OWASP Top 10

Conor Rynne* Oihana Gallastegi †

April 3, 2024

Introducción

Bienvenidos/as a OWASP Barcelona. Barcelona cuenta con una comunidad floreciente de start-ups tecnológicas, llena de personas con ideas brillantes y con las habilidades necesarias para llevarlas a cabo.

Pero ser capaz de hacer cosas y ser capaz de hacerlas bien pueden ser dos cosas muy distintas. Por supuesto, lo que has diseñado quizás funcione técnicamente, pero podría hacerlo de tal modo que ponga en riesgo tu empresa.

Esto puede comprometer la base de datos de tus clientes. Puede permitir que tus clientes obtengan tus servicios de forma gratuita. Incluso puede facilitar que algunas personas perjudiquen tu servicio o roben datos de pago.

Esta conferencia trata de los errores y peligros más comunes de las aplicaciones web en la actualidad, y de cómo los puedes evitar en tus proyectos. En este caso nos centraremos principalmente en aplicaciones web; las aplicaciones móviles y las APIs web cuentan con sus propias consideraciones.

*Autor original [inglés]
†Traducción [castellano]

Contents

1 A01:2021 – Pérdida de Control de Acceso

Los controles de acceso existen para asegurar que los usuarios solo puedan hacer lo que se supone que tienen permitido hacer y no actúen fuera de estos límites. Al fin y al cabo, no querrás que un usuario pueda ver la información privada de otro, ¡o incluso borrarla del todo de tu servicio!

A la hora de decidir qué se supone que deben poder hacer los usuarios, hay dos reglas de oro:

1. **Principio de mínimo privilegio** – Aquí es donde defines todas las capacidades que un usuario debería tener y le asignas únicamente los permisos para realizar dichas acciones. Por ejemplo, en el caso de un escritor de artículos, se le pueden dar permisos *solo* para crear y modificar sus propios artículos.

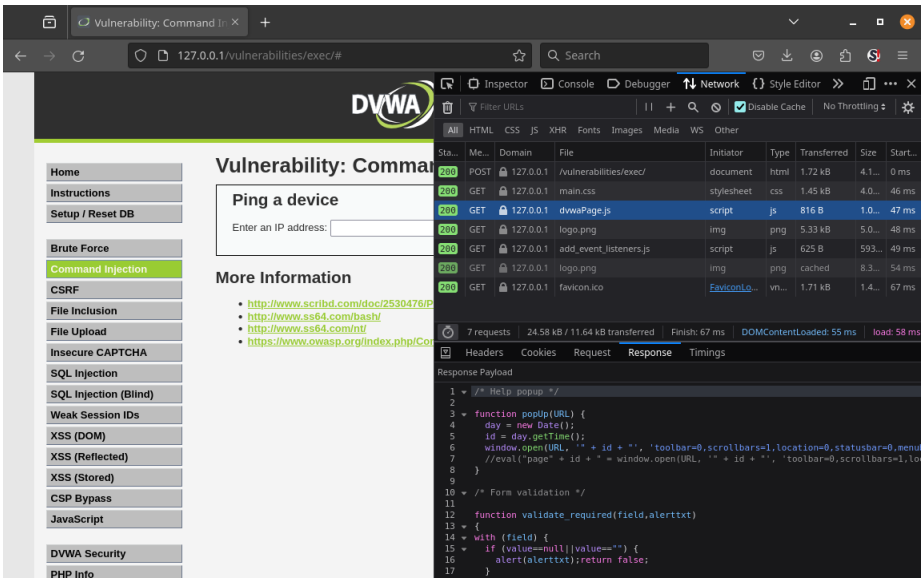
2. **Denegación por defecto** – Si a un usuario no se le asigna un permiso de forma explícita, la aplicación debería actuar como si este usuario directamente no lo tuviera.

Cumplimiento

Incluso si no tienes estos permisos por escrito, es importante que se implemente su cumplimiento. No sirve de nada decir "solo los administradores pueden acceder a las funciones administrativas" si tus funciones administrativas no cuentan con un sistema que verifique que el usuario es un administrador.

