use an HTTP-proxy tool to ensure that the password is sent over HTTPS, not HTTP. If not, then the test fails. Using an HTTP-proxy tool, browse through each page in the site and look for any requests sent using HTTP rather than HTTPS. If any of the HTTP requests transmit the session ID, then the test fails.

T81: Test account logout

Priority: 9

Problem: The software does not implement sufficient measures to prevent multiple failed authentication attempts within in a short time frame, making it more susceptible to brute force attacks.

Solution: Attempt to log in with a valid username and an invalid password five times in quick succession. If you are not temporarily locked out and you do not encounter an anti-automation control such as a CAPTCHA image, then the test fails.

How-To Implement:

I131: Manually with browser

1. Open your web browser and navigate to the login page.
2. Attempt to authenticate with a valid username and incorrect password. Repeat this 5 times.
3. If you are able to log in with the correct password on the 6th try without having to wait or solve a CAPTCHA (or other technique that might stop a script from executing), then this test fails.

T106: Test that site is not vulnerable to direct object access attacks

Priority: 9

Problem: Retrieval of a user record occurs in the system based on some key value that is under user control. The key would typically identify a user related record stored in the system and would be used to lookup that record for presentation to the user. It is likely that an attacker would have to be an authenticated user in the system. However, the authorization process would not properly check the data access operation to ensure that the authenticated user performing the operation has sufficient entitlements to perform the requested data access, hence bypassing any other authorization checks present in the system. One manifestation of this weakness would be if a system used sequential or otherwise easily guessable session ids that would allow one user to easily switch to another user's session and view/modify their data.

Solution: Look for all parts of the web application that provide access to files on the application or on other servers, excluding static files normally served by a web server (e.g., HTML, JavaScript, Cascading Style Sheets, etc.) For each part of the application, inspect the mechanism which specifies the file name. For example, is the file name specified as an HTTP parameter? Once you've identified the mechanism, use an HTTP-proxy tool to try to change the values to files to which you shouldn't have access (e.g., change account158.pdf to account186.pdf or ../../WEB-INF/web.xml.) If you can access a file that you are not authorized to view, then this test fails.

T128: Test for access control bypass through user-controlled key

Priority: 9

Problem: Retrieval of a user record occurs in the system based on some key value that is under user control. The key would typically identify a user related record stored in the system and would be used to lookup that record for presentation to the user. It is likely that an attacker would have to be an authenticated user in the system. However, the authorization process would not properly check the data access operation to ensure that the authenticated user performing the operation has sufficient entitlements to perform the requested data access, hence bypassing any other authorization checks present in the system. One manifestation of this weakness would be if a system used sequential or otherwise easily guessable session ids

that would allow one user to easily switch to another user's session and view/modify their data.

Solution: Inspect the application for areas where information unique to each user, such as account balance information, is determined by input from that user, such as an HTTP-post parameter. Use an HTTP-proxy tool to modify the input, and then attempt to access another user's data. If you are able to view a different user's data where normally you wouldn't be able to do so, then this test fails.

T109: Test that users cannot supply XML Stylesheet Language Transforms (XSLTs)

Priority: 9

Problem: XSLTs transform an XML document from one format to another. XSLTs are themselves XML documents with programming language constructs such as if statements and for loops. An attacker-controlled XSLT could, for example, cause a Denial of Service (DoS) condition with an infinite loop. In some cases, XSLT engines such as Xalan for Java provide extensions into the underlying programming language. This means that, in some cases, an attacker may be able to perform remote command execution using an XSLT.

Solution: Inspect a site for any input areas or XML interfaces which accept an XSLT from the end-user. If any exist, then this test fails.

T110: Test that users cannot supply XSLTs in XML digital signatures

Priority: 9

Problem: XSLTs transform an XML document from one format to another. XSLTs are themselves XML documents with programming language constructs such as if statements and for loops. An attacker-controlled XSLT could, for example, cause a Denial of Service (DoS) condition with an infinite loop. In some cases, XSLT engines such as Xalan for Java provide extensions into the underlying programming language. This means that, in some cases, an attacker may be able to perform remote command execution using an XSLT.

Solution: Ensure you have access to a machine to which the attack target can send an ICMP message. Make note of your IP address, and use it in the following payload in place of the text

"INSERT_IP_ADDRESS_HERE." This is an example of an XML digital signature that exploits a Java security vulnerability, which you can append to the WS-Security envelope. If there is already an XML digital signature, then you need to modify the format accordingly:

```
<!-- XSLT command execution without digital signature change the ping command to  
<?xml version="1.0" encoding="UTF-8"?>  
<Envelope xmlns="urn:envelope">  
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">  
    <SignedInfo>  
      <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-  
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha  
      <Reference URI="">  
        <Transforms>  
          <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#envel  
          <Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-1999  
            <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.or  
              <xsl:template match="/">  
                <xsl:variable name="runtimeObject" select="rt:get  
                <xsl:variable name="command" select="rt:exec($run  
                <xsl:variable name="commandAsString" select="ob:t  
                <xsl:value-of select="$commandAsString"/>  
              </xsl:template>  
            </xsl:stylesheet>  
          </Transform>  
        </Transforms>  
        <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>  
        <DigestValue>uooqbWYa5VCqcJCbuymBKqml7vY=</DigestValue>  
      </Reference>  
    </SignedInfo>  
    <SignatureValue>hYlWlHBy+nwft0pcr64IdS3Hobd+RhAF6kZa1ZwA6EW3gavRXGnxIkBJC  
    <KeyInfo>  
      <X509Data>  
        <X509Certificate>MIICMzCCAZygAwIBAgIEB1vNFTANBgkqhkiG9w0BAQUFADB  
      </X509Data>  
    </KeyInfo>  
  </Signature>  
</Envelope>
```

