MODULE 2- ADVANCED WEB APPLICATION ATTACKS AND REMEDY

Introduction –

Web application security is broken in to two distinct parts: the ability to attack servers and systems that host web applications, and the ability to use web applications to attack the clients and users that use them. Below are the attacks which are responsible for creating a glitch in the software requirements and leading to some serious business impacts.

A1-Injection

Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code. They are often found in SQL, LDAP, Xpath, or NoSQL queries; OS commands; XML parsers, SMTP Headers, program arguments, etc. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.

Mitigation –

Preventing injection requires keeping untrusted data separate from commands and queries.

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

Example Attack Scenarios –

Scenario #1: The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='" +
request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts WHERE
custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in her browser to send: ' or '1'='1. For example:

```
http://example.com/app/accountView?id=' or '1'='1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.

Relevant Vulnerabilities –

1. SQL Injection –

Summary –

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application

Mitigation –

Primary Defenses:

- Option #1: Use of Prepared Statements (Parameterized Queries)
- Option #2: Use of Stored Procedures
- Option #3: Escaping all User Supplied Input

Additional Defenses:

- Also Enforce: Least Privilege
- Also Perform: White List Input Validation

Use of Prepared Statements (Parameterized Queries) –

Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.

Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker. In the safe example below, if an attacker were to enter the userID of tom' or '1'='1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'='1.

Language specific recommendations:

- Java EE – use PreparedStatement() with bind variables
- .NET – use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
- PHP – use PDO with strongly typed parameterized queries (using bindParam())
- Hibernate - use createQuery() with bind variables (called named parameters in Hibernate)
- SQLite - use sqlite3_prepare() to create a statement object

In rare circumstances, prepared statements can harm performance. When confronted with this situation, it is best to either a) strongly validate all data or b) escape all user supplied input using an escaping routine specific to your database vendor as described below, rather than using a prepared statement. Another option which might solve your performance issue is to use a stored procedure instead.

Safe Java Prepared Statement Example

The following code example uses a PreparedStatement, Java's implementation of a parameterized query, to execute the same database query.

```
String custname = request.getParameter("customerName"); // This should REALLY be validated too

 // perform input validation to detect attacks

 String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";

  PreparedStatement pstmt = connection.prepareStatement( query );

 pstmt.setString( 1, custname);

 ResultSet results = pstmt.executeQuery( );
```

Safe C# .NET Prepared Statement Example

With .NET, it's even more straightforward. The creation and execution of the query doesn't change. All you have to do is simply pass the parameters to the query using the Parameters.Add () call as shown here.

```
String query =
          "SELECT account_balance FROM user_data WHERE user_name = ?";
try {
         OleDbCommand command = new OleDbCommand(query, connection);
         command.Parameters.Add(new OleDbParameter("customerName", CustomerName Name.Text));
         OleDbDataReader reader = command.ExecuteReader();
         // …
} catch (OleDbException se) {
         // error handling
}
```

We have shown examples in Java and .NET but practically all other languages, including Cold Fusion, and Classic ASP, support parameterized query interfaces. Even SQL abstraction layers, like the Hibernate Query Language (HQL) have the same type of injection problems (which we call HQL Injection). HQL supports parameterized queries as well, so we can avoid this problem:

Hibernate Query Language (HQL) Prepared Statement (Named Parameters) Examples

First is an unsafe HQL Statement

```
Query unsafeHQLQuery = session.createQuery("from Inventory where productID='"+userSuppliedParameter+"'");
```

Here is a safe version of the same query using named parameters

```
Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid");

safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

For examples of parameterized queries in other languages, including Ruby, PHP, Cold Fusion, and Perl, see the Query Parameterization Cheat Sheet.

Developers tend to like the Prepared Statement approach because all the SQL code stays within the application. This makes your application relatively database independent. However, other options allow you to store all the SQL code in the database itself, which has both security and non-security advantages. That approach, called Stored Procedures, is described next.

Use of Stored Procedures

Stored procedures have the same effect as the use of prepared statements when implemented safely. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application. Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you.

Safe Java Stored Procedure Example

The following code example uses a Callable Statement, Java's implementation of the stored procedure interface, to execute the same database query. The "sp_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above

```
String custname = request.getParameter("customerName"); // This should
REALLY be validated
 try {
        CallableStatement cs = connection.prepareCall("{call
sp_getAccountBalance(?)}");
        cs.setString(1, custname);
        ResultSet results = cs.executeQuery();
        // … result set handling
 } catch (SQLException se) {
        // … logging and error handling
 }
```

Safe VB .NET Stored Procedure Example

The following code example uses a SqlCommand, .NET's implementation of the stored procedure interface, to execute the same database query. The "sp_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

```
Try
        Dim command As SqlCommand = new
SqlCommand("sp_getAccountBalance", connection)
        command.CommandType = CommandType.StoredProcedure
        command.Parameters.Add(new SqlParameter("@CustomerName",
CustomerName.Text))
        Dim reader As SqlDataReader = command.ExecuteReader()
        ' …
 Catch se As SqlException
        ' error handling
 End Try
```

Escaping All User Supplied Input

This third technique is to escape user input before putting it in a query. This methodology is frail compared to using parameterized queries and we cannot guarantee it will prevent all SQL Injection in all situations. This technique should only be used, with caution, to retrofit legacy code in a cost effective way. Applications built from scratch, or applications requiring low risk tolerance should be built or re-written using parameterized queries

Least Privilege

To minimize the potential damage of a successful SQL injection attack, you should minimize the privileges assigned to every database account in your environment. Do not assign DBA or admin type access rights to your application accounts. We understand that this is easy, and everything just 'works' when you do it this way, but it is very dangerous. Start from the ground up to determine what access rights your application accounts require, rather than trying to figure out what access rights you need to take away. Make sure that accounts that only need read access are only granted read access to the tables they need access to. If an account only needs access to portions of a table, consider creating a view that limits access to that portion of the data and assigning the account access to the view instead, rather than the underlying table. Rarely, if ever, grant create or delete access to database accounts.

White List Input Validation

Input validation can be used to detect unauthorized input before it is passed to the SQL query. White list validation is appropriate for all input fields provided by the user. White list validation involves defining exactly what IS authorized, and by definition, everything else is not authorized. If it's well-structured data, like dates, social security numbers, zip codes, e-mail addresses, etc. then the developer should be able to define a very strong validation pattern, usually based on regular expressions, for validating such input. If the input field comes from a fixed set of options, like a drop down list or radio buttons, then the input needs to match exactly one of the values offered to the user in the first place. The most difficult fields to validate are so called 'free text' fields, like blog entries. However, even those types of fields can be validated to some degree, you can at least exclude all non-printable characters, and define a maximum size for the input field.

2. Command Injection –

Summary –

Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application. Command injection attacks are possible largely due to insufficient input validation.

Mitigation –

Input validation refers to the process of validating all the input to an application before using it. Input validation is absolutely critical to application security, and most application risks involve tainted input at some level.

3. XML External Entity (XXE) Processing

Summary –

An XML External Entity attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of confidential data, denial of service, port scanning from the perspective of the machine where the parser is located, and other system impacts.

