Auditorías Sistema Web Lolipop

Sistemas de Gestión de Seguridad de Sistemas de Información

Asier Astorquiza e Iñigo Ozalla

19 de diciembre de 2021

Índice

1. Introducción

Cuando inicialmente diseñamos nuestro sistema web no nos enfocamos en la seguridad si no en que funcionara bien y fuera fácil de usar. Sin embargo, aplicamos algunas medidas de seguridad básicas que nos han hecho el trabajo mas fácil para esta entrega.

Tras ver la importancia de encriptar las contraseñas decidimos usar hashes para almacenar las contraseñas y para comprobarlas. También leímos documentación sobre como hacer los accesos a la base de datos para crear, actualizar y borrar datos. Nos dimos cuenta que mucha gente destacaba la importancia de hacer este proceso seguro contra inyecciones SQL por lo que en vez de usar los datos que metiera el usuario directamente en las instrucciones SQL los sanitizamos y comprobamos primero.

2. Vulnerabilidades

2.1. Inyecciones SQL

Como hemos mencionado en la introducción en la entrega anterior ya sanitizamos los input para evitar este tipo de ataques. Para crear las instrucciones usamos la extensión MySQLI [Del20], esta extensión facilita la santificación de input y así evita inyecciones SQL. Antes de enviar la instrucción comprueba si va a crear un error de sintaxis o si va a hacer una inyección, si es así no se envía esta instrucción. Esto lo hemos usado en todas las instrucciones que requerían parámetros que insertaba o poda modificar un usuario. No hemos encontrado ningún modo de generar una inyección SQL. Aunque si que teniamos un problema en la base de datos, si navegabas multiples veces a la web `http://localhost:81/follow.php?fmail=iozono11@gmail.com&follow=True` podías hacer seguir múltiples veces al mismo usuario (en este caso iozono11@gmail.com), es decir en la base de datos aparecía que seguías al usuario otra vez cada vez que navegabas a esa pagina. Esto ocurría2 por que no habíamos puesto como clave primaria las dos columnas de la base de datos ahora cuando intentas hacer eso se genera un error.

2.2. Pérdida de autenticación

Para este apartado hemos creado una nueva página *cerrarSesion.php*. Cuando se hace clic en el botón de cerrar sesión se te redirige a dicha página, se borran las variables de sesión, se destruye la sesión y se redirige al login. También se cierra la sesión si se detecta que llevas inactivo durante 60 segundos o mas. Para hacer eso se guarda el tiempo en el que accedes a una página y en el momento en el que accedes a otra página se comprueba si entre el tiempo guardado y el tiempo actual hay mas de 60 segundos, en ese caso se redirige a *cerrarSesion.php* y se cierra la sesión.

2.3. Exposición de datos sensibles

Nosotros decidimos guardar las contraseñas hasheadas y con sal única para cada usuario desde el primer laboratorio por que pensamos que era una parte importante de la página web. Para encriptar la contraseña de registro usamos la función *password_hash* y para comprobar que la contraseña insertada en el login es correcta usamos *password_verify*. La primera función encripta la contraseña de registro con una sal y con el algoritmo *bcrypt*. Se almacena la contraseña hasheada y la sal. Para el login se encripta la contraseña insertada por el usuario con el sal del mail correspondiente (asignado cuando se encripta la contraseña al registrarse) y se comprueba si coinciden la contraseña encriptada de la base de datos y la contraseña encriptada que ha metido el usuario.