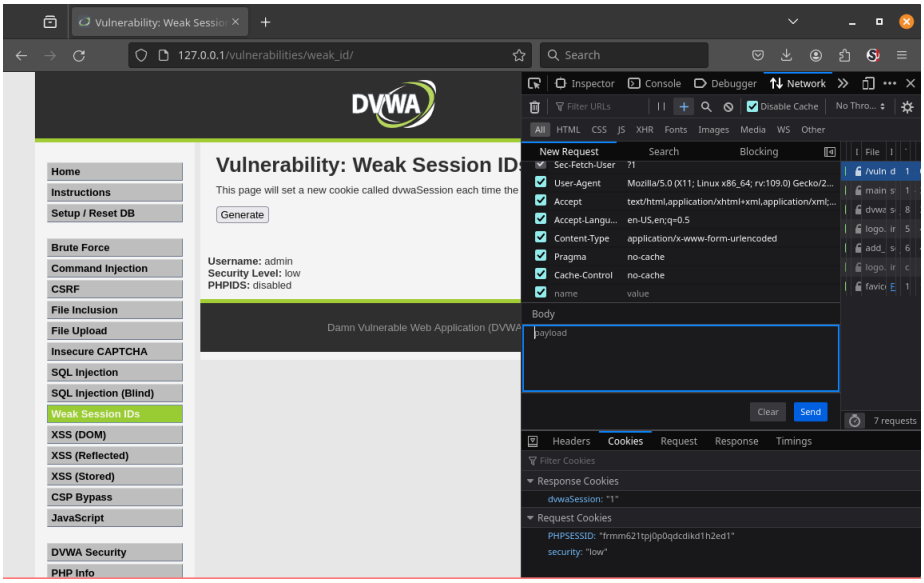
Dicho esto, hay formas buenas y malas de garantizar este cumplimiento. Debemos tener en cuenta otras dos reglas de oro:

1. **No envíes datos innecesarios al usuario. Todo se puede leer.** - Incluso si has ejecutado código JavaScript localmente para asegurarte de que solo los datos que el usuario se supone que debe ver se muestren en el navegador web, el usuario puede ver todos los datos sin procesar que le envíes si así lo desea. Esta capacidad se incluye en todos los navegadores web modernos como parte de sus consolas de desarrollador.



El visor de peticiones en Mozilla Firefox

2. **No confíes ciegamente en los datos enviados por el usuario. Todo se puede manipular.** – Del mismo modo, que tú no des la opción de ajustar ciertos datos en tu aplicación web no significa que el usuario no tenga la capacidad de hacerlo manualmente. Esta capacidad también se incluye en todos los navegadores web modernos como parte de sus consolas de desarrollador.



El editor de peticiones en Mozilla Firefox

CORS en HTTP

Una política de Intercambio de Recursos de Origen Cruzado (CORS) en HTTP puede ayudarte a restringir los sitios web que tu aplicación puede usar.

Sin embargo, no impide el acceso directo, ni es particularmente eficaz contra clientes HTTP que no son navegadores web, como los cURL, donde los usuarios pueden editar fácilmente las cabeceras HTTP enviadas en las peticiones.

Las políticas CORS se envían al navegador mediante cabeceras HTTP, específicamente la cabecera `AccessControl-Allow-Origin` header. El acceso a verbos HTTP específicos (como GET y POST) y la restricción de otras cabeceras HTTP también se pueden gestionar con las cabeceras `Access-Control-Allow-Methods` y `Access-Control-Allow-Headers` respectivamente.

En cuanto a cómo construir una política segura, las reglas de oro son otra vez el **Principio de mínimo privilegio** y la **Denegación por defecto**. Esto significa que deberás permitir el acceso únicamente a los sitios que necesitan utilizar esos recursos y denegar todo lo demás.