Attacks can include disclosing local files, which may contain sensitive data such as passwords or private user data, using file: schemes or relative paths in the system identifier. Since the attack occurs relative to the application processing the XML document, an attacker may use this trusted application to pivot to other internal systems, possibly disclosing other internal content via http(s) requests. In some situations, an XML processor library that is vulnerable to client-side memory corruption issues may be exploited by dereferencing a malicious URI, possibly allowing arbitrary code execution under the application account. Other attacks can access local resources that may not stop returning data, possibly impacting application availability if too many threads or processes are not released.

Mitigation –

Since the whole XML document is communicated from an untrusted client, it's not usually possible to selectively validate or escape tainted data within the system identifier in the DTD. Therefore, the XML processor should be configured to use a local static DTD and disallow any declared DTD included in the XML document.

C/C++

libxml2

The Enum xmlParserOption should not have the following options defined:

- XML_PARSE_NOENT: Expands entities and substitutes them with replacement text
- XML_PARSE_DTDLOAD: Load the external DTD

Java

Java applications using XML libraries are particularly vulnerable to XXE because the default settings for most Java XML parsers is to have XXE enabled. To use these parsers safely, you have to explicitly disable XXE in the parser you use. The following describes how to disable XXE in the most commonly used XML parsers for Java.

JAXP DocumentBuilderFactory and SAXParserFactory

Both DocumentBuilderFactory and SAXParserFactory XML Parsers can be configured using the same techniques to protect them against XXE. Only the DocumentBuilderFactory example is presented here. The JAXP DocumentBuilderFactory setFeature method allows a developer to control which implementation-specific XML processor features are enabled or disabled. The features can either be set on the factory or the underlying XMLReader setFeature method. Each XML processor implementation has its own features that govern how DTDs and external entities are processed.

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException; // catching
unsupported features
...

    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    try {
      // This is the PRIMARY defense. If DTDs (doctypes) are
disallowed, almost all XML entity attacks are prevented
      // Xerces 2 only - http://xerces.apache.org/xerces2-
j/features.html#disallow-doctype-decl
      String FEATURE = "http://apache.org/xml/features/disallow-
doctype-decl";
      dbf.setFeature(FEATURE, true);

      // If you can't completely disable DTDs, then at least do the
following:
      // Xerces 1 - http://xerces.apache.org/xerces-
j/features.html#external-general-entities
      // Xerces 2 - http://xerces.apache.org/xerces2-
j/features.html#external-general-entities
      FEATURE = "http://xml.org/sax/features/external-general-
entities";
      dbf.setFeature(FEATURE, false);

      // Xerces 1 - http://xerces.apache.org/xerces-
j/features.html#external-parameter-entities
      // Xerces 2 - http://xerces.apache.org/xerces2-
j/features.html#external-parameter-entities
      FEATURE = "http://xml.org/sax/features/external-parameter-
entities";
      dbf.setFeature(FEATURE, false);

      // and these as well, per Timothy Morgan's 2014 paper: "XML
Schema, DTD, and Entity Attacks" (see reference below)
      dbf.setXIncludeAware(false);
      dbf.setExpandEntityReferences(false);

      // And, per Timothy Morgan: "If for some reason support for
inline DOCTYPEs are a requirement, then
      // ensure the entity settings are disabled (as shown above) and
beware that SSRF attacks
      // (http://cwe.mitre.org/data/definitions/918.html) and denial
      // of service attacks (such as billion laughs or decompression
bombs via "jar:") are a risk."

      // remaining parser logic
      ...

        catch (ParserConfigurationException e) {
            // This should catch a failed setFeature feature
            logger.info("ParserConfigurationException was thrown. The
feature '" +
                        FEATURE +
                        "' is probably not supported by your XML
processor.");
            ...
        }
        catch (SAXException e) {
            // On Apache, this should be thrown when disallowing
DOCTYPE
            logger.warning("A DOCTYPE was passed into the XML
document");
            ...
        }
        catch (IOException e) {
            // XXE that points to a file that doesn't exist
            logger.error("IOException occurred, XXE may still
possible: " + e.getMessage());
            ...
        }
```

Xerces 1 Features:

- Do not include external entities by setting this feature to false.
- Do not include parameter entities by setting this feature to false.

Xerces 2 Features:

- Disallow an inline DTD by setting this feature to true.
- Do not include external entities by setting this feature to false.
- Do not include parameter entities by setting this feature to false.

StAX and XMLInputFactory

StAX parsers such as XMLInputFactory allow various properties and features to be set.

To protect a Java XMLInputFactory from XXE, do this:

- xmlInputFactory.setProperty(XMLInputFactory.SUPPORT_DTD, false); // This disables DTDs entirely for that factory

.NET

.NET 3.5

The following information for .NET are almost direct quotes from this great article on how to prevent XXE and XML Denial of Service in .NET: http://msdn.microsoft.com/en-us/magazine/ee335713.aspx.

In .NET Framework versions 3.5 and earlier, DTD parsing behavior is controlled by the Boolean ProhibitDtd property found in the System.Xml.XmlTextReader and System.Xml.XmlReaderSettings classes. Set this value to true to disable inline DTDs completely:

```
XmlTextReader reader = new XmlTextReader(stream);
reader.ProhibitDtd = true;
```

The default value of ProhibitDtd in XmlReaderSettings is true, but the default value of ProhibitDtd in XmlTextReader is false, which means that you have to explicitly set it to true to disable inline DTDs.

If you need DTD parsing enabled, but need to know how to do it safely, the above referenced MSDN article has detailed instructions on how to do that too.

iOS

libxml2

iOS includes the C/C++ libxml2 library described above, so that guidance applies if you are using libxml2 directly. However, the version of libxml2 provided up through iOS6 is prior to version 2.9 of libxml2 (which protects against XXE by default).

NSXMLDocument

iOS also provides an NSXMLDocument type, which is built on top of libxml2. However, NSXMLDocument provides some additional protections against XXE that aren't available in libxml2 directly. Per the 'NSXMLDocument External Entity Restriction API' section of: http://developer.apple.com/library/ios/#releasenotes/Foundation/RN-Foundation-iOS/Foundation_iOS5.html:

- iOS4 and earlier: All external entities are loaded by default.
- iOS5 and later: Only entities that don't require network access are loaded. (which is safer)

However, to completely disable XXE in an NSXMLDocument in any version of iOS you simply specify NSXMLNodeLoadExternalEntitiesNever when creating the NSXMLDocument.

PHP

Per the PHP documentation, the following should be set when using the default PHP XML parser in order to prevent XXE:

libxml_disable_entity_loader(true);

A description of how to abuse this in PHP is presented in a good SensePost article describing a cool PHP based XXE vulnerability that was fixed in Facebook.

4. Blind XPath Injection

Summary –

XPath is a type of query language that describes how to locate specific elements (including attributes, processing instructions, etc.) in an XML document. Since it is a query language, XPath is somewhat similar to Structured Query Language (SQL), however, XPath is different in that it can be used to reference almost any part of an XML document without access control restrictions. In SQL, a "user" (which is a term undefined in the XPath/XML context) may be restricted to certain databases, tables, columns, or queries. Using an XPATH Injection attack, an attacker is able to modify the XPATH query to perform an action of his choosing.

Blind XPath Injection attacks can be used to extract data from an application that embeds user supplied data in an unsafe way. When input is not properly sanitized, an attacker can supply valid XPath code that is executed. This type of attack is used in situations where the attacker has no knowledge about the structure of the XML document, or perhaps error message are suppressed, and is only able to pull once piece of information at a time by asking true/false questions(booleanized queries), much like Blind SQL Injection.

Test Example

The XPath attack pattern was first published by Amit Klein [1] and is very similar to the usual SQL Injection. In order to get a first grasp of the problem, let's imagine a login page that manages the authentication to an application in which the user must enter his/her username and password. Let's assume that our database is represented by the following XML file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
```

```
<user>
<username>gandalf</username>
<password>!c3</password>
<account>admin</account>
</user>
<user>
<username>Stefan0</username>
<password>w1s3c</password>
<account>guest</account>
</user>
<user>
<username>tony</username>
<password>Un6R34kb!e</password>
<account>guest</account>
</user>
</users>
```