T78: Test strength of password recovery mechanism

Priority: 8

Problem: It is common for an application to have a mechanism that provides a means for a user to gain access to their account in the event they forget their password. Very often the password recovery mechanism is weak, which has the effect of making it more likely that it would be possible for a person other than the legitimate system user to gain access to that user's account. This weakness may be that the security question is too easy to guess or find an answer to (e.g. because it is too common). Or there might be an implementation weakness in the password recovery mechanism code that may for instance trick the system into e-mailing the new password to an e-mail account other than that of the user. There might be no throttling done on the rate of password resets so that a legitimate user can be denied service by an attacker if an attacker tries to recover their password in a rapid succession. The system may send the original password to the user rather than generating a new temporary password. In summary, password recovery functionality, if not carefully designed and implemented can often become the system's weakest link that can be misused in a way that would allow an attacker to gain unauthorized access to the system. Weak password recovery schemes completely undermine a strong password authentication scheme.

Solution: In the forgotten-password function of the web application, try the following:

1. Attempt to enter in an invalid username 10 times consecutively. If you do not see a CAPTCHA (or other anti-automation technique) and you are not prevented from making further requests, then this implementation is vulnerable to user enumeration and the test fails.
2. Attempt to trigger an email to reset your password. If the email contains a plaintext password then the password is transmitted over the clear, and the test fails.
3. Attempt to answer any forgotten password questions incorrectly 10 times. If you do not see a CAPTCHA (or other anti-automation technique) and you are not prevented from making further requests, then this implementation is vulnerable to brute-forcing and this test fails.
4. If the questions are either user-generated or easy to guess (e.g., "What high school did you go to?" or "What was the color of your first car?") and you did not need to follow an emailed link to answer the questions, then this implementation may be guessed by attackers and this test fails.

How-To Implement:

I155: Manully with browser

Attempt to enter in an invalid user name 10 times consecutively. If you do not see a CAPTCHA (or other anti-automation technique) and you are not prevented from making further requests, then this implementation is vulnerable to user enumeration and this test fails.

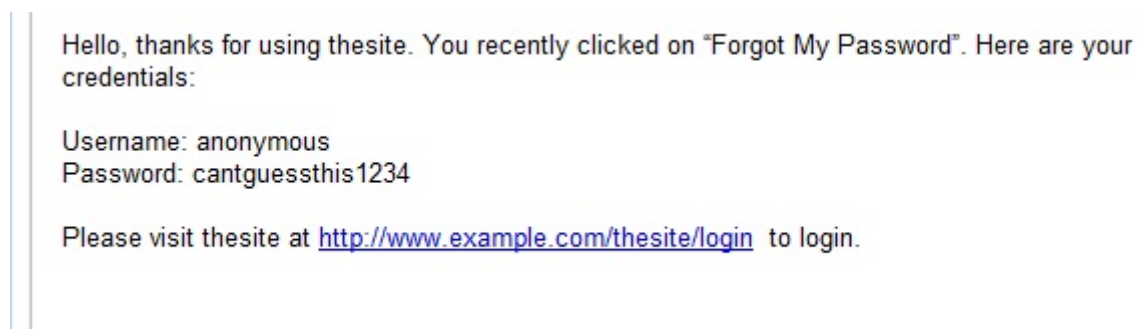
Password assistance for testaccountforsdepractice

Type the characters you see in the picture below.



Example of a CAPTCHA

Attempt to trigger an email to reset your password. If the email contains a plain text password, then the password is transmitted over the clear and this test fails.



Example of a password sent plain text

Attempt to answer any forgotten password questions incorrectly 10 times. If you do not see a CAPTCHA (or other anti-automation technique) and you are not prevented from making further requests, then this implementation is vulnerable to brute-forcing and this test fails.

If the questions are either user-generated or easy to guess (e.g. "What high school did you go to?" or "What was the color of your first car?") and you did not need to follow an e-mailed link to answer the questions, then this implementation may be guessed by attackers and this test fails.

T85: Test server-side enforcement of authorization

Priority: 8

Problem: When the server relies on protection mechanisms placed on the client side, an attacker can modify the client-side behavior to bypass the protection mechanisms resulting in potentially unexpected interactions between the client and server. The consequences will vary, depending on what the mechanisms are trying to protect.

Solution: For any page or function that appears to enforce an authorization check, attempt to browse to a page for which you do have access. Next, use an HTTP-proxy tool to modify the request sent to the unauthorized page. If you are able to successfully view the page, then this test fails.

T89: Test that site is not vulnerable to cross site scripting (XSS)

Priority: 8

Problem: Cross-site scripting (XSS) vulnerabilities occur when the software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output.

Once the malicious script is injected, the attacker can perform a variety of malicious activities. The attacker could transfer private information, such as cookies that may include session information, from the victim's machine to the attacker.

The attacker could send malicious requests to a web site on behalf of the victim, which could be especially dangerous to the site if the victim has administrator privileges to manage that site. Phishing attacks could be used to emulate trusted web sites and trick the victim into entering a password, allowing the attacker to compromise the victim's account on that web site.

Finally, the script could exploit a vulnerability in the web browser itself possibly taking over the victim's machine, sometimes referred to as "drive-by hacking." In many cases, the attack can be launched without the victim even being aware of it. Even with careful users, attackers frequently use a variety of methods to encode the malicious portion of the attack, such as URL encoding or Unicode, so the request looks less suspicious.

