INFORME DE AUDITORÍA DE SEGURIDAD WEB: WEBGOAT

Fecha: 15/01/2025

Auditor: Renatto Minaya Rojas

Metodología: OWASP Top 10:2025

Objetivo: Plataforma de entrenamiento WebGoat

---

1. Ámbito y Alcance de la Auditoría

El objetivo de esta auditoría es evaluar la seguridad de la aplicación web “webGoat” enfocándose en aspectos técnicos basados en el análisis de posibles formas de inyección de código malicioso, así como posibles puertos abiertos y vulnerabilidades conocidas por el software utilizado para la construcción de la aplicación web mencionada. Las pruebas serán realizadas a la URL http://localhost:8080/WebGoat dentro de un entorno de container docker que a su vez se encuentra siendo ejecutado dentro de una máquina virtual. La auditoría presentada es de tipo caja gris, ya que, por la naturaleza de la plataforma webGoat, podemos obtener credenciales para acceder a ella fácilmente.

El alcance máximo de este informe es el de una auditoría web por lo que se excluyen ciertos tipos de pruebas tales como:

- Denegación de servicio (DDoS)
- Intentos de phishing
- Infestar el servidor con malware
- Instalar backdoors persistentes
- Tipos de ataques de red como MITM y otros
- Diferentes formas de ataques de ingeniería social

Por otro lado, el alcance del informe si abarca escanear y explotar vulnerabilidades inherentes a la construcción de la página web: webGoat. Se usará como referencia la metodología OWASP con su top 10 de vulnerabilidades web del 2025.

2. Informe Ejecutivo

a. Breve resumen del proceso realizado

Se ha llevado a cabo una auditoría de "Caja Gris" siguiendo el estándar de **OWASP**. Cada vulnerabilidad fue desarrollada siguiendo los siguientes pasos:

1. Reconocimiento de la vulnerabilidad: Se identifica la vulnerabilidad usando herramientas y también de forma manual.
2. Explotación: una vez identificada se procede a ejecutar todas las posibles formas de explotar la vulnerabilidad.
3. Post-explotación: que acciones podría realizar un atacante si logra descubrir y explotar la vulnerabilidad.
4. Posibles mitigaciones: recomendaciones de medidas de seguridad para evitar exponer la vulnerabilidad.
5. Resultado: utilizando la calculadora CVSS 3.1 se obtiene el vector y la puntuación de cada vulnerabilidad.

Cada paso ha sido documentado usando capturas de pantalla y descripciones breves de lo que se está ejecutando. Se especifica también el porqué es importante mitigar dichos fallos y el impacto que tendrían de ser descubiertos, esto considerando la tríada CIA (confidencialidad, integridad y disponibilidad).

b. Vulnerabilidades destacadas

ID OWASP 2025	Vulnerabilidad	Severidad (CVSS)	Riesgo
A05:2025	SQL injection	7.8	Alto
A05:2025	XSS reflected	6.1	Medio
A05:2025	XSS stored	7.2	Alto
A02:2025	Security misconfiguration - CSRF	6.5	Medio

A03:2025	Software supply chain failures - Outdated components XSS	6.1	Medio
A07:2025	Authentication Failures - Insecure passwords	9.0	Crítico

c. Conclusiones

La plataforma WebGoat presenta múltiples deficiencias que permiten a un atacante:

- Robo de identidad
- Manipular datos de la plataforma
- Filtrar datos confidenciales
- Elevar y revocar permisos de acceso
- Insertar código malicioso visible en el frontend

Entre las deficiencias más críticas se encuentran:

- A07:2025 Authentication Failures - Insecure passwords
- A05:2025 XSS stored
- A05:2025 SQL injection

Se debe priorizar el desarrollo de parches de seguridad, aplicación de mejores prácticas en la configuración y desarrollo de software, implementación de logs y monitoreos para mitigar la presencia de las deficiencias más críticas mencionadas. Una vez cubiertas estas vulnerabilidades proceder a ejecutar las medidas de seguridad correspondientes para las deficiencias de riesgo medio.

d. Recomendaciones

Para fortalecer la seguridad de la plataforma webGoat, se recomienda implementar las siguientes medidas:

1. Estandarizar el uso de prepared statements o ORM.
2. Utilizar validadores de datos en el frontend para evitar ataques XSS.
3. Gestionar las dependencias utilizadas eliminando dependencias innecesarias y actualizarlas a las últimas versiones.
4. Implementar tokens de sesión únicos desde el servidor y exponer las cookies de sesión sólo en el mismo origen.
5. Implementar un segundo factor de autenticación para el login de usuarios.
6. Monitoreo y logging en tiempo real de peticiones y accesos a webGoat.
7. Re-auditar periódicamente la plataforma en busca de más vulnerabilidades.