Para guardar la información de la tarjeta hemos encriptado todos los números con la misma clave, sin embargo, hemos usado un vector de inicialización distinto y pseudoaleatorio para cada usuario. Este vector de inicialización no es privado y se guarda junto a la tarjeta encriptada para poder luego realizar la descripción. A la hora de desencriptar usamos la misma clave que para encriptar y recuperamos el vector de inicialización de la base de datos. Para hacer esto hemos usado la librería *Open SSL*. [PHP15]

2.4. Logueo y monitorización insuficientes

Una de las vulnerabilidades de nuestro proyecto era que no monitorizábamos ningún tipo de intento de acceso respecto a los login-s, por lo que en caso de haber un incidente, no podríamos comprobar quien ha tenido acceso o quien lo ha intentado, ni tampoco su fecha y hora. Hemos modificado el archivo *login.php* , que es donde se realizaban las conexiones a la base de datos y donde se comprueban si los datos son correctos, y hemos corregido esa vulnerabilidad. A partir de ahora, cada vez que alguien intenta acceder a su cuenta, ya sea con éxito o no, quedará registrado en el archivo *logs.txt* su email, la fecha y la hora del intento.[PHP20] Para que esto sea posible, *logs.txt* debe tener permisos de escritura, por lo que tendremos que dárselo antes. De no ser así, saltará un error de *"Permission denied"* y no se registrará el intento.

2.5. Secuencia de comandos en sitios cruzados

Hemos encontrado múltiples Inyecciones SQL debido a que no comprobábamos los inputs de los usuarios. Todas las ocurrencias de SQL han sido detectadas manualmente.

Ejemplo de ataque SQL en la entrega inicial:

1. Iniciar sesión con una cuenta existente
2. En perfil podemos hacer una inyección SQL persistente. Para en gustos escribiremos (Ref1):

```
</textarea><script>alert("Stored XSS")</script>
```

Lo que ocurre aquí es lo siguiente. Primero tenemos que cerrar el elemento *textarea* y una vez hecho eso podemos comenzar la inyección XSS. En nuestro caso solo mostramos una alerta por pantalla pero los atacantes podrían hacer cosas mucho peores como redirigir a una web maliciosa, a una web phising, o robar cookies si están mal configuradas. Debido a que los gustos se guardan en la base de datos se trata de una inyección almacenada y cualquiera que vea nuestro perfil ejecutará el script. Se nos ocurrió que podíamos usar esta vulnerabilidad para hacer que todo el mundo nos siguiera, para ello usamos el siguiente código en gustos:

```
d</textarea>
<script>
window.location.href = "follow.php3Ffmail%3Diozono11%40gmail.com%26follow%3DTrue"
</script>
<textarea>h
```

Cuando alguien vea nuestro perfil se va a ejecutar el script que nos llevará a la web `follow.php?fmail=iozono11@gmail.com&follow=True`. Esto hará que el usuario que este logueado empiece a seguir a el usuario que esta en la URL (en este caso iozono11@gmail.com). Se puede comprobar que el ataque ha funcionado viendo la tabla seguidores de la base de datos. Por la naturaleza del ataque cada vez que veamos los gustos de una persona con el script inyectado se ejecutará el código, por lo

que cada vez que accedamos a nuestro perfil se redirigirá a la página mencionada anteriormente y no podremos editar nuestro perfil.

Para solucionar las vulnerabilidades de los ataques de tipo XSS hemos decidido *sanitizar* los parámetros input del usuario usando la función *htmlspecialchars*[Laz20]. Anteriormente, como la contraseña la hasheabamos, no hacia falta ocuparnos de esa codificación. Sin embargo, los demás parámetros estaban desprotegidos en las páginas registro, perfil y la de añadir elementos. La función *htmlspecialchars* se encarga de escapar los caracteres especiales como pueden ser "corchetes HTML, &, $..."para que el usuario no pueda interactuar con el HTML a través de los datos que inserte. Decidimos que era una mala idea comprobar los datos en el JavaScript por que nuestro PHP coge los datos de la dirección con *$_GET["PARAMETRO"]*. Si alguien decide hacer un ataque de *HTTP Parameter Pollution* podría inyectar el HTML saltándose la comprobación de JavaScript. Un ejemplo de este ataque sería navegar a la URL:

`http://localhost:81/profileEdition.php?fsexualidad=hetero&fnombre=test0&fapellidos=assa&fdni=16102466-M&ftelefono=111111111&ffechanac=11-11-1111&fgustos=</textarea><script>alert("StoredXSS")</script>&fpeso=111&faltura=222&ftarjeta=1234567890987654321`

Como podemos ver, hemos inyectado el script en el campo de gustos y estamos navegando a la pagina que se encarga de actualizar los datos en la base de datos por lo que no se comprueban los datos en JavaScript. [Alo11] Las vulnerabilidades XSS encontradas afectaban a las paginas de editar perfil, añadir elemento y editar elemento. Para reproducir las inyecciones solo hay que meter el codigo mencionado antes (Ref1) en el campo gustos.