An XPath query that returns the account whose username is "gandalf" and the password is "!c3" would be the following:

```
string(//user[username/text()='gandalf' and
password/text()='!c3']/account/text())
```

If the application does not properly filter user input, the tester will be able to inject XPath code and interfere with the query result. For instance, the tester could input the following values:

```
Username: ' or '1' = '1
Password: ' or '1' = '1
```

Looks quite familiar, doesn't it? Using these parameters, the query becomes:

```
string(//user[username/text()='' or '1' = '1' and password/text()=''
or '1' = '1']/account/text())
```

As in a common SQL Injection attack, we have created a query that always evaluates to true, which means that the application will authenticate the user even if a username or a password have not been provided. And as in a common SQL Injection attack, with XPath injection, the first step is to insert a single quote (') in the field to be tested, introducing a syntax error in the query, and to check whether the application returns an error message.

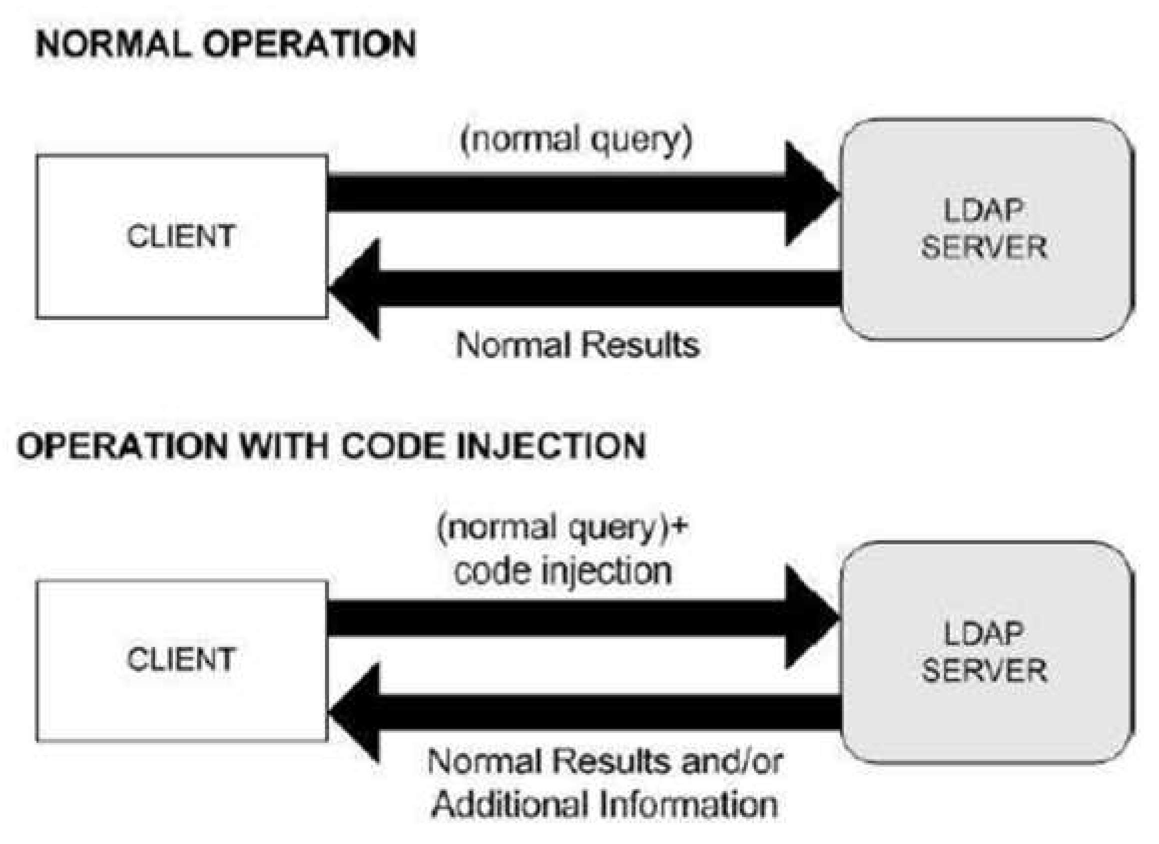
Mitigation –

As in a common SQL Injection attack, we have created a query that always evaluates to true, which means that the application will authenticate the user even if a username or a password have not been provided. And as in a common SQL Injection attack, with XPath injection, the first step is to insert a single quote (') in the field to be tested, introducing a syntax error in the query, and to check whether the application returns an error message.

5. LDAP injection

Summary –

LDAP Injection is an attack used to exploit web based applications that construct LDAP statements based on user input. When an application fails to properly sanitize user input, it's possible to modify LDAP statements using a local proxy. This could result in the execution of arbitrary commands such as granting permissions to unauthorized queries, and content modification inside the LDAP tree. The same advanced exploitation techniques available in SQL Injection can be similarly applied in LDAP Injection.


NORMAL OPERATION
(normal query)
CLIENT
LDAP SERVER
Normal Results
OPERATION WITH CODE INJECTION
(normal query)+ code injection
CLIENT
LDAP SERVER
Normal Results and/or Additional Information

Examples –

In a page with a user search form, the following code is responsible to catch input value and generate a LDAP query that will be used in LDAP database.

```
<input type="text" size=20 name="userName">Insert the username</input>
```

The LDAP query is narrowed down for performance and the underlying code for this function might be the following:

```
String ldapSearchQuery = "(cn=" + $userName + ")";
 System.out.println(ldapSearchQuery);
```

If the variable $userName is not validated, it could be possible accomplish LDAP injection, as follows:

- If a user puts “*” on box search, the system may return all the usernames on the LDAP base

- If a user puts “jonys) (| (password = *))”, it will generate the code bellow revealing jonys’ password (cn = jonys) (| (password = *))

Mitigation –

Incoming Data Validation - All client-supplied data needs to be cleaned of any characters or strings that could possibly be used maliciously. This should be done for all applications, not just those that use LDAP queries. Stripping quotes or putting backslashes in front of them is nowhere near enough. The best way to filter data is with a default-deny regular expression that includes only the type of characters that you want. For instance, the following regular expression will return only letters and numbers: *s/[^0-9a-zA-Z]//g*

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

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

Broken Authentication and Session Management

Description –

Attacker uses leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to impersonate users. Consider anonymous external attackers, as well as users with their own accounts, who may attempt to steal accounts from others. Also consider insiders wanting to disguise their actions.