Solution: For each HTTP parameter name, HTTP parameter value, HTTP header name, HTTP header value, cookie name, and cookie value, attempt to send a set of known Cross-Site Scripting meta-characters. Inspect the results. If the results appear to have the same meta-character without any encoding in the resultant web page, JavaScript file, or Cascading Style Sheet (CSS) file, attempt to include a full script attack such as `<script>alert('xss')</script>`, `' onmouseover=alert(/XSS/)`, or `javascript:alert('xss')`. See the RSnake Cross-Site Scripting (XSS) cheatsheet for more potential vectors: <http://ha.ckers.org/xss.html>. If you are able to create an on-screen pop-up box, then this test fails. Note that there may be cases where your browser is immune to XSS but other browsers are vulnerable. Where possible, attempt these attacks with all supported browsers.

T99: Test that clients cannot manipulate read-only data

Priority: 8

Problem: If a web product does not properly protect assumed-immutable values from modification in hidden form fields, parameters, cookies, or URLs, this can lead to modification of critical data. Web and client-server applications often mistakenly make the assumption that data passed to the client in hidden fields or cookies is not susceptible to tampering. Failure to validate portions of data that are user-controllable can lead to the application processing incorrect, and often malicious, input. For example, custom cookies commonly store session data or persistent data across sessions. This kind of session data is normally involved in security related decisions on the server side, such as user authentication and access control. Thus, the cookies might contain sensitive data such as user credentials and privileges. This is a dangerous practice, as it can often lead to improper reliance on the value of the client-provided cookie by the server side application.

Solution: Search the site for input which should not normally be modifiable by the end-user, such as hidden form fields or cookie values. Examples include the price or currency parameter for a purchase, or the access control level of the user. Using an HTTP proxy tool, attempt to modify the input. If you can successfully modify the input, then this test fails.

T127: Test for null byte injection

Priority: 8

Problem: A null byte (NUL character) can have different meanings across representations or languages. For example, it is a string terminator in standard C libraries, but Perl and PHP strings do not treat it as a terminator. When two representations are crossed - such as when Perl or PHP invokes underlying C functionality - this can produce an interaction error with unexpected results. Similar issues have been reported for ASP. Other interpreters written in C might also be affected.

Solution: Determine which direct input the operating system uses. In particular, look for input that may be used in opening a file, such as a file-name parameter. Attempt to circumvent protections such as a restriction on file types by appending the null character (%00 in URL encoding, \0 in other forms of input). For example, "../../web.xml%00abcdef.pdf" may appear to be a legitimate PDF file, but in actuality it will open the file "web.xml" in the operating system. If you are able to circumvent a security control with the null byte, then this test fails.

T169: Test that site is not vulnerable to DOM-based cross site scripting (XSS)

Priority: 8

Problem: Cross-site scripting (XSS) vulnerabilities occur when the software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output.

Once the malicious script is injected, the attacker can perform a variety of malicious activities. The attacker could transfer private information, such as cookies that may include session information, from the victim's machine to the attacker.

The attacker could send malicious requests to a web site on behalf of the victim, which could be especially dangerous to the site if the victim has administrator privileges to manage that site. Phishing attacks could be used to emulate trusted web sites and trick the victim into entering a password, allowing the attacker to compromise the victim's account on that web site.

Finally, the script could exploit a vulnerability in the web browser itself possibly taking over the victim's machine, sometimes referred to as "drive-by hacking." In many cases, the attack can be launched without the victim even being aware of it. Even with careful users, attackers frequently use a variety of methods to encode the malicious portion of the attack, such as URL encoding or Unicode, so the request looks less suspicious.

Solution: To test for DOM-based XSS, inspect all the JavaScript content, either embedded inside HTML files (in script tags, event handlers, etc.), or inside .js files. Look for and identify

user-controlled values (also known as "Sources"), and look for places that can render the values/functions (also known as "Sinks").

The most commonly known Sources are:

```
document.URL
document.URLUnencoded
document.location (and many of its properties)
document.referrer
window.location (and many of its properties)
location (note that window.location can be access as just location)
```

The commonly used Sinks are:

```
element.innerHTML = "...";
element.outerHTML = "...";
document.write(...);
document.writeln(...);
```

If the value of a Source variable is directly used for output generation via a Sink method without explicit validation/sanitization of the content, then this test fails and the web site is vulnerable to DOM-based cross site scripting. For example, the following code would fail:

```
document.writeln(document.referrer)
```

Note that we recommend performing this test by inspecting the JavaScript source (which is always available) rather than executing runtime tests because inspecting the source is more reliable.

T123: Test for integer overflow

Priority: 8

Problem: An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may wrap to become a very small or negative number. While this may be intended behavior in circumstances that rely on wrapping, it can have security consequences if the wrap is unexpected. This is especially the case if the integer overflow can be triggered using user-supplied inputs. This becomes security-critical when the result is used to control looping, make a security decision, or determine the offset or size in behaviors such as memory

allocation, copying, concatenation, etc.

Solution: Identify sections of the site that perform integer arithmetic with user-supplied values. Provide large negative and positive values and inspect the result. If the result is a negative when it should be a positive, or vice-versa, then this test fails. Here are some values to try: -1 0 0x100 0x1000 0x3ffffff 0x7ffffffe 0x7ffffff 0x80000000 0xffffffffe 0xffffffff 0x10000 0x100000

How-To Implement:

I165: Manually with browser or client

1. Identify transactions that perform integer arithmetic with user-supplied values. For security testing, look for transactions which could have a security impact, such as currency transactions.