2.6. Configuración de seguridad incorrecta

En la primera entrega no tuvimos en cuenta el nivel de seguridad de las contraseñas de los usuarios registrados, algo bastante descuidado, ya que pueden ser hackeados con mas facilidad. Para corregir esta vulnerabilidad hemos modificado el archivo donde se procesan los datos introducidos para el registro (*procesarRegistro.js*). Usamos expresiones regulares para comprobar que en la contraseña introducida haya uno o varios caracteres de: minúsculas, mayúsculas, dígitos, caracteres especiales y por ultimo comprueba que haya al menos 8 caracteres. Si no cumple las condiciones se toma como contraseña inválida y no se permite el registro.

2.7. Rotura de control de acceso

El sistema no debe permitir que un usuario modifique los datos de otro usuario aunque conozca su identificador, su DNI o cualquiera de sus datos. En la primera entrega ya controlamos esta vulnerabilidad. Para ello utilizamos *variables de sesión*. En el archivo *login.php* recogemos en dos variables de sesión el mail del usuario y un booleano que nos permite saber si se ha logueado o no. De esta manera, si queremos acceder a un listado o a una página que solo puedes tener acceso si estas logueado, se comprueban las variables de sesión. En caso de que no haya iniciado sesión se te negará el acceso. Ocurrirá lo mismo si intentamos acceder a una página a la que no tenemos permiso. Por ejemplo, editar el perfil de otra persona. Además, también usamos el mail de las variables de sesión si queremos realizar consulta o modificación en SQL.

Respecto al cierre automático por inactividad ya se ha explicado en el apartado de Pérdida de autenticación.

3. Conclusión

Este ultimo proyecto nos ha enseñado la importancia de fortificar un sistema web ante ataques externos. Nos ha parecido un trabajo entretenido y la información consultada ha sido interesante. En la primera auditoría hemos encontrado múltiples errores ya mencionados anteriormente y hemos logrado solucionarlos todos. En la segunda auditoría nos hemos dado cuenta que podíamos saltarnos las comprobaciones de JavaScript si accedíamos directamente a la pagina PHP que se iba a encargar de realizar las instrucciones SQL. No hemos solucionado esta vulnerabilidad ya que no esta en los objetivos de el trabajo. De todas formas esto nos ha influenciado a la hora de implementar la protección contra XSS. Hemos decidido añadir esta protección en la pagina PHP encargada de realizar las instrucciones

SQL pertinentes. Creemos que esa vulnerabilidad es de tipo *HTTP Parameter Pollution* ya que consiste en editar los parámetros de la URL para saltarse la comprobación JavaScript. Podría solucionarse no pasando los datos como parámetros de la URL por que cualquiera puede editar estos datos.

Referencias

[Alo11] Chema Alonso. Http parameter pollution. https://www.elladodelmal.com/2011/03/hpp-http-parameter-pollution.html, 2011. Explicacion de HTTP Parameter Pollution.

[Del20] PHP Delusions. Mysqli examples. https://phpdelusions.net/mysqli_examples/, 2020. Uso de MySQLI.

[Laz20] Diego Lazaro. Ataques xss: Cross-site scripting en php. https://diego.com.es/ataques-xss-cross-site-scripting-en-php, 2020. Prevencion de ataques XSS.

[PHP15] PHP. Openssl. https://www.php.net/manual/es/book.openssl.php, 2015. Informacion para encriptar la tarjeta.

[PHP20] PHP. File put contents. https://www.php.net/manual/es/function.file-put-contents.php, 2020. Uso de la función PHP file put contents.