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

The primary recommendation for an organization is to make available to developers:

1. **A single set of strong authentication and session management controls.** Such controls should strive to:

1. Meet all the authentication and session management requirements defined in OWASP's Application Security Verification Standard (ASVS) areas V2 (Authentication) and V3 (Session Management).

2. Have a simple interface for developers. Consider the ESAPI Authenticator and User APIs as good examples to emulate, use, or build upon.

2. Strong efforts should also be made to avoid XSS flaws which can be used to steal session IDs.

Below is the Authentication Verification Requirement for authentication:

V2.1	Verify all pages and resources require authentication except those specifically intended to be public (Principle of complete mediation).
V2.2	Verify all password fields do not echo the user's password when it is entered.
V2.3	Verify all authentication controls are enforced on the server side.
V2.4	Verify all authentication controls (including libraries that call external authentication services) have a centralized implementation.
V2.5	Verify all authentication controls fail securely to ensure attackers cannot log in.
V2.6	Verify password entry fields allow or encourage the use of passphrases, and do not prevent long passphrases or highly complex passwords being entered, and provide a sufficient minimum strength to protect against the use of commonly chosen passwords.

V2.7	Verify all account identity authentication functions (such as registration, update profile, forgot username, forgot password, disabled / lost token, help desk or IVR) that might regain access to the account are at least as resistant to attack as the primary authentication mechanism.
V2.8	Verify users can safely change their credentials using a mechanism that is at least as resistant to attack as the primary authentication mechanism.
V2.9	Verify that all authentication decisions are logged. This should include requests with missing required information, needed for security investigations.
V2.10	Verify that account passwords are salted using a salt that is unique to that account (e.g., internal user ID, account creation) and use bcrypt, scrypt or PBKDF2 before storing the password.
V2.11	Verify that credentials, and all other identity information handled by the application(s), do not traverse unencrypted or weakly encrypted links.
V2.12	Verify that the forgotten password function and other recovery paths do not reveal the current password and that the new password is not sent in clear text to the user.
V2.13	Verify that username enumeration is not possible via login, password reset, or forgot account functionality
V2.14	Verify there are no default passwords in use for the application framework or any components used by the application (such as “admin/password”).
V2.15	Verify that a resource governor is in place to protect against vertical (a single account tested against all possible passwords) and horizontal brute forcing (all accounts tested with the same password e.g. “Password1”). A correct credential entry should incur no delay. Both these governor mechanisms should be active simultaneously to protect against diagonal and distributed attacks.
V2.16	Verify that all authentication credentials for accessing services external to the application are encrypted and stored in a protected location (not in source code).
V2.17	Verify that forgot password and other recovery paths send a link including a time-limited activation token rather than the password itself. Additional authentication based on soft-tokens (e.g. SMS token, native mobile applications, etc.) can be required as well before the link is sent over.
V2.18	Verify that forgot password functionality does not lock or otherwise disable the account until after the user has successfully changed their password. This is to prevent valid users from being locked out.
V2.19	Verify that there are no shared knowledge questions/answers (so called "secret" questions and answers).
V2.20	Verify that the system can be configured to disallow the use of a configurable number of previous passwords.

V2.21	Verify re-authentication, step up or adaptive authentication, SMS or other two factor authentication, or transaction signing is required before any application-specific sensitive operations are permitted as per the risk profile of the application.

Below is the Session Management Verification Requirement:

V3.1	Verify that the framework's default session management control implementation is used by the application.
V3.2	Verify that sessions are invalidated when the user logs out.
V3.3	Verify that sessions timeout after a specified period of inactivity
V3.4	Verify that session's timeout after an administratively-configurable maximum time period regardless of activity (an absolute timeout).
V3.5	Verify that all pages that require authentication to access them have logout links.
V3.6	Verify that the session id is never disclosed other than in cookie headers; particularly in URLs, error messages, or logs. This includes verifying that the application does not support URL rewriting of session cookies.
V3.7	Verify that the session id is changed on login to prevent session fixation.
V3.8	Verify that the session id is changed upon re-authentication.
V3.9	Verify that only session ids generated by the application framework are recognized as valid by the application.
V3.10	Verify that authenticated session tokens are sufficiently long and random to withstand session guessing attacks.
V3.11	Verify that authenticated session tokens using cookies have their path set to an appropriately restrictive value for that site. The domain cookie attribute restriction should not be set unless for a business requirement, such as single sign on.
V3.12	Verify that authenticated session tokens using cookies sent via HTTP, are protected by the use of "HttpOnly".
V3.13	Verify that authenticated session tokens using cookies are protected with the "secure" attribute and a strict transport security header (such as Strict-Transport-Security: max-age=60000; includeSubDomains) are present.
V3.14	Verify that the application does not permit duplicate concurrent user sessions, originating from different machines.

Example Attack Scenarios –

Scenario #1: Airline reservations application supports URL rewriting, putting session IDs in the URL:

```
http://example.com/sale/saleitems?sessionid=268544541&dest=Hawaii
```

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

Scenario #2: Application's timeouts aren't set properly. User uses a public computer to access site. Instead of selecting "logout" the user simply closes the browser tab and walks away. Attacker uses the same browser an hour later, and that browser is still authenticated.

Scenario #3: Insider or external attacker gains access to the system's password database. User passwords are not properly hashed, exposing every users' password to the attacker.

Relevant Vulnerabilities –

1. Session Fixation

Description –

Session Fixation is an attack that permits an attacker to hijack a valid user session. The attack explores a limitation in the way the web application manages the session ID, more specifically the vulnerable web application. When authenticating a user, it doesn't assign a new session ID, making it possible to use an existent session ID. The attack consists of obtaining a valid session ID (e.g. by connecting to the application), inducing a user to authenticate himself with that session ID, and then hijacking the user-validated session by the knowledge of the used session ID. The attacker has to provide a legitimate Web application session ID and try to make the victim's browser use it.

There are several techniques to execute the attack; it depends on how the Web application deals with session tokens. Below are some of the most common techniques:

• **Session token in the URL argument:** The Session ID is sent to the victim in a hyperlink and the victim accesses the site through the malicious URL.

• **Session token in a hidden form field:** In this method, the victim must be tricked to authenticate in the target Web Server, using a login form developed for the attacker. The form could be hosted in the evil web server or directly in html formatted e-mail.