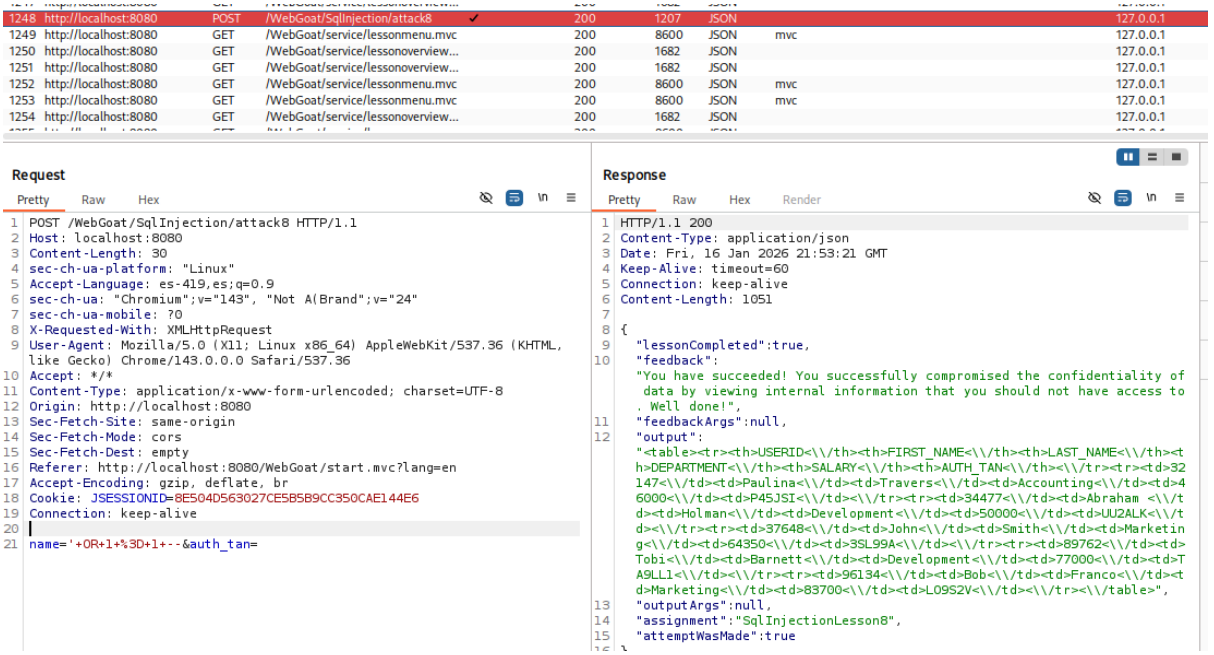
3. Descripción del proceso de auditoría

3.1. SQL injection (A05:2025 Injection)

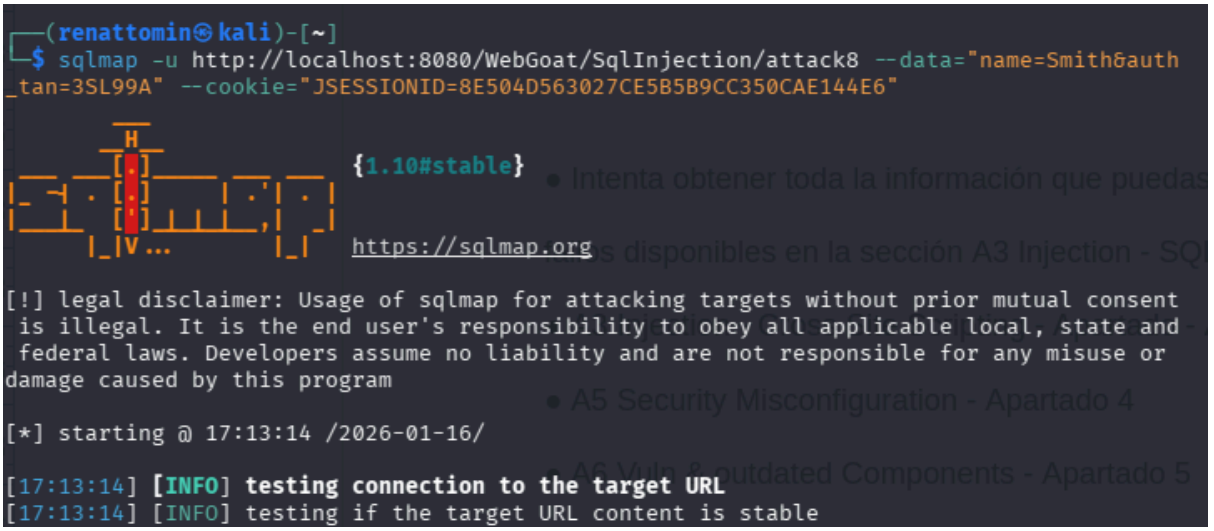
Según el apartado 11 de sql injection(Intro) de webGoat, una aplicación web es vulnerable a un ataque de sql injection si el código solo concatena, al final de cada consulta SQL, el texto insertado por el usuario.

a. Reconocimiento de la vulnerabilidad

Utilizaremos Burp Suite como proxy para interceptar el tráfico de webGoat. De esta manera obtenemos la información que se envía en el request de manera detallada.