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://www.mydomain.com
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: Authorization, Content-Type
Content-Type: application/json
...
```

Un ejemplo de respuesta HTTP con cabeceras de la política CORS

Cookies y datos de sesión

Muchos sitios web utilizan mecanismos de almacenamiento local como cookies y bases de datos locales. Uno de los usos más comunes es almacenar un token de sesión para los usuarios conectados. Cuando un navegador web envía un token de sesión y una página web identifica que ese token pertenece a un usuario conectado llamado X, la página web realizará el resto de su procesamiento sabiendo que el usuario del navegador web es el usuario llamado X.

Para que este sea un medio seguro de gestión de sesiones, es imprescindible que el token de sesión no pueda adivinarse fácilmente. Es decir, necesita ser un valor completamente aleatorio que un atacante no pueda adivinar. La mayoría de los frameworks modernos se encargan de esto por ti, y si tienes acceso a esto, es mejor usarlo; los mecanismos proporcionados por los frameworks de aplicaciones web suelen estar mucho más probados y reforzados que cualquier cosa que puedas crear de forma realista.

Pero si eso no es una opción y necesitas implementarlo tú mismo, es importante utilizar un buen generador de números pseudoaleatorios criptográficamente seguros como base para la generación de tokens de sesión.

El uso de datos predecibles o adivinables, como la hora o los identificadores de usuario, plantea muchos problemas. Dado que los identificadores de usuario son estáticos y la hora es predecible (es decir, un minuto después de las 12:47 serán las 12:48), estos no son buenos candidatos para utilizarlos como tokens de sesión, incluso cuando se someten a un algoritmo de hash.

Los tokens de Java Web (JWT) cifrados pueden ser útiles en este caso, ya que el cifrado JWT puede utilizarse como verificación de que los datos no han sido manipulados. Sin embargo, ten en cuenta que el cifrado JWT convencional es útil *only* para fines de verificación. Debido a que también incluye una copia en texto claro (es decir, no cifrado) de los datos, la información no está protegida de su lectura (el usuario la puede leer).

Prevención

El resumen de los métodos que se pueden utilizar para evitar estos problemas es:

- Asigna a los usuarios el número mínimo de permisos que necesitan para sus tareas legítimas.
- Deniega explícitamente el acceso a todo lo demás.
- No envíes datos innecesarios al usuario.
- No confíes ciegamente en los datos enviados por el usuario.
- Para la gestión de sesión, no crees tu propia solución cuando no es necesario.

2 A02:2021 – Fallas Criptográficas

El cifrado se ha convertido en una parte importante de la mayoría de nuestras interacciones en línea. Nos asegura que nuestra información no está siendo interceptada y que las comunicaciones no están siendo modificadas. Pero como ocurre con todo lo relacionado con la ciberseguridad, hay formas correctas y muchas, muchas formas incorrectas de hacer las cosas.

Transmisión Inapropiada en Texto Claro

La primera forma de hacer mal el cifrado es directamente no hacerlo, especialmente en lo que respecta a las comunicaciones que implican información sensible. Lo ideal hoy en día es que todas las páginas estén cifradas, pero cuando se trata de datos de acceso o de información personal identificable (PII) se convierte en un requisito indiscutido, a menudo por ley.

Protocolos como HTTP, FTP y Telnet no admiten cifrado. En lugar de estos, deben utilizarse homólogos más seguros como HTTPS, SFTP y SSH. El tráfico SMTP no siempre está cifrado, por lo que hay que asegurarse de que el cifrado es obligatorio en este tipo de servicios.

En el caso de los navegadores web, es aconsejable imponer el uso de tráfico HTTPS cifrado mediante el uso de HTTP Strict Transport Security (HSTS). Esto puede garantizar que, si no se puede realizar una conexión cifrada, la conexión fallará.

Almacenamiento Inapropiado en Texto Claro

Cifrar el tráfico está muy bien, pero si tus sistemas sufren una brecha o te roban los discos duros, almacenar datos confidenciales en texto claro y sin cifrar hará aún más daño.

El cifrado de disco completo (FDE) cifra todo el disco y requiere una clave de cifrado para poder utilizarlo. Esto es bastante útil contra el robo físico, pero no tanto en un sistema ya en ejecución; una vez que se conecta y se da la clave, cualquier usuario o proceso en el dispositivo actualmente en ejecución puede acceder a los datos en la unidad cifrada FDE, incluyendo programas maliciosos o bases de datos SQL que cumplan con consultas SQL maliciosas.