• Session ID in a cookie:

Client-side script

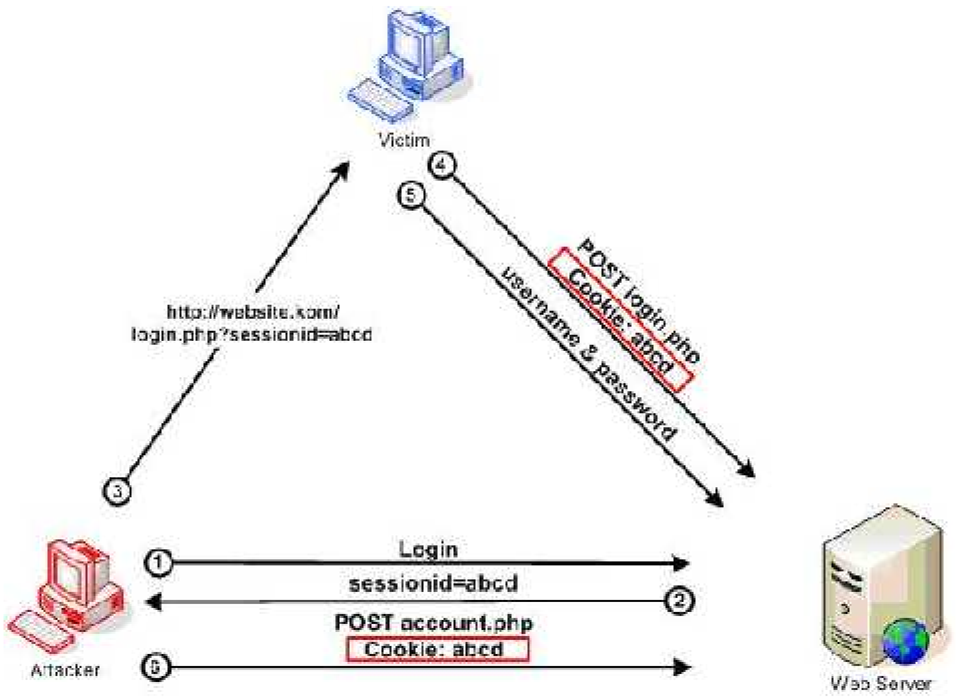
Most browsers support the execution of client-side scripting. In this case, the aggressor could use attacks of code injection as the XSS (Cross-site scripting) attack to insert a malicious code in the hyperlink sent to the victim and fix a Session ID in its cookie. Using the function document.cookie, the browser which executes the command becomes capable of fixing values inside of the cookie that it will use to keep a session between the client and the Web Application.

<META> tag

<META> tag also is considered a code injection attack, however, different from the XSS attack where undesirable scripts can be disabled, or the execution can be denied. The attack using this method becomes much more efficient because it's impossible to disable the processing of these tags in the browsers.

HTTP header response

This method explores the server response to fix the Session ID in the victim's browser. Including the parameter Set-Cookie in the HTTP header response, the attacker is able to insert the value of Session ID in the cookie and sends it to the victim's browser.



Mitigation –

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

Anti-Fixation in ASP

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

2. Session Hijacking

Description –

The Session Hijacking attack consists of the exploitation of the web session control mechanism, which is normally managed for a session token.

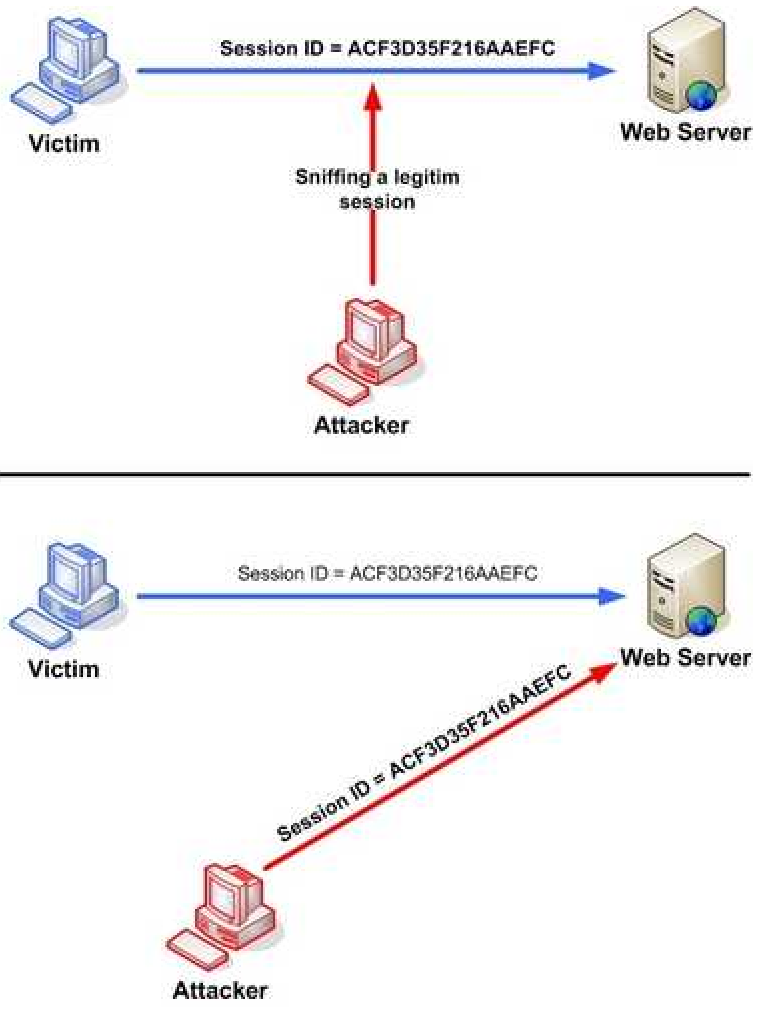
Because http communication uses many different TCP connections, the web server needs a method to recognize every user’s connections. The most useful method depends on a token that the Web Server sends to the client browser after a successful client authentication. A session token is normally composed of a string of variable width and it could be used in different ways, like in the URL, in the header of the http requisition as a cookie, in other parts of the header of the http request, or yet in the body of the http requisition.

The Session Hijacking attack compromises the session token by stealing or predicting a valid session token to gain unauthorized access to the Web Server.

The session token could be compromised in different ways; the most common are:

- Predictable session token;
- Session Sniffing;

In the example, as we can see, first the attacker uses a sniffer to capture a valid token session called "Session ID", then he uses the valid token session to gain unauthorized access to the Web Server.


Session ID = ACF3D35F216AAEFC
Victim
Web Server
Sniffing a legitim session
Attacker
Session ID = ACF3D35F216AAEFC
Victim
Web Server
Session ID = ACF3D35F216AAEFC
Attacker

- Client-side attacks (XSS, malicious JavaScript Codes, Trojans, etc);
- Man-in-the-middle attack
- Man-in-the-browser attack

A3-Cross-Site Scripting (XSS)

Description –

Attacker sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.

XSS is the most prevalent web application security flaw. XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content. There are two different types of XSS flaws: 1) Stored and 2) Reflected, and each of these can occur on the a) Server or b) on the Client.

Detection of most Server XSS flaws is fairly easy via testing or code analysis. Client XSS is very difficult to identify.

Mitigation –

Preventing XSS requires separation of untrusted data from active browser content.

1. The preferred option is to properly escape all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. See the OWASP XSS Prevention Cheat Sheet for details on the required data escaping techniques.