For each transaction from step 1, try entering in the following decimal values:

2147483647
-2147483648
9223372036854775807
-9223372036854775808

If the result of the arithmetic is erroneous, then this test fails.

T80: Test password requirements

Priority: 7

Problem: An authentication mechanism is only as strong as its credentials. For this reason, it is important to require users to have strong passwords. Lack of password complexity significantly reduces the search space when trying to guess user's passwords, making brute-force attacks easier.

Solution: Log in as an end-user and attempt to change your password such that it does not meet all individual requirements in your organization's password standards. If you are successful in changing your password, then this test fails.

How-To Implement:

I157: Manually with browser

1. Log on to application.
2. Locate the "Change password" form.
3. Try to change the password such that it violates your organization's password policies. For example, see if you can change the password to a length of less than six characters.
4. If you are able to change the password successfully, then this test fails.

T84: Test page-level authorization

Priority: 7

Problem: Web applications susceptible to direct request attacks often make the false assumption that such resources can only be reached through a given navigation path and so only apply authorization at certain points in the path.

Solution: Attempt to browse each unique dynamic page in the authenticated portion of the site without a valid session cookie. The test fails for any page that does not require the session cookie.

How-To Implement:

I140: Manually with browser

1. Get two sets of credentials to the application; one for a regular or low-privilege user, and one for an administrator or other high-privilege user.
2. Log in as the high-privilege user and make note of the pages you have access to which should not be visible to the low-privilege user. If there are 10 or more such pages, take a sample of 10 pages from across the application.
3. Note down the URLs for the pages identified in step 2, including any required GET or POST parameters.
Log in as the low-privilege user and attempt to access the pages from step 3. If you are able to access any of the pages, then this test fails.

T93: Test that sessions expire upon logout

Priority: 7

Problem: A user's session should expire upon logging out of the application. If the session does not expire, this may provide a window of opportunity for an attacker to hijack the session. This is particularly important for applications on shared devices.

Solution: Authenticate into the application, view some pages, and then log out. After logging out of the application, clear your cache and attempt to revisit one of the authenticated pages. If you are able to access the page successfully, then the test fails.

T96: Test that site is not vulnerable to cross site request forgery (CSRF)

Priority: 7

Problem: When a web server is designed to receive a request from a client without any mechanism for verifying that it was intentionally sent, then it might be possible for an attacker to trick a client into making an unintentional request to the web server which will be treated as an authentic request. This can be done via a URL, image load, XMLHttpRequest, etc. and can result in data disclosure or unintended code execution.

Solution: Determine which transactions are considered "high-value" according to organizational values (e.g., impact on peoples' safety, monetary impact, brand reputation, etc.) For each such transaction, note down the HTTP request using an HTTP proxy tool. Next, authenticate into the application as a different user with the same privileges and access the same transaction with the same values. Compare the HTTP requests from both users. If they are exactly the same, then this test fails. If they are different, but the difference is easily guessable (e.g., an incremental, guessable parameter such as 'transactionid'), then this test fails.

T98: Test that input validation is performed on server

Priority: 7

Problem: When software fails to validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.

Solution: Find HTTP parameters and cookies which appear to be validated correctly. For each such input, enter a valid value. Next, using an HTTP-proxy tool, convert the value to invalid. If the input is no longer validated, then this test fails.

T125: Test for open redirect

Priority: 7

Problem: An http parameter may contain a URL value and could cause the web application to redirect the request to the specified URL. By modifying the URL value to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials. Because the server name in the modified link is identical to the original site, phishing attempts have a more trustworthy appearance.

Solution: Look for input values with a redirect used by the application. For example, many sites follow a pattern such as any of the following (the name of parameter may vary):

```
http://www.example.org/login?url=startPage
http://www.example.org/login?next=startPage
http://www.example.org/login?ref=/startPage
...
```

where after successful authentication you are automatically redirected to
`www.example.org/startPage`.

For each parameter used in a redirect, attempt to specify an external site such as
`http://www.google.com`. If you are redirected to the external site upon successful login, then this test fails.

T167: Test that the site is not vulnerable to JSON Hijacking

Priority: 7

Problem: JSON Hijacking takes advantage of a feature in some browsers that allow script to override the core language's object setter routines, which allows malicious JavaScript to interject logic that allows it to monitor JSON messages returned from a server.

In a JSON Hijacking attack, the attacker overrides the JavaScript array constructor to steal information passed to it. The attacker then embeds the JSON endpoint of the domain being hijacked in an HTML script tag. The browser will treat the cross domain resource as a JavaScript file and tries to render it, resulting in a call to the attacker's version of the array constructor, leading to disclosure of the information across domains.

Solution: Use an HTTP-proxy/inspection tool to ensure that the returned JSON format for AJAX endpoints uses the correct MIME-type, **application/json**, AND conforms to at least one of these secure formats:

- Use JSON objects (dictionary) as the highest level construct
- Use Comment Filtered JSON format

You can use any inspection suite, such as Burpsuite, Fiddler, Firebug, IE Developer Tools, to view the AJAX messages. Check the first few character of each JSON message.

- Set the filter to XHR (XMLHttpRequest) to capture only AJAX messages. (if the tool allows that)
- Inspect AJAX responses. If the first non-white-space character is "[" (array constructor), then the AJAX endpoint is vulnerable to JSON Hijacking. Otherwise, if it starts with "/" (comment), or "{" (object constructor), then it is not vulnerable.
- Check the MIME type. The correct MIME-type is application/json, text/json-comment-filtered, or text/json-comment-filtered.