Gracias a burp podemos obtener el formato del request y con esta información podemos usar sqlMap como se adjunta en el siguiente screen. Utilizaremos la herramienta sqlMap para automatizar diferentes ataques de sql injection y observar qué tanta información está expuesta.



El comando utilizado es el siguiente:

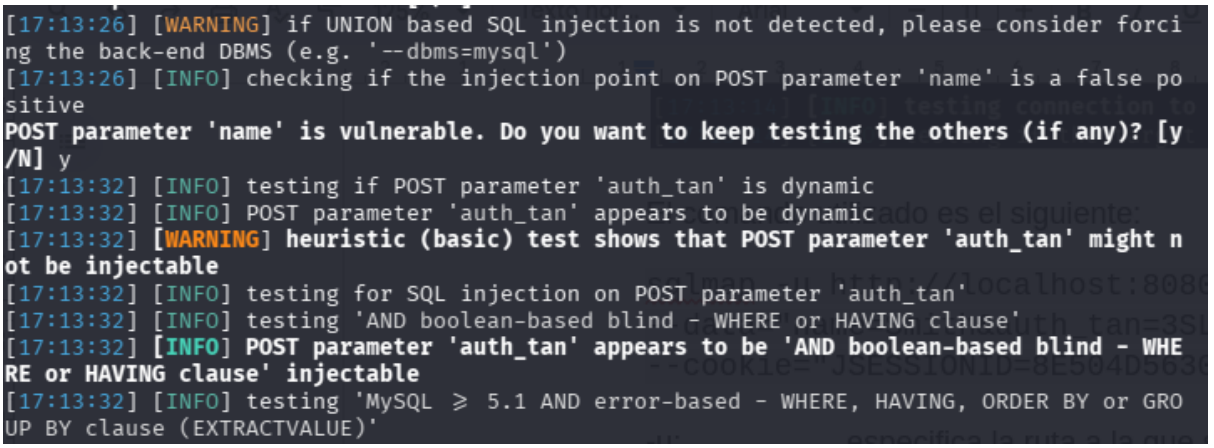
```
sqlmap -u http://localhost:8080/WebGoat/SqlInjection/attack8
--data="name=Smith&auth_tan=3SL99A"
--cookie="JSESSIONID=8E504D563027CE5B5B9CC350CAE144E6"
```