2. Positive or "whitelist" input validation is also recommended as it helps protect against XSS, but is not a complete defense as many applications require special characters in their input. Such validation should, as much as possible, validate the length, characters, format, and business rules on that data before accepting the input.

3. For rich content, consider auto-sanitization libraries like OWASP's AntiSamy or the Java HTML Sanitizer Project.

4. Consider Content Security Policy (CSP) to defend against XSS across your entire site.

XSS Prevention rules –

RULE #0 - Never Insert Untrusted Data except in Allowed Locations

The first rule is to **deny all** - don't put untrusted data into your HTML document unless it is within one of the slots defined in Rule #1 through Rule #5. The reason for Rule #0 is that there are so many strange contexts within HTML that the list of escaping rules gets very complicated. We can't think of any good reason to put untrusted data in these contexts. This includes "nested contexts" like a URL inside a javascript -- the encoding rules for those locations are tricky and dangerous. If you insist on putting untrusted data into nested contexts, please do a lot of cross-browser testing and let us know what you find out.

```
<script>...NEVER PUT UNTRUSTED DATA HERE...</script>   directly in a
script

 <!--...NEVER PUT UNTRUSTED DATA HERE...-->             inside an HTML
comment

 <div ...NEVER PUT UNTRUSTED DATA HERE...=test />       in an
attribute name

 <NEVER PUT UNTRUSTED DATA HERE... href="/test" />   in a tag name

 <style>...NEVER PUT UNTRUSTED DATA HERE...</style>   directly in CSS
```

Most importantly, never accept actual JavaScript code from an untrusted source and then run it. For example, a parameter named "callback" that contains a JavaScript code snippet. No amount of escaping can fix that.

RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content

Rule #2 is for putting untrusted data into typical attribute values like width, name, value, etc. This should not be used for complex attributes like href, src, style, or any of the event handlers like onmouseover. It is extremely important that event handler attributes should follow Rule #3 for HTML JavaScript Data Values.

```
<div attr=...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...>content</div>     inside UNquoted attribute

 <div attr='...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...'>content</div>   inside single quoted attribute

 <div attr="...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...">content</div>   inside double quoted attribute
```

xcept for alphanumeric characters, escape all characters with ASCII values less than 256 with the &#xHH; format (or a named entity if available) to prevent switching out of the attribute. The reason this rule is so broad is that developers frequently leave attributes unquoted. Properly quoted attributes can only be escaped with the corresponding quote. Unquoted attributes can be broken out of with many characters, including [space] % * + , - / ; < = > ^ and |.

RULE #3 - JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values

Rule #3 concerns dynamically generated JavaScript code - both script blocks and event-handler attributes. The only safe place to put untrusted data into this code is inside a quoted "data value." Including untrusted data inside any other JavaScript context is quite dangerous, as it is extremely easy to switch into an execution context with characters including (but not limited to) semi-colon, equals, space, plus, and many more, so use with caution.

```
<script>alert('...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...')</script>     inside a quoted string

 <script>x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'</script>
one side of a quoted expression

 <div onmouseover="x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...'"</div>  inside quoted event handler
```

RULE #4 - CSS Escape and Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values

Rule #4 is for when you want to put untrusted data into a stylesheet or a style tag. CSS is surprisingly powerful, and can be used for numerous attacks. Therefore, it's important that you only use untrusted data in a property **value** and not into other places in style data. You should stay away from putting untrusted data into complex properties like url, behavior, and custom (-moz-binding). You should also not put untrusted data into IE’s expression property value which allows JavaScript.

```
<style>selector { property : ...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...; } </style>     property value

 <style>selector { property : "...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE..."; } </style>   property value

 <span style="property : ...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...">text</span>       property value
```

Please note there are some CSS contexts that can never safely use untrusted data as input - **EVEN IF PROPERLY CSS ESCAPED**! You will have to ensure that URLs only start with "http" not "javascript" and that properties never start with "expression".

RULE #5 - URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values

Rule #5 is for when you want to put untrusted data into HTTP GET parameter value.

```
<a href="http://www.somesite.com?test=...ESCAPE UNTRUSTED DATA BEFORE
PUTTING HERE...">link</a >
```

Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the %HH escaping format. Including untrusted data in data: URLs should not be allowed as there is no good way to disable attacks with escaping to prevent switching out of the URL. All attributes should be quoted. Unquoted attributes can be broken out of with many characters including [space] % * + , - / ; < = > ^ and |. Note that entity encoding is useless in this context.

WARNING: Do not encode complete or relative URL's with URL encoding! If untrusted input is meant to be placed into href, src or other URL-based attributes, it should be validated to make sure it does not point to an unexpected protocol, especially Javascript links. URL's should then be encoded based on the context of display like any other piece of data. For example, user driven URL's in HREF links should be attribute encoded. For example:

```
String userURL = request.getParameter( "userURL" )
 boolean isValidURL = ESAPI.validator().isValidInput("URLContext",
userURL, "URL", 255, false);
 if (isValidURL) {
     <a href="<%=encoder.encodeForHTMLAttribute(userURL)%>">link</a>
 }
```

RULE #6 - Sanitize HTML Markup with a Library Designed for the Job

If your application handles markup -- untrusted input that is supposed to contain HTML -- it can be very difficult to validate. Encoding is also difficult, since it would break all the tags that are supposed to be in the input. Therefore, you need a library that can parse and clean HTML formatted text.

Bonus Rule #1: Use HTTPOnly cookie flag

Preventing all XSS flaws in an application is hard, as you can see. To help mitigate the impact of an XSS flaw on your site, OWASP also recommends you set the HTTPOnly flag on your session cookie and any custom cookies you have that are not accessed by any Javascript you wrote. This cookie flag is typically on by default in .NET apps, but in other languages you have to set it manually.

Bonus Rule #2: Implement Content Security Policy

There is another good complex solution to mitigate the impact of an XSS flaw called Content Security Policy. It's a browser side mechanism which allows you to create source whitelists for client side resources of your web application, e.g. JavaScript, CSS, images, etc. CSP via special HTTP header instructs the browser to only execute or render resources from those sources.

```
Content-Security-Policy: default-src: 'self'; script-src: 'self'
static.domain.tld
```

For example this CSP will instruct web browser to load all resources only from the page's origin and JavaScript source code files additionally from static.domain.tld.

Relevant Vulnerabilities

3. Reflected XSS –

Description –

Reflected Cross-site Scripting (XSS) occur when an attacker injects browser executable code within a single HTTP response. The injected attack is not stored within the application itself; it is non-persistent and only impacts users who open a maliciously crafted link or third-party web page. The attack string is included as part of the crafted URI or HTTP parameters, improperly processed by the application, and returned to the victim.

Reflected XSS are the most frequent type of XSS attacks found in the wild. Reflected XSS attacks are also known as non-persistent XSS attacks and, since the attack payload is delivered and executed via a single request and response, they are also referred to as first-order or type 1 XSS.

Mitigation –

The rules defined for XSS mitigation are same for mitigating the reflected XSS.

4. Stored XSS

Description –

Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information. Stored XSS is also sometimes referred to as Persistent or Type-I XSS.

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

Mitigation –

The rules defined for XSS mitigation are same for mitigating the reflected XSS.