T82: Test authentication error consistency

Priority: 6

Problem: This issue frequently occurs during authentication, where a difference in failed-login messages could allow an attacker to determine if the username is valid or not. These exposures can be inadvertent (bug) or intentional (design).

Solution: Attempt to log in with an invalid username. Inspect the result. Next, log in with a valid username but invalid password and inspect the result. If the two results differ, then the test fails.

T107: Test that site forbids uploading malware

Priority: 6

Problem: The software allows the attacker to upload or transfer files of dangerous types that can be automatically processed within the product's environment.

Solution: For any file-upload functionality, attempt to upload the EICAR test file. This file is a benign file that should trigger a virus alert. You can retrieve the file from here: http://www.eicar.org/anti_virus_test_file.htm. If the file uploads successfully, then the test fails.

T116: Test for regular expression denial of service

Priority: 6

Problem: Improperly crafted regular expressions may lead to denial of service conditions. In particular, large payloads may lead to exponential complexity which can allow an attacker to cause a denial of service with relatively little input. See [OWASP Regular Expression DoS](#) for more details.

Solution: Determine which inputs might be subject to a regular expression. Send input that contains several consecutive valid characters followed by one invalid character. For example, for alpha-numeric input you can try:

"aaa!" If the server hangs or crashes as a result of this input, then the test fails.

How-To Implement:

I137: Manually with browser

NOTE: This test involves a denial-of-service attack and may significantly disrupt availability. Perform this test only if you wish to test for denial-of-service.

1. Browse the site and attempt to look for functionality where the application validates user input. In particular, look for string rather than simple numeric validation.
2. Figure out the validation logic by sending a variety of input strings. For example, send an alphabetic string, an alpha-numeric string, a numeric string, and a string with alphabetic and numeric characters. Try to isolate which characters pass validations and which ones don't.
3. For the input from step 2, create a string of 15 valid characters and one invalid character. If the application fails to respond for 10 or more seconds because of server processing, then the test fails. If it does respond but there is a noticeable time lag, double the number of valid characters. If the server fails to respond for 10 or more seconds, then the test fails.

T126: Test for XML external entity disclosure

Priority: 6

Problem: XML documents optionally contain a Document Type Definition (DTD), which, among other features, enables the definition of "XML entities". It is possible to define an entity locally by providing a substitution string in the form of a URL whose content is substituted for the XML entity when the DTD is processed. The attack can be launched by defining an XML entity whose content is a file URL (which, when processed by the receiving end, is mapped into a file on the server), that is embedded in the XML document, and thus, is fed to the processing application. This application may echo back the data (e.g. in an error message), thereby exposing the file contents.

Solution: For any input that may be output inside of an XML document, attempt to add a payload that introduces a new Document Type Definition (DTD). This DTD should reference an external file on the system, such as c:\boot.ini for Windows or /etc/passwd for Unix. Next, attempt to reference that entity later in the document. If the contents of the file are returned back to you, then this test fails. Example payload:

```
<!DOCTYPE foo [  
<!ELEMENT foo ANY>  
<!ENTITY xxe SYSTEM "file:///c:/boot.ini">  
<foo>&xxe;</foo>
```

or

```
<!DOCTYPE foo [  
<!ELEMENT foo ANY>  
<!ENTITY xxe SYSTEM "file:///etc/shadow">  
<foo>&xxe;</foo>
```

How-To Implement:

I139: Manually with browser

1. Look for functions where end-users can supply XML to the application directly.
2. Determine what a normal XML message should look like. Keep a sample of the normal XML message.

Modify the normal XML message such that it has the following line:

```
<!DOCTYPE foo [<!ELEMENT root ANY><!ENTITY xxe SYSTEM "file:///c:/">]>
```

for Windows machines or

```
<!DOCTYPE foo [<!ELEMENT root ANY><!ENTITY xxe SYSTEM "file:/// ">]>
```

for Unix machines. If in doubt, try both. This line should be just after the line starting with "<?xml version." If you don't have such a line, then the "<!DOCTYPE" line should be first. In place of "root," you should enter the name of the first XML element in your document. For example, if your document starts with <xmlMessage> and the server is a Unix server, the attack should be

```
<!DOCTYPE foo [<!ELEMENT xmlMessage ANY><!ENTITY xxe SYSTEM "file:/// ">]>
```

1. Next, you will need to find a place to put "&xxe;" within your XML message. Try to find an element that will be echoed back to you in the response. This normally happens in error conditions. In the accompanying video, we modify the <root> element to be <root>&xxe;</root>.
2. Send the XML request and examine the response. If the response contains the directory listing, then this test fails. If the response returns an error message about insufficient access rights to open a file (thus indicating that it attempted to open a file), then this test fails. If the response does not indicate any error, there is still a good chance that this attack worked, but you should move on to step 6. If the response indicates a generic error or an error saying that DTDs are not allowed, then this test passes.

Attempt to change the file path from "c:/" or "/" to a non-existent path, such as "//aaaaaaabbbbb." If the response indicates that no such file exists, then this test fails.

T79: Test password change function

Priority: 5

Problem: This could be used by an attacker to change passwords for another user, thus gaining the privileges associated with that user.

Solution: Inspect the application for all pages where you can change your password after authentication. In each such page, attempt to reset your password. If you can reset your password without entering the old password, then this test fails. If you can reset your password by entering a false old password, then this test fails.

How-To Implement:

I156: Manually with browser

Inspect the application for all pages where you can change your password after authentication. In each such page, attempt to reset your password. If you can reset your password without entering an old password, then this test fails. If you can reset your password by entering a false old password, then this test fails.