El cifrado de bases de datos cifra una base de datos SQL completa. Si bien esto puede proteger contra otros programas maliciosos en el sistema, todavía cederá los datos al cumplir con una consulta SQL maliciosa realizada por un atacante.

Cifrar los datos *dentro* de la base de datos dará mejores resultados a la hora de protegerse contra ataques como la inyección SQL. A diferencia de los dos últimos esquemas de cifrado, no hay una manera de hacer esto con un solo clic, sino que es algo incorporado en tu aplicación.

Sin embargo, cuando se trata de credenciales y contraseñas, es mejor evitar el uso del cifrado y utilizar en su lugar un algoritmo de hash. La principal diferencia entre

un algoritmo de cifrado y un algoritmo de hash es que con el segundo es imposible recuperar los datos originales. Sin embargo, los algoritmos de hash pueden utilizarse para comprobar si dos bits de datos son idénticos; con un buen algoritmo de hash, no hay dos bits de datos no idénticos que produzcan el mismo hash, el resultado cuando los datos se procesan mediante este algoritmo.

Esto convierte a los hashes en un método ideal para almacenar credenciales. En lugar de almacenar las contraseñas, se almacena el hash y se compara con él la contraseña hash de un intento de inicio de sesión reciente. Aun así, esto todavía puede traer problemas si lo dejas tal cual.

Un algoritmo de hash produce siempre el mismo hash cuando se le da un bit idéntico de datos, y lo mismo ocurre con las contraseñas. Por eso se intenta crear diccionarios de hashes que correspondan a contraseñas determinadas. Estos diccionarios se denominan rainbow tables (tablas arcoíris) y facilitan la obtención de contraseñas a partir de hashes. Para combatir esto, es una práctica común insertar una pieza consistente de código adicional en la contraseña antes de hacer el hash.

Esto se llama "sal" y puedes incluso almacenarla junto a la contraseña del usuario en la base de datos SQL. No obstante, para mayor seguridad, también puedes añadir una sal que se origine fuera de la base de datos, de forma que alguien que solo tenga acceso a la base de datos no pueda acceder a ella.

Algoritmos y Protocolos Débiles

Un fallo común que tienen muchos sitios es que utilizan o aceptan algoritmos y protocolos anticuados a la hora de configurar el tráfico cifrado. Los algoritmos y protocolos antiguos suelen presentar vulnerabilidades que facilitan enormemente el descifrado no autorizado de los mensajes.

Aunque ofrecer soporte para protocolos y algoritmos antiguos puede parecer una buena idea para la compatibilidad con dispositivos antiguos, en la práctica permite a los atacantes obligar a los usuarios a utilizar los algoritmos y protocolos más débiles. Esto se debe a que con SSL y TLS, los dos protocolos más usados para el tráfico cifrado en redes, las primeras comunicaciones consisten en decidir qué algoritmos de cifrado utilizar, por lo que no están cifradas y son susceptibles de manipulación. Si un atacante puede forzar las comunicaciones para que utilicen algoritmos como RC4, ¡el cifrado podría romperse en cuestión de minutos!

Confidencialidad de Claves

Puede parecer obvio, pero las claves secretas de cifrado son más útiles cuando son... secretas. Cuanta más gente las conozca, más peligroso puede ser el uso de estas claves. Del mismo modo que no querrías usar 'admin:password' en tu router, tampoco querrás usar claves de cifrado por defecto en ninguno de tus dispositivos o productos.

Además, si tu software desarrollado, ya sea privado o público, utiliza claves de cifrado en algún punto, te interesará mantenerlas fuera de tus repositorios de código.

Muchos servicios de repositorio de código, como Azure DevOps y GitHub, ofrecen opciones para almacenar por separado las claves de cifrado necesarias.

Almacenamiento en Caché

Muchos navegadores y servicios web almacenan datos en caché para evitar tener que recargarlos. Sin embargo, el almacenamiento en caché de datos personales sensibles puede ser muy problemático, ya que puede permitir a otros usuarios (por ejemplo, de un ordenador público) ver esta información sensible de un usuario anterior.

Esto se puede controlar mediante el uso de cabeceras HTTP. Específicamente, para desactivar el almacenamiento en caché, tendrás que usar:

```
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: 0
```

Las cabeceras HTTP que se usan para desactivar el almacenamiento en caché

3 A03:2021 – Inyección

Imagina por un momento que tuvieras algo como un GameShark que, en lugar de modificar tus videojuegos, modificara las cosas que te rodean en la vida real. Podría fijar los precios de una tienda a tu antojo, hacer que las vallas publicitarias mostraran mágicamente a todo el mundo lo que tú quisieras, hacer que una empresa te diera toda su base de datos de tarjetas de crédito de clientes o, si quisieras, podría básicamente paralizar un negocio que no te gustara.

Contra las aplicaciones que no se defienden adecuadamente, los ataques de inyección *son* ese dispositivo GameShark. Los ataques de inyección permiten a un atacante insertar su propio código en tus aplicaciones, y los efectos pueden ir desde la modificación de los parámetros del sitio (como los precios) o la divulgación de datos sensibles, hasta el control absoluto de tus servidores web e incluso clientes.

Estos ataques suelen funcionar porque la aplicación que ejecuta el código, ya sea tu servidor SQL o el navegador web de un cliente, no puede diferenciar entre el código legítimo y los datos introducidos por el usuario.

Existen muchos tipos de ataques de inyección, por lo que repasaremos los más comunes.

Secuencia de Comandos en Sitios Cruzados

La Secuencia de Comandos en Sitios Cruzados (XSS) es el proceso de inyectar código JavaScript en una aplicación web. Este código se ejecuta en la máquina del cliente. Si bien esto puede no parecer inmediatamente peligroso para el servidor, puede ser extremadamente peligroso para tu reputación como empresa; en este punto, el atacante tiene un control casi completo sobre lo que tu usuario puede ver o con lo que puede interactuar en la pantalla.

Si tu almacenamiento de sesión no está protegido contra el acceso de JavaScript, el atacante puede incluso pasar desapercibido y enviarse los datos de sesión del cliente a sí mismo, lo que le permite iniciar sesión en tu aplicación como el usuario afectado sin una contraseña.

La principal defensa contra esto es el filtrado de los datos introducidos. El filtro sustituye caracteres especiales por sus equivalentes en notación HTML. Así, ">" se convierte en ">" en el código y aparecería como ">" en el navegador web, pero sin ninguna de las propiedades que harían que el navegador lo confundiera con parte del código HTML real. Sin embargo, esto es mucha simplificación y el proceso de filtrado es muy complejo debido a la gran cantidad de posibles derivaciones disponibles, por lo que se recomienda utilizar la función de filtrado de tu framework web o una biblioteca de software en su lugar.

Inyección de Plantillas del Lado del Servidor

Muchos frameworks modernos de aplicaciones web vienen con un motor de plantillas. Por lo general, se supone que el archivo de plantilla, que contiene el código HTML

estático y los marcadores de posición para el contenido dinámico, se carga desde un archivo, se genera el contenido dinámico y, a continuación, el motor de plantillas lo añade al archivo final que se entrega al cliente.

Los problemas surgen cuando procesos distintos al motor de plantillas realizan modificaciones en la plantilla base. Si se añade contenido generado dinámicamente a la plantilla base antes de que el motor de plantillas pueda trabajar, un atacante puede insertar su propio código para que el motor de plantillas lo ejecute. Dependiendo de las tecnologías subyacentes involucradas, esto puede otorgar a un atacante capacidades de Ejecución Remota de Código. Aquí es donde pueden controlar efectivamente tu servidor.

Las pruebas para detectar vulnerabilidades de Inyección de Plantillas del Lado del Servidor (SSTI) varían en función de las tecnologías subyacentes utilizadas, pero se puede encontrar una buena colección en `https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection`.

La mejor defensa contra esto es no permitir que ningún proceso que no sea el motor de plantillas inserte contenido dinámico en las plantillas.

Inyección SQL

Esta técnica se considera un clásico entre los atacantes y la popularizó el grupo hacktivista Anonymous. Los comandos SQL, o consultas, son órdenes escritas, no muy diferentes de la línea de comandos. Por ejemplo, una comprobación básica de contraseñas podría tener este aspecto:

```
SELECT username FROM users WHERE username = '<inputted username>' AND
    password = '<the submitted password, hashed>'
```

Modelo de código SQL simple para iniciar sesión

donde el servidor web y la tecnología de páginas web dinámicas añadirían el nombre del usuario real y la contraseña cifrada obtenida del usuario. Si el usuario introdujera las credenciales para el usuario "bob" y la contraseña "hunter2", el comando tendría el siguiente aspecto:

```
SELECT username FROM users WHERE username = 'bob' AND password = '84
    ca85078d6fa3a9b01dae0242938a9b71c9c6920f8d790505cad7a7' # The SHA2
    -224 hash for 'hunter2'
```

Código SQL para el usuario "bob" con la contraseña "hunter2"

Pero, ¿y si yo fuera un usuario malicioso simplemente intentando acceder a la cuenta de Bob? Entonces podría poner mi nombre de usuario como:

```
bob' WHERE 1=1;#
```

Y el código tendría este aspecto:

```
1 SELECT username FROM users WHERE username = 'bob' WHERE 1=1;#' AND
      password = '<irrelevant>'
```

Esencialmente, lo que he hecho aquí es reescribir la consulta SQL para ignorar completamente la contraseña. Como referencia, el carácter ";" termina una instrucción en SQL y el carácter "#" se considera un marcador de comentario; cualquier cosa a la derecha de ese símbolo será ignorado por el servidor SQL.

Esto funciona porque el servidor web está rellenando ciegamente los espacios en blanco de la consulta SQL directamente con nuestras entradas, y el servidor SQL no puede diferenciar entre lo que es código y lo que es entrada del usuario. Por este motivo, nunca utilizamos consultas directas con entradas del usuario, sino consultas parametrizadas. Algunos servidores SQL le dan nombres diferentes, pero la idea general es que se trata de una consulta almacenada que acepta parámetros. A diferencia del enfoque de consulta directa, cualquier dato pasado como parámetro se marca como dato de entrada y nunca se ejecuta como código a menos que lo utilices en un comando `eval()` (¡no lo hagas!).

Cómo arreglarlo

- Filtra todas las entradas. No confíes ciegamente en cualquier entrada del usuario.
- Si tu framework web tiene la función de plantillas, asegúrate de que su motor de plantillas sea el único que inserte código dinámico en tus páginas.
- Cuando realices consultas a la base de datos con datos de usuario, utiliza siempre consultas parametrizadas.

4 A04:2021 – Diseño Inseguro

El diseño inseguro es un problema mucho más nebuloso. Básicamente, se debe a que las aplicaciones se desarrollan y diseñan en torno a procesos inseguros o sin pensar demasiado en la seguridad de los datos.

El problema de hacer las cosas de esta manera es que la seguridad debería formar parte del proceso de diseño, y tratar de implementar la seguridad a posteriori es caro, lleva mucho tiempo y es menos eficaz.

Un ejemplo de esto podría ser que quisieras recoger datos de tarjetas, pero luego te olvidaras completamente de poner cualquier tipo de disposiciones para proteger dichos datos de tarjetas. Así que construyes tu aplicación sin tener en cuenta la seguridad necesaria, y luego te das cuenta de que tienes que replantear toda tu arquitectura, al menos para cumplir con el PCI DSS, el principal estándar de seguridad que rige el almacenamiento de datos de tarjetas. Además, si almacenas datos de tarjetas sin cumplir la norma PCI DSS, te expones a grandes sanciones, ¡incluso si no sufres ningún ataque!

Es imprescindible tener en cuenta las necesidades de seguridad de tus aplicaciones en todas las fases de diseño y desarrollo. Considera cuáles son tus amenazas, entiende tus requisitos de seguridad, crea procesos para cumplir con esos requisitos y utilízalos en tus aplicaciones.

Como puedes ver, hay más cosas de las que preocuparse más allá de que simplemente te ataquen. Si no se tiene el debido cuidado con la información personal de los clientes, también puede haber consecuencias legales: la filtración de información personal de los clientes puede acarrear grandes multas, cortesía de la legislación RGPD. Implementar pagos con tarjeta sin cumplir con el PCI DSS dañará tus relaciones con Visa y MasterCard, hasta el punto de que ya no te dejarán aceptar sus tarjetas en tu procesador de pagos. Y todo esto sin entrar en el resto de leyes de privacidad de datos de otros países en los que puedas estar operando.

Si estás creando tu propio software, deberías adoptar un Ciclo de Desarrollo de Software Seguro (SSDLC). La Fundación OWASP ha creado el Modelo de Madurez para el Aseguramiento del Software (SAMM) para ayudarte en esta tarea. Encontrarás más información en `https://owaspsamm.org/`.

Las aplicaciones en recursos compartidos deberían ser de recursos limitados para que, en caso de ataque, la aplicación no desconecte otros servicios. También puedes considerar el uso de equilibradores de carga e instalaciones redundantes.

5 A05:2021 – Configuración de Seguridad Incorrecta

Así que ya tenemos nuestro producto y nuestros nuevos juguetitos; el siguiente paso es configurarlos. Por desgracia, esto puede ser más fácil de decir que de hacer, ya que una configuración adecuada requiere una comprensión de las capacidades del software/dispositivo. La configuración por defecto de la mayoría de los softwares/dispositivos enfocados al público general es dar prioridad a la operatividad y la facilidad de uso sobre la seguridad. Esto puede dar lugar a una serie de problemas:

- Funciones innecesarias activadas o instaladas: Vale la pena recordar que cuantas más funciones tengas activas y más servicios estén escuchando en los puertos, más posibles vectores de ataque puede tener un atacante.