5. DOM Based XSS

Description –

DOM Based XSS (or as it is called in some texts, "type-0 XSS") is an XSS attack wherein the attack payload is executed as a result of modifying the DOM "environment" in the victim's browser used by the original client side script, so that the client side code runs in an "unexpected" manner. That is, the page itself (the HTTP response that is) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment.

This is in contrast to other XSS attacks (stored or reflected), wherein the attack payload is placed in the response page (due to a server side flaw).

Mitigation –

1. Avoiding client side document rewriting, redirection, or other sensitive actions, using client side data. Most of these effects can be achieved by using dynamic pages (server side).

2. Analyzing and hardening the client side (Javascript) code. Reference to DOM objects that may be influenced by the user (attacker) should be inspected, including (but not limited to):

- `document.URL`
- `document.URLUnencoded`
- `document.location (and many of its properties)`
- `document.referrer`
- `window.location (and many of its properties)`

Note that a document object property or a window object property may be referenced syntactically in many ways - explicitly (e.g. `window.location`), implicitly (e.g. `location`), or via obtaining a handle to a window and using it (e.g. `handle_to_some_window.location`).

Special attention should be given to scenarios wherein the DOM is modified, either explicitly or potentially, either via raw access to the HTML or via access to the DOM itself, e.g. (by no means an exhaustive list, there are probably various browser extensions):

- Write raw HTML, e.g.:
    - `document.write(…)`
    - `document.writeln(…)`
    - `document.body.innerHtml=…`
- Directly modifying the DOM (including DHTML events), e.g.:
    - `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(…)`
- Replacing the document URL, e.g.:
    - `document.location=…` (and assigning to location's href, host and hostname)
    - `document.location.hostname=…`
    - `document.location.replace(…)`
    - `document.location.assign(…)`
    - `document.URL=…`
    - `window.navigate(…)`
- Opening/modifying a window, e.g.:
    - `document.open(…)`
    - `window.open(…)`
    - `window.location.href=…` (and assigning to location's href, host and hostname)
- Directly executing script, e.g.:
    - `eval(…)`
    - `window.execScript(…)`
    - `window.setInterval(…)`
    - `window.setTimeout(…)`

To continue the above example, an effective defense can be replacing the original script part with the following code, which verifies that the string written to the HTML page consists of alphanumeric characters only:

```
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
var name=document.URL.substring(pos,document.URL.length);
if (name.match(/^[a-zA-Z0-9]$/))
{
    document.write(name);
}
else
{
     window.alert("Security error");
}
</SCRIPT>
```

Such functionality can (and perhaps should) be provided through a generic library for sanitation of data (i.e. a set of Javascript functions that perform input validation and/or sanitation). The downside is that the security logic is exposed to the attackers - it is embedded in the HTML code. This makes it easier to understand and to attack it. While in the above example, the situation is very simple, in more complex scenarios wherein the security checks are less than perfect, this may come to play.

3. Employing a very strict IPS policy in which, for example, page welcome.html is expected to receive a one only parameter named “name”, whose content is inspected, and any irregularity (including excessive parameters or no parameters) results in not serving the original page, likewise with any other violation (such as an Authorization header or Referrer header containing problematic data), the original content must not be served. And in some cases, even this cannot guarantee that an attack will be thwarted.

A4-Insecure Direct Object References

Description –

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

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

Example Attack Scenarios –

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

```
String query = "SELECT * FROM accts WHERE account = ?";

PreparedStatement pstmt =
connection.prepareStatement(query , … );

pstmt.setString( 1, request.getParameter("acct"));

ResultSet results = pstmt.executeQuery( );
```

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

```
http://example.com/app/accountInfo?acct=notmyacct
```

Mitigation –

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

1. Use per user or session indirect object references. This prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource's database key, a drop down list of six resources authorized for the current user could use the numbers 1 to 6 to indicate which value the user selected. The application has to map the per-user indirect reference back to the actual database

key on the server. OWASP's ESAPI includes both sequential and random access reference maps that developers can use to eliminate direct object references.

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

A5-Security Misconfiguration

Description –

Attacker accesses default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.

Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, database, framework, and custom code. Developers and system administrators need to work together to ensure that the entire stack is configured properly. Automated scanners are useful for detecting missing patches, misconfigurations, use of default accounts, unnecessary services, etc.

The system could be completely compromised without you knowing it. All of your data could be stolen or modified slowly over time. Recovery costs could be expensive.

Example Attack Scenarios –

Scenario #1: The app server admin console is automatically installed and not removed. Default accounts aren't changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

Scenario #2: Directory listing is not disabled on your server. Attacker discovers she can simply list directories to find any file. Attacker finds and downloads all your compiled Java classes, which she decompiles and reverse engineers to get all your custom code. She then finds a serious access control flaw in your application.

Scenario #3: App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws. Attackers love the extra information error messages provide.

Scenario #4: App server comes with sample applications that are not removed from your production server. Said sample applications have well known security flaws attackers can use to compromise your server.

Mitigation –

The primary recommendations are to establish all of the following:

1. A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically (with different passwords used in each environment). This process should be automated to minimize the effort required to setup a new secure environment.

2. A process for keeping abreast of and deploying all new software updates and patches in a timely manner to each deployed environment. This needs to include all code libraries as well.

3. A strong application architecture that provides effective, secure separation between components.

4. Consider running scans and doing audits periodically to help detect future misconfigurations or missing patches.

A6-Sensitive Data Exposure

Description –

Attackers typically don't break crypto directly. They break something else, such as steal keys, do man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user's browser.

The most common flaw is simply not encrypting sensitive data. When crypto is employed, weak key generation and management, and weak algorithm usage is common, particularly weak password hashing techniques. Browser weaknesses are very common and easy to detect, but hard to exploit on a large scale. External attackers have difficulty detecting server side flaws due to limited access and they are also usually hard to exploit.

Failure frequently compromises all data that should have been protected. Typically, this information includes sensitive data such as health records, credentials, personal data, credit cards, etc.

Example Attack Scenarios –

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this means it also decrypts this data automatically when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text. The system should have encrypted the credit card numbers using a public key, and only allowed back-end applications to decrypt them with the private key.

Scenario #2: A site simply doesn't use SSL for all authenticated pages. Attacker simply monitors network traffic (like an open wireless network), and steals the user's session cookie. Attacker then replays this cookie and hijacks the user's session, accessing the user's private data.

Scenario #3: The password database uses unsalted hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password file. All of the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes.

Mitigation –

The full perils of unsafe cryptography, SSL usage, and data protection are well beyond the scope of the Top 10. That said, for all sensitive data, do all of the following, at a minimum:

1. Considering the threats you plan to protect this data from (e.g., insider attack, external user), make sure you encrypt all sensitive data at rest and in transit in a manner that defends against these threats.

2. Don’t store sensitive data unnecessarily. Discard it as soon as possible. Data you don’t have can’t be stolen.

3. Ensure strong standard algorithms and strong keys are used, and proper key management is in place. Consider using FIPS 140 validated cryptographic modules.

4. Ensure passwords are stored with an algorithm specifically designed for password protection, such as bcrypt, PBKDF2, or scrypt.