Change password: root

Enter a new password for the user **root**.

Password:	<input type="password"/>
Password (again):	<input type="password"/>
Enter the same password as above, for verification.	
<input type="button" value="Change password"/>	

Example of a of change password form that does not require an old password

T83: Test for transactional authentication

Priority: 5

Problem: Due to the high number of real world attacks that bypass authentication or session management, attackers are often able to perform any transaction once they compromise a user's account or session. Without sufficient controls, an attacker may be able to perform high sensitive transactions such as transfer large sums of money.

Solution: Determine which transactions are considered "high-value" according to organizational values (e.g., impact on peoples' safety, monetary impact, brand reputation, etc.) For each such transaction, determine whether or not a user is required to re-enter a password or to use another form of authentication (e.g., a one-time-use password). If not, then the test fails.

How-To Implement:

I158: Manually with browser

Determine which transactions are considered high-value according to organizational values (e.g. impact on peoples' safety, monetary impact, brand reputation, etc.).

For each such transaction, determine if you are required to re-enter your password or to enter in another form of authentication (e.g. a one-time use password). If not, then this test fails.

Normally, you can verify step #2 by attempting a high-value transaction via the application. If the steps needed from the start, where you selected to start the transaction, to end, where the transaction was placed in the system, could be followed without having some form of password, then the test fails.

Are you sure you wish to transfer **\$100,000** to **Company, Inc.**?

Submit

Example of a transaction lacking authentication

T90: Test idle session timeout

Priority: 5

Problem: According to WASC, "Insufficient Session Expiration is when a web site permits an attacker to reuse old session credentials or session IDs for authorization."

Solution: Authenticate into the application and browse to an access-controlled page. Do not use this session for the stipulated number of minutes (e.g., 30 minutes) and then attempt to use the same session to re-access the page. Note that you may need to clear your cache prior to testing this. If you are able to access the same page successfully without being logged out automatically, then this test fails.

How-To Implement:

I159: Manually with browser

1. Log on to the application.
2. Do not use the application for 31 minutes (or 1 minute longer than the invalid session timeout should be). Refresh the current page of the application. If the session is still valid, then this test fails.

T94: Test that session IDs are not leaked through URLs

Priority: 5

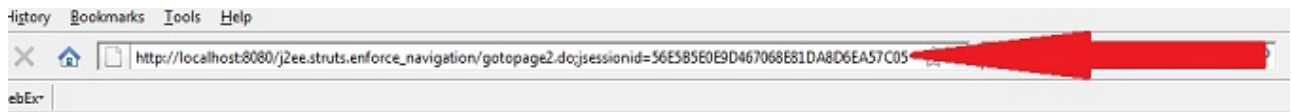
Problem: Applications that allow URL rewriting for session IDs may leak the session IDs to other users on the same computer in the browser history or through links to external sites.

Solution: If at any point in the application you notice the session ID in the URL, then this test fails. Using a cookie editor, make note of the session ID. Using a different browser or computer, attempt to navigate to an authenticated page with the session ID in the URL. For example: `http://www.example.com/page;jsessionid=12345678914?param1=val`. If you are able to access this page successfully, then the test fails.

How-To Implement:

I160: Manually with browser

While browsing the site, look for any instance in which the session ID is in the URL. If you see the session ID in the URL, then this test fails.



Example of a Session ID in the URL

T95: Test for the absence of remember-me

Priority: 5

Problem: According to WASC, "Insufficient Session Expiration is when a web site permits an attacker to reuse old session credentials or session IDs for authorization."

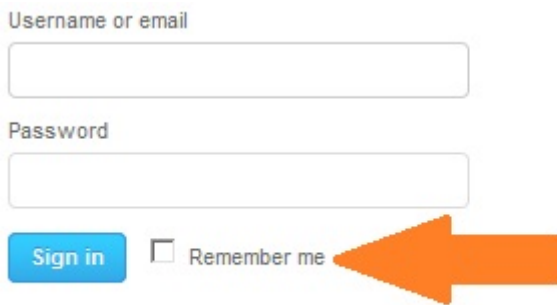
Solution: Inspect the login page for a "remember-me" function that allows for persistent sessions (i.e., you are never logged out). If the function exists, then this test fails.

How-To Implement:

I151: Test for the absense of remember-me manually with browser

1. Open your web browser and navigate to the login page.

2. Inspect the login page. If you see "remember-me" functionality then this test fails



A screenshot of a login form. It features two input fields: 'Username or email' and 'Password'. Below the password field is a blue 'Sign in' button and a checkbox labeled 'Remember me'. A large orange arrow points to the 'Remember me' checkbox, indicating its presence as a failure condition for the test.

Example of a remember me function

T86: Test session ID rotation after authentication

Priority: 4

Problem: Such a scenario is commonly observed when: 1. A web application authenticates a user without first invalidating the existing session, thereby continuing to use the session already associated with the user 2. An attacker is able to force a known session identifier on a user so that, once the user authenticates, the attacker has access to the authenticated session 3. The application or container uses predictable session identifiers. 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 associate, and possibly authenticate, against the server using that session identifier, giving the attacker access to the user's account through the active session.

Solution: Browse to the login page. Make note of current session ID using either an HTTP-proxy tool or a cookie viewer. Authenticate and inspect the session ID. If the session ID did not change, then this test fails.

T88: Test that secure flag is set on session cookie

Priority: 4

Problem: The Secure attribute for sensitive cookies in HTTPS sessions is not set, which could cause the user agent to send those cookies in plaintext over an HTTP session.