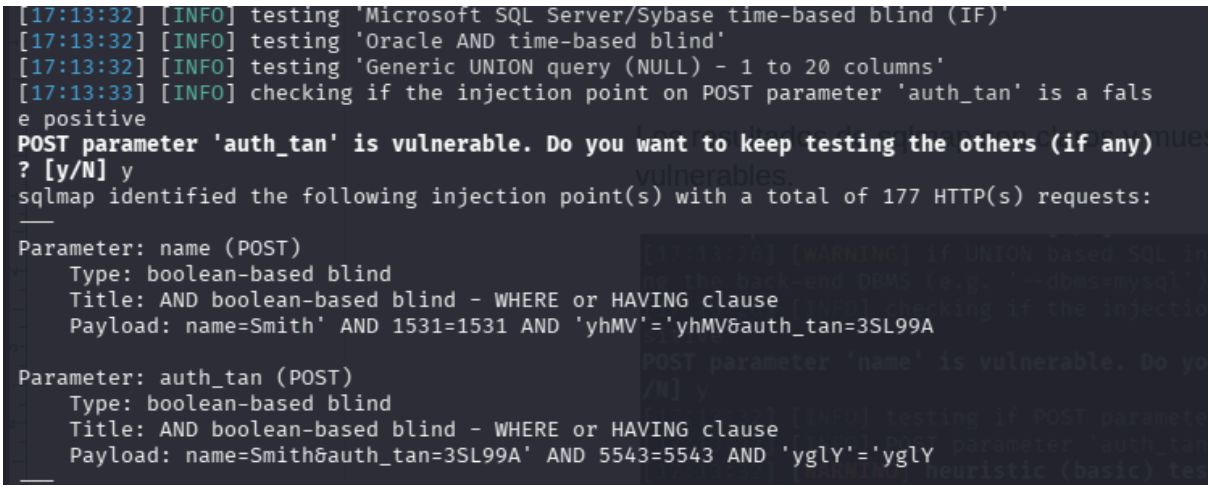
-u: especifica la ruta a la que sqlmap intentara inyectar sql

--data: el body enviado en la petición POST

--cookie: la cookie de sesión del usuario obtenida gracias a burp

Los resultados de sqlmap son claros y muestran que ambos parámetros son vulnerables.





b. Explotación de la vulnerabilidad

Ataques que perjudican la confidencialidad

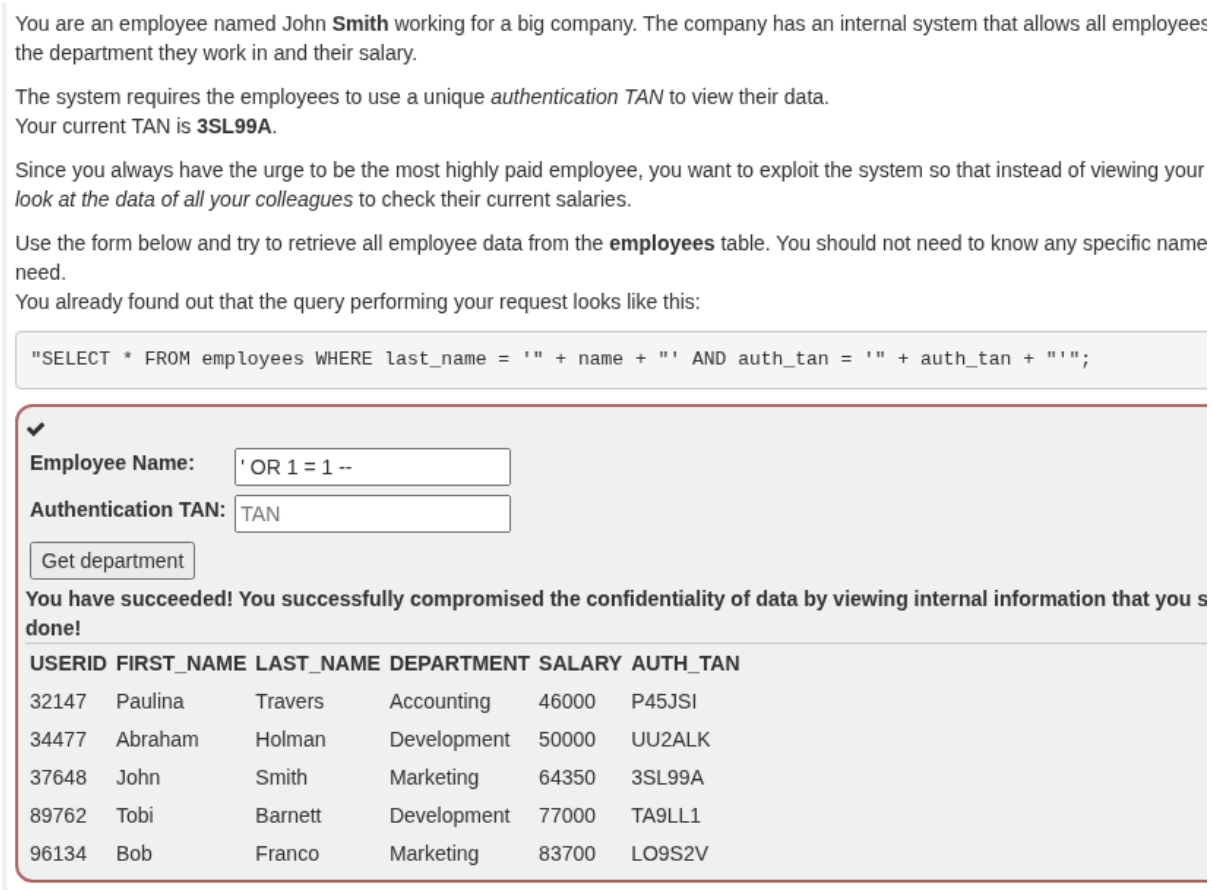
WebGoat nos explica que cada usuario de una empresa ficticia puede acceder a su información personal para verificar su salario y el departamento donde trabaja. Nos ofrece 2 campos de texto: 1 para ingresar el apellido del empleado y otro para ingresar el TAN (identificador) del empleado.