5. Disable autocomplete on forms collecting sensitive data and disable caching for pages that contain sensitive data.

A7-Missing Function Level Access Control

Description –

Attacker, who is an authorized system user, simply changes the URL or a parameter to a privileged function. Is access granted? Anonymous users could access private functions that aren't protected. Applications do not always protect application functions properly. Sometimes, function level protection is managed via configuration, and the system is misconfigured. Sometimes, developers must include the proper code checks, and they forget.

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

Such flaws allow attackers to access unauthorized functionality. Administrative functions are key targets for this type of attack.

Example Attack Scenarios –

Scenario #1: The attacker simply force browses to target URLs. The following URLs require authentication. Admin rights are also required for access to the admin_getappInfo page.

```
http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo
```

If an unauthenticated user can access either page, that's a flaw. If an authenticated, non-admin, user is allowed to access the admin_getappInfo page, this is also a flaw, and may lead the attacker to more improperly protected admin pages.

Scenario #2: A page provides an 'action' parameter to specify the function being invoked, and different actions require different roles. If these roles aren't enforced, that's a flaw.

Relevant Vulnerabilities –

1. Failure to Restrict URL Access

Description –

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

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

Some common examples of these flaws include:

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

Mitigation –

- Ensure the access control matrix is part of the business, architecture, and design of the application
- Ensure that all URLs and business functions are protected by an effective access control mechanism that verifies the user’s role and entitlements prior to any processing taking place. Make sure this is done during every step of the way, not just once towards the beginning of any multi-step process
- Perform a penetration test prior to deployment or code delivery to ensure that the application cannot be misused by a motivated skilled attacker
- Pay close attention to include/library files, especially if they have an executable extension such as .php. Where feasible, they should be kept outside of the web root. They should verify that they are not being directly accessed, e.g. by checking for a constant that can only be created by the library’s caller
- Do not assume that users will be unaware of special or hidden URLs or APIs. Always ensure that administrative and high privilege actions are protected
- Block access to all file types that your application should never serve. Ideally, this filter would follow the "accept known good" approach and only allow file types that you intend to serve, e.g., .html, .pdf, .php. This would then block any attempts to access log files, xml files, etc. that you never intend to serve directly.

- Keep up to date with virus protection and patches to components such as XML processors, word processors, image processors, etc., which handle user supplied data

A8-Cross-Site Request Forgery (CSRF)

Description –

Attacker creates forged HTTP requests and tricks a victim into submitting them via image tags, XSS, or numerous other techniques. If the user is authenticated, the attack succeeds.

CSRF takes advantage the fact that most web apps allow attackers to predict all the details of a particular action.

Because browsers send credentials like session cookies automatically, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones.

Detection of CSRF flaws is fairly easy via penetration testing or code analysis.

Example Attack Scenarios –

The application allows a user to submit a state changing request that does not include anything secret. For example:

```
http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243
```

So, the attacker constructs a request that will transfer money from the victim's account to the attacker's account, and then embeds this attack in an image request or iframe stored on various sites under the attacker's control:

```
<img
src="http://example.com/app/transferFunds?amount=1500&destinationAccount=attackersAcct#"
width="0" height="0" />
```

If the victim visits any of the attacker's sites while already authenticated to example.com, these forged requests will automatically include the user's session info, authorizing the attacker's request.

Mitigation –

Preventing CSRF usually requires the inclusion of an unpredictable token in each HTTP request. Such tokens should, at a minimum, be unique per user session.

1. The preferred option is to include the unique token in a hidden field. This causes the value to be sent in the body of the HTTP request, avoiding its inclusion in the URL, which is more prone to exposure.

2. The unique token can also be included in the URL itself, or a URL parameter. However, such placement runs a greater risk that the URL will be exposed to an attacker, thus compromising the secret token.
   OWASP’s CSRF Guard can automatically include such tokens in Java EE, .NET, or PHP apps. OWASP’s ESAPI includes methods developers can use to prevent CSRF vulnerabilities.
3. Requiring the user to re-authenticate, or prove they are a user (e.g., via a CAPTCHA) can also protect against CSRF.

A9-Using Components with Known Vulnerabilities

Description –

Attacker identifies a weak component through scanning or manual analysis. He customizes the exploit as needed and executes the attack. It gets more difficult if the used component is deep in the application.

Virtually every application has these issues because most development teams don't focus on ensuring their components/libraries are up to date. In many cases, the developers don't even know all the components they are using, never mind their versions. Component dependencies make things even worse.

The full range of weaknesses is possible, including injection, broken access control, XSS, etc. The impact could range from minimal to complete host takeover and data compromise.

Example Attack Scenarios –

Component vulnerabilities can cause almost any type of risk imaginable, ranging from the trivial to sophisticated malware designed to target a specific organization. Components almost always run with the full privilege of the application, so flaws in any component can be serious.

Every application using either of these vulnerable libraries is vulnerable to attack as both of these components are directly accessible by application users. Other vulnerable libraries, used deeper in an application, may be harder to exploit.

Mitigation –

Most component projects do not create vulnerability patches for old versions. Instead, most simply fix the problem in the next version. So upgrading to these new versions is critical. Software projects should have a process in place to:

1. Identify all components and the versions you are using, including all dependencies. (e.g., the versions plugin).
2. Monitor the security of these components in public databases, project mailing lists, and security mailing lists, and keep them up to date.
3. Establish security policies governing component use, such as requiring certain software development practices, passing security tests, and acceptable licenses.
4. Where appropriate, consider adding security wrappers around components to disable unused functionality and/ or secure weak or vulnerable aspects of the component.

A10-Unvalidated Redirects and Forwards

Description –

Attacker links to un-validated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site. Attacker targets unsafe forward to bypass security checks.

Applications frequently redirect users to other pages, or use internal forwards in a similar manner. Sometimes the target page is specified in an un-validated parameter, allowing attackers to choose the destination page.

Detecting unchecked redirects is easy. Look for redirects where you can set the full URL. Unchecked forwards are harder, because they target internal pages.

Such redirects may attempt to install malware or trick victims into disclosing passwords or other sensitive information. Unsafe forwards may allow access control bypass.

Example Attack Scenarios –

Scenario #1: The application has a page called "redirect.jsp" which takes a single parameter named "url". The attacker crafts a malicious URL that redirects users to a malicious site that performs phishing and installs malware.

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

Scenario #2: The application uses forwards to route requests between different parts of the site. To facilitate this, some pages use a parameter to indicate where the user should be sent if a transaction is successful. In this case, the attacker crafts a URL that will pass the application's access control check and then forwards the attacker to administrative functionality for which the attacker isn't authorized.

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

Mitigation –

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

1. Simply avoid using redirects and forwards.
2. If used, don't involve user parameters in calculating the destination. This can usually be done.

3. If destination parameters can’t be avoided, ensure that the supplied value is valid, and authorized for the user.
   It is recommended that any such destination parameters be a mapping value, rather than the actual URL or portion of the URL, and that server side code translate this mapping to the target URL.
   Applications can use ESAPI to override the `sendRedirect()` method to make sure all redirect destinations are safe.

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