- Credenciales por defecto: Suelen estar bien documentadas y/o son fáciles de adivinar.

- Falta del hardening de seguridad adecuado: Depende de ti asegurarte de que solo los servicios y características necesarios se estén ejecutando y de que estos hayan sido protegidos de la mejor manera posible, incluyendo la prueba de las configuraciones de seguridad.

Las herramientas para desarrolladores también pueden ser problemáticas en lo que respecta a sus valores predeterminados, ya que tienden a dar prioridad a ayudar a los desarrolladores a detectar problemas proporcionando mucha información. Esto está muy bien en un entorno de desarrollo, pero en producción, esta misma información puede ser utilizada por un atacante para tratar de entrar en tus sistemas.

- Asegúrate de que los modos de depuración están desactivados en producción, y de que los mensajes de error proporcionan la menor información posible.

- Asegúrate de que los entornos de desarrollo están configurados para conectarse a repositorios de código y otros sistemas necesarios de forma segura.

- Asegúrate de que los frameworks de aplicaciones web están configurados con estándares seguros, desactivando todos los modos de depuración, sustituyendo las claves y credenciales SSL predeterminadas y desactivando cualquier servicio y función innecesarios.

El proceso de hardening se debe documentar para futuras referencias. Incluso si no utilizas ese producto concreto en el futuro, debería servirte de guía para la configuración segura de un dispositivo similar.

6 A06:2021 – Componentes Vulnerables y Desactualizados

El software está escrito por humanos y, como todo lo hecho por humanos, puede tener sus fallos. Algunos de estos fallos pueden tener consecuencias para la ciberseguridad, desde la revelación de información hasta el control total de una máquina por parte de un atacante. Por lo tanto, es de vital importancia que las organizaciones cuenten con un procedimiento regular de aplicación de parches en todos los dispositivos que utilizan. Desde estaciones de trabajo y servidores hasta routers.

Esto también se aplica al software de creación propia. A menos que estés construyéndolo todo desde cero, probablemente estés utilizando algún tipo de biblioteca o intérprete de terceros para ayudar a crear y ejecutar el software. Las bibliotecas de terceros tienen debilidades como cualquier otro tipo de software, y es probable que tu aplicación herede dichas debilidades si está utilizando una versión vulnerable de la biblioteca. Sin embargo, mantener las librerías actualizadas no es tan sencillo, ya que estas también cambian su sintaxis y funciones, lo que requiere que también actualices el código de tu aplicación para asegurarte de que funciona con las versiones más recientes.

7 A07:2021 – Fallas de Identificación y Autenticación

La protección de los procesos para autenticar usuarios es de vital importancia para la protección de tu aplicación y sus usuarios. Las fallas comunes pueden incluir:

- Falta de protección contra el relleno de credenciales: el proceso de tomar una lista de credenciales conocidas, de lugares como una filtración de otro sitio web, y probarlas en otras aplicaciones como la tuya propia.
- Permitir el uso de ataques de fuerza bruta, donde un atacante intenta adivinar la contraseña de un usuario o lista de usuarios muy rápidamente.
- Permitir el uso de contraseñas débiles.
- No cifrar el tráfico ni utilizar funciones hash en las contraseñas correctamente.
- No poseer autenticación multi-factor o que la implementada sea ineficaz.
- Exponer el identificador de sesión en la URL.
- No invalidar correctamente los tokens de sesión o autenticación. Se deben invalidar al cerrar la sesión o después de un largo período de tiempo.

Las soluciones para estos problemas incluyen:

- Implementar la autenticación multi-factor. Puedes utilizar contraseñas de un solo uso basadas en el tiempo a través de una aplicación de autenticación como Google Authenticator en un smartphone. Esto evitará que los atacantes puedan realizar automáticamente ataques de relleno de credenciales.
- Bloquear temporalmente una cuenta después de varios intentos incorrectos: esto evitará o dificultará en gran medida que los atacantes puedan utilizar herramientas automatizadas para intentar adivinar contraseñas.
- Garantizar que el registro, las vías de recuperación de credenciales y los puntos finales de la API estén reforzados contra los ataques de enumeración de cuentas asegurándonos de que los mensajes de fallo de autenticación no revelen la presencia de esa cuenta.
- Registrar todos los fallos y alertar a los administradores si se sospecha de posibles ataques.
- Utilizar las herramientas integradas de gestión de sesiones de los frameworks de tus aplicaciones web.

8 A08:2021 – Fallas en el Software y en la Integridad de los Datos

Las fallas de integridad del software y los datos se producen cuando hay deficiencias de seguridad en la forma en que se gestionan elementos como las dependencias, los datos críticos, las actualizaciones de software y los procesos de construcción.

Si, por ejemplo, tu aplicación depende de dependencias alojadas por un tercero que no es de confianza, alguien que controle esa dependencia podría inyectar código malicioso en tu aplicación simplemente modificando esa dependencia externa. Si tu proceso de construcción implica un mecanismo automatizado de integración continua/despliegue continuo (CI/CD), es vital asegurarse de que el proceso de construcción está protegido de la interferencia de empleados deshonestos. De lo contrario, podrían modificar el proceso de construcción para inyectar código malicioso sin que este llegue a tocar el repositorio. Dado que muchos programas incluyen ahora la función de actualización automática, si no existe un proceso de verificación, los atacantes podrían crear una "actualización" maliciosa que insertara código malicioso en tus aplicaciones.