La consulta SQL está formada de la siguiente manera:

```
"SELECT * FROM employees WHERE last_name = '" + name + "' AND auth_tan = '" + auth_tan + "'";
```

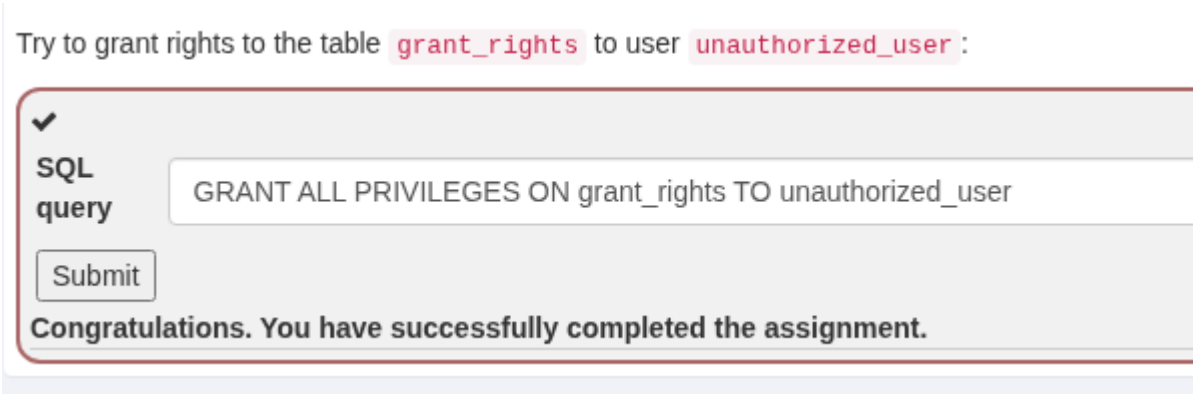
Esta información es ofrecida por la misma plataforma de webGoat. Como se puede apreciar en la consulta, se puede concatenar código SQL y se ejecutará sin problemas. Asi que si agregamos last_name lo siguiente: **' OR 1 = 1 --**

Es posible que accedamos a la información de todos los empleados de la empresa. Se adjunta screen de la prueba y el resultado.