Solution: Authenticate into the application and use a cookie-editing tool to inspect the session ID. Determine the presence or absence of the ";secure" flag. If the flag is absent, the test fails.

T91: Test that httponly flag is set on session cookie

Priority: 4

Problem: The 'HttpOnly' cookie flag reduces the impact of cross-site scripting. Cookies with the 'HttpOnly' cookie flag cannot be accessed by JavaScript. Session IDs in cookie without the 'HttpOnly' cookie flag may be read and transmitted by malicious JavaScript, allowing an adversary to gain access to the user's session.

Solution: Authenticate into the application and use a cookie-editing tool to inspect the session ID. Determine the presence or absence of the ";HttpOnly" flag. If the flag is not present, then this test fails.

T111: Test that site turns off auto-complete for confidential data fields

Priority: 4

Problem: The autocomplete function of most browsers may cache sensitive information, such as credit card numbers. The cached data from autocomplete may be accessed by other users on the same computer.

Solution: For any confidential field (e.g., user ID, password, credit card number, social insurance/security number, etc.) in an HTML form-input tag, inspect the HTML for the presence of the `autocomplete=off` attribute. If the field is not present, then this test fails.

Note that the attribute `autocomplete` might be set in the parent `form` tag. In such a case, the attribute gets inherited by all the child `input` tags.

Functional Test

Alternatively, you can try entering the confidential data once and submitting the form. The browsing back to the same page and start typing the same data slowly. If the auto-complete feature suggests the rest of the data, the test fails.

How-To Implement:

I147: Test that site turns off autocomplete for confidential data fields manually with browser

1. Browse the site for confidential data within form fields (e.g. user ID, password, credit card number, social insurance/security number, etc.).
2. For any such page, right-click on the page and select "View source."
3. In the page source, look for corresponding fields from Step 1. Inspect the HTML for the presence of the `autocomplete='off'` attribute. The `autocomplete` tag might be set for each field, or set for the parent `form` tag. If `autocomplete='off'` is not present for the HTML `input` tag corresponding to the confidential field, or the parent HTML `form` tag, then the test fails.

T129: Test for reliance on file name or extension of externally-supplied file

Priority: 4

Problem: An application might use the file name or extension of a user-supplied file to determine the proper course of action, such as selecting the correct process to which control should be passed, deciding what data should be made available, or what resources should be allocated. If the attacker can cause the code to misclassify the supplied file, then the wrong action could occur. For example, an attacker could supply a file that ends in a ".php.gif" extension that appears to be a GIF image, but would be processed as PHP code. In extreme cases, code execution is possible, but the attacker could also cause exhaustion of resources, denial of service, information disclosure of debug or system data (including application source code), or being bound to a particular server side process. This weakness may be due to a vulnerability in any of the technologies used by the web and application servers, due to misconfiguration, or resultant from another flaw in the application itself.

Solution: Inspect the application for file-upload functionality. For every part of the site that provides file upload, determine if there are restrictions on the file type. If there are restrictions, save a file of a different type with the file extension that the application inspects. For example, if the application only allows users to upload .jpg images: Create a new text file, save the extension as a JPG rather than a TXT, and then upload it. If the application does not restrict the upload, then this test fails.

How-To Implement:

I138: Manually with browser

1. Find file-upload functionality within the application site.
2. Determine whether the functionality restricts files based on their type. For example, the functionality may permit images or PDFs only.
3. Open a text editor, type in "test" (or some other text), click File->Save As, choose "Save as type: All Files," and give the file an extension from Step 2. For example, if the site allows .jpg images only, call the file "file.jpg."
Upload the file from step 3. If the file uploads successfully, then this test fails.

T131: Test for forced password change upon login

Priority: 4

Problem: If a system automatically generates a password, a malicious administrator may be able to retrieve the automatically-generated password. If an automatically generated password is distributed in plain text a malicious user may intercept the password.

Solution: 1. Either register as a new user or ask an administrator to give you a new password 2. Log in to the application 3. If the application does not ask you to change your password immediately then this test fails

T104: Test that site does not reveal detailed information in error pages

Priority: 3

Problem: The software fails to return custom error pages to the user, possibly exposing sensitive information.

Solution: Attempt to send invalid characters and invalid paths to the web application. If the error message provides details about the internals of the application such as a stack trace, then this test fails.

How-To Implement:

I163: Manually with browser

Attempt to send invalid characters, such as "(%^&&#@!<>)" and invalid paths to the web application. If the error message provides details about the internals of the application, such as a stack trace, then this test fails.

HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

```
org.apache.jasper.JasperException: An exception occurred processing JSP page /index.jsp at line 13

10:
11:
12:
13: <% throw new RuntimeException("Test"); %>
```

Stacktrace:

```
org.apache.jasper.servlet.JspServletWrapper.handleJspException(JspServletWrapper.java:510)
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:419)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:313)
org.apache.jasper.servlet.JspServlet.service(JspServlet.java:260)
javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
```

root cause

```
java.lang.RuntimeException: Test
    org.apache.jsp.index_jsp._jspService(index_jsp.java:61)
    org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:70)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
    org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:377)
    org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:313)
```

Example of a detailed error message

T112: Test that site does not cache confidential pages

Priority: 3

Problem: For each web page, the application should have an appropriate caching policy specifying the extent to which the page and its form fields should be cached.

Solution: For pages containing confidential data such as credit card numbers, inspect the page in an HTTP-proxy tool. If the request uses HTTP 1.0, then look for either of the following:

- Pragma=no-cache
- Date - Expires=-1

If the request uses HTTP 1.1 then look for:

- Cache-Control=no-store, no-cache, must-revalidate

If the appropriate header isn't included, then this test fails.