Las principales formas de evitarlo son:

- Utilizar firmas digitales o mecanismos similares para verificar las actualizaciones.
- Eliminar los derechos de modificación de las configuraciones del proceso de construcción en el software de CI/CD para todos, excepto para aquellos que realmente lo necesiten (jefes de proyecto, por ejemplo).
- Utilizar, si se desea, herramientas como OWASP Dependency Check u OWASP CycloneDX para buscar vulnerabilidades en tus dependencias.
- Asegurarse de que todas las dependencias de la aplicación web estén alojadas por ti mismo o por un tercero en el que confíes.
- Si se deben enviar datos serializados sin firmar o sin cifrar a un cliente, asegurarse de que existe un mecanismo de comprobación de la integridad para analizar los datos entrantes.

9 A09:2021 – Fallas en el Registro y Monitoreo

El registro es una parte importante de la detección de brechas. Sin él, simplemente no tienes ni idea de lo que está pasando en tus sistemas. Sin embargo, como siempre, hay formas correctas e incorrectas de implementar esta función tan importante. Para obtener los mejores resultados de tu sistema de registro, este debe:

- Registrar eventos sensibles y auditables, como intentos de inicio de sesión (ya sea con éxito o fallidos), pagos con tarjeta de crédito (Nota: los datos de la tarjeta en sí nunca deben ser registrados) u otras acciones importantes.
- Filtrar las entradas del usuario, si procede.
- Generar mensajes de registro claros e inmediatamente comprensibles para advertencias y errores.
- Ser monitoreado de forma rutinaria para detectar actividades sospechosas.
- Disponer de umbrales de alerta adecuados. Configurarlo para que solo registre los ataques confirmados puede omitir datos necesarios para ver otros ataques que no se pueden confirmar tan fácilmente en el servidor. Sin embargo, hacer que registre cada pequeño detalle saturará a tu equipo de monitorización y obstaculizará sus esfuerzos.
- Disponer de un sistema de escalamiento adecuado.
- Generar alertas por la actividad de pruebas de penetración, como las acciones realizadas por OWASP ZAP.
- Escalar, detectar y alertar sobre ataques activos en tiempo real.

Los datos de registro tampoco deben estar a disposición del usuario, y si lo están puede que se esté filtrando información sensible. Debe formularse un plan de respuesta a incidentes y de recuperación para ayudar a resolver los problemas en caso de que surjan.

10 A10:2021 – Falsificación de Solicitudes del Lado del Servidor (SSRF)

La Falsificación de Solicitudes del Lado del Servidor (SSRF) es un tipo específico de ataque que puede permitir a un atacante ver e interactuar con sistemas que se supone que tiene restringidos, normalmente porque están detrás de un firewall o de un puerto con el que normalmente no se espera que el usuario pueda contactar.

El funcionamiento normal de un servicio vulnerable es el siguiente:

1. El cliente envía una solicitud al servidor. Los datos de la solicitud incluyen el destino específico que debe ser contactado por el servidor.

2. El servidor envía entonces una solicitud al destino indicado, posiblemente incluyendo datos de la solicitud original del paso 1.

3. El destino envía datos de vuelta, y el servidor envía estos datos, modificados o no, de vuelta al cliente.

El problema comienza cuando el cliente cambia la variable de destino de la API de algo así como `/api/status` a `http://127.0.0.1:3000`. Entonces, el servidor intenta ponerse en contacto con esta dirección, esencialmente el puerto 3000 en el mismo sistema. Esto se puede utilizar para escanear los puertos de los sistemas (el atacante puede tratar de escanear también otras máquinas en la red de esta manera), y si se encuentra un servicio HTTP válido en el sistema, se puede identificar lo que se está ejecutando en esos puertos. Esto permite a un atacante mapear con precisión tu red interna incluso detrás de un firewall o VPN.

Existen diversas formas de solucionarlo:

- **Desde la capa de red:**

  - Asegúrate de que las comunicaciones entre la aplicación web y los sistemas de la red interna pasan a través de un firewall interno. Establece políticas de denegación por defecto para evitar cualquier comunicación no deseada hacia y desde los sistemas internos.

  - En ese mismo firewall, permite solo las comunicaciones hacia y desde los objetivos deseados.

  - Utiliza la segmentación de red para impedir la comunicación con sistemas no deseados.

  - Establece un ciclo de vida, propiedad y revisión periódica para las reglas del firewall.

  - Registra los flujos de red aceptados y bloqueados de los firewalls.

- **Desde la capa de aplicación:**
  - Crea una lista blanca de todos los destinos de comunicación permitidos. Compara con un mapa u objeto Dictionary conocidos y utiliza los destinos de ese objeto.
  - Filtra todas las entradas de usuario.
  - No envíes respuestas sin procesar a los clientes, ni siquiera para que JavaScript las analice.
  - Deshabilita las redirecciones HTTP.
  - Ten en cuenta la coherencia de la URL para evitar ataques como el enlace DNS y las condiciones de carrera "tiempo de verificación, tiempo de uso" (TOCTOU).
  - NO utilices como solución una lista de denegación (en el que la aplicación tiene una lista de destinos que denegar específicamente). Los atacantes tienen múltiples maneras de eludirlas.