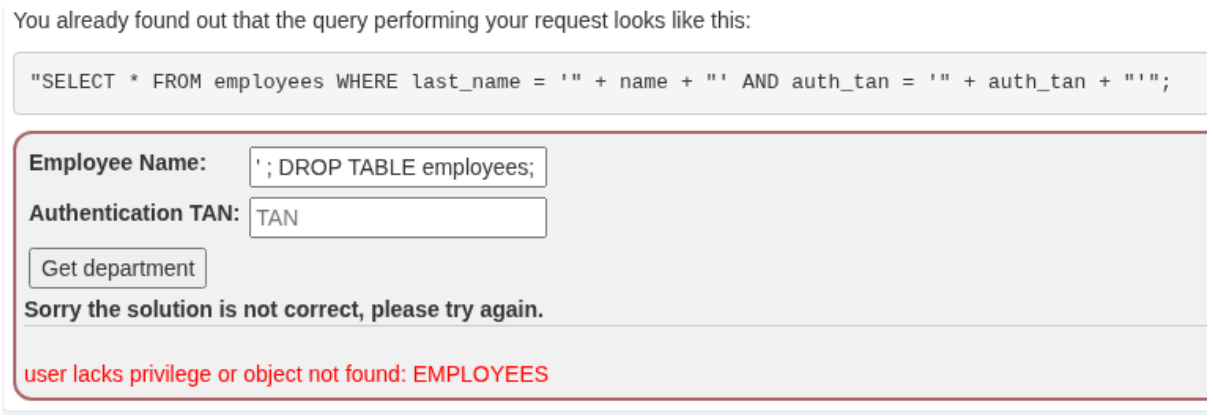
Al agregar el or 1 = 1 hacemos que siempre se cumpla True el query y si agregamos adicionalmente el -- que es símbolo de comentario en SQL, hacemos que el resto del query donde se inserta el TAN no surta efecto y como resultado tenemos la lista de empleados con sus salarios y el departamento donde trabajan. Información muy sensible expuesta. Esto afecta directamente la **confidencialidad** de la información.

Otra forma en la que la confidencialidad de los datos se ve expuesta es con consultas DCL, este tipo de consultas puede ejecutar un GRANT y de esa forma hacer que un usuario desconocido tenga privilegios de administrador de la base de datos. De esta manera toda la información de la base de datos queda a merced total de este usuario.



En webGoat podemos ejecutar un GRANT para un unauthorized_user perjudicando la **confidencialidad** de los datos.

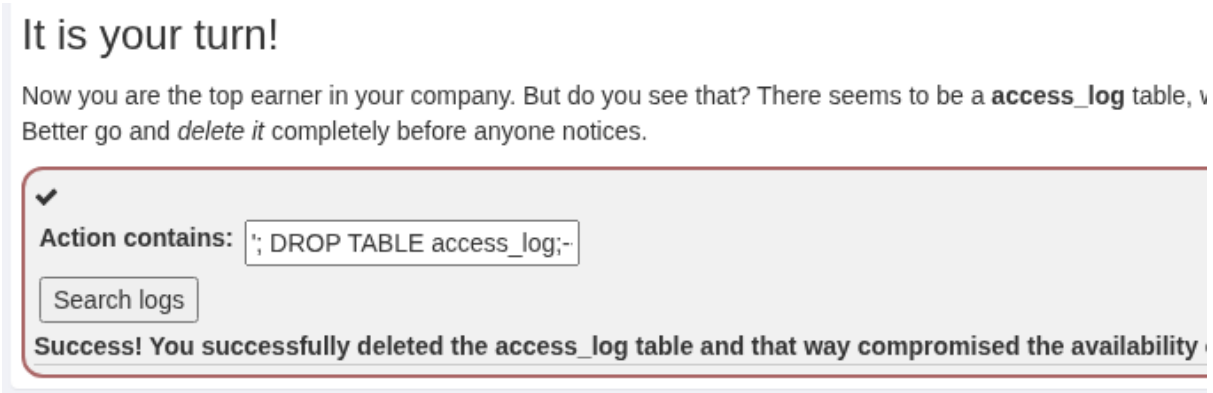
Ataques que perjudican la disponibilidad



Al agregar: `' ; DROP TABLE employees; –`

Se borró por completo la tabla employees, por lo que el usuario actual tiene los permisos de hacer un drop de tablas afectando la **disponibilidad**.