Note that for HTTP 1.1, the important keyword is "no-store". A common misuse is to use only the "no-cache" directive which according to the RFC, tells the browser that it should revalidate with the server before serving the page from the cache. Hence, if the "no-store" directive is missing and the "no-cache" directive is used, the browser, can still store the page in its cache, but would display it only after revalidating with the server, resulting in unwanted storage and leakage of confidential data.

Although in practice, newer versions of IE and Firefox have started treating the no-cache directive as if it instructs the browser not to even cache the page (due to widespread misuse of no-cache for no-store), it is strongly advised that the "no-store" directive be specified.

T113: Test that site is not vulnerable to HTTP verb tampering

Priority: 3

Problem: HTTP specifies many different methods (or verbs), most prominently GET, POST, HEAD, PUT, etc. Web applications generally use GET and POST methods exclusively for requests.

Many web applications will limit which HTTP verbs can be used to access a resource (most likely GET or POST) by defining a security constraint within web.xml and explicitly listing GET and POST within separate <http-method> tags.

The HTTP HEAD method works exactly like a GET method, with the exception that the server must not return a message body in the response [1]. Note also that the HTTP specification intended that GET requests (along with HEAD) "should not have the significance of taking an action other than retrieval." In other words, GET requests are meant to be idempotent – they should not change the state of the application.

Unfortunately, many applications define non-idempotent GET requests, that is, they treat GET requests just like POST requests in that they can execute transactions that change the state of the application. An example of such a URL would be:

www.myapp.org/mgmt/accts.jsp?q=delAcct.

Therefore, an attacker could attempt to trap the above GET request in an HTTP proxy, modify it to a HEAD request, and submit the request. Since HEAD is not one of the HTTP methods listed in web.xml, the authorization check would be bypassed and the code that processes GET requests may be executed. This is known as an HTTP Verb Tampering attack.

Furthermore, some frameworks such as Java EE allows the use of arbitrary HTTP verbs. An attacker can send a "BOB" requests rather than HEAD, and since it is not listed in the security constraint, the constraint will be bypassed. Arbitrary verbs will not work when accessing a Servlet, but is acceptable if the request targets a JSP directly. This is because when a JSP is compiled its components are sent to a service() method, rather than the doGet() or doPost() methods of a Servlet. Therefore, any arbitrary verb that is submitted will be directed to the service() method.

[1] <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>

Solution: Using an HTTP-proxy tool, attempt to browse to authenticated web pages without a valid session. Modify the HTTP verb from "GET" or "POST" to a random set of characters, such as "PWRLWA." If you are able to view the page with the random verb but are unable to view the page with a valid HTTP verb, then this test fails. You can also try using the "HEAD" request instead of "GET." Note that you are unlikely to receive a response, however, try to ascertain if the request was processed successfully (e.g., view transaction history). If the "HEAD" request worked but the "GET" request did not, then this test fails.

T92: Test absolute session timeout

Priority: 2

Problem: Absolute or hard session timeouts impose a maximum age on the life of a session, regardless of activity level. This control limits the potential harm an attacker can inflict upon session hijacking to a finite period. Without this control, an attacker who successfully hijacks a session can keep the session alive for as long as the server keeps track of sessions.

Solution: Authenticate into the application and attempt to browse to an access-controlled page. Write a simple script to access the page regularly for the stipulated time period (e.g., 24 hours). If you are able to access the same page successfully without being automatically logged out, then this test fails.

How-To Implement:

I129: Manually with browser

1. Open your browser.
2. Browse to the application and authenticate.
3. Open the [attached HTML file](#) in your browser. Set the URL to the home page of the site you are testing (e.g. `https://mail.google.com/mail/` for GMail). If necessary, change the "GET" to "POST."
4. Click on the Start KeepAlive button.
5. Let the page run for a full day (or however long the absolute session timeout should be) and then browse back to the homepage in your browser. If the session is still valid, then this test fails.

T124: Test for authentication timing vulnerability

Priority: 2

Problem: Two separate operations in a product require different amounts of time to complete, in a way that is observable to an actor and reveals security-relevant information about the state of the product, such as whether a particular operation was successful or not.

Solution: Collect five valid usernames and five invalid usernames. Write a script that attempts authentication with each username. Time the latency in response for all 10 usernames. If there is a clear difference in the time it takes for invalid usernames versus valid usernames, then this test fails.

Note that this test looks for the commonly used prevention mechanism for timing vulnerabilities, which is adding random delay to failed logins, rather than the exploit-ability of the vulnerability itself.

T132: Test for standard encoding format for all HTML content

Priority: 2

Problem: The software fails to properly handle encoding or decoding of the data, resulting in unexpected values.

Solution: Find a sample of pages from the application to test. For each page, use a proxy tool to check if the HTTP response specifies a character set such as "Content-Type: text/html; charset=UTF-8". If the character set is not specified then check the HTML page for the "meta" tag and see if the content=attribute is set, such as "content="text/html". If, for any of the pages, neither the HTTP header nor the HTML specify a character set then this test fails. If the pages specify different character sets then this test fails.

T108: Test that site restricts access to WSDL file

Priority: 2

Problem: The Web services architecture may require exposing a WSDL file that contains information on the publicly accessible services and how callers of these services should interact with them (e.g. what parameters they expect and what types they return).

Solution: For each web service endpoint, attempt to browse to the endpoint without a valid session. Next, append a "?WSDL" to the URL. If you are able to access the WSDL, then this test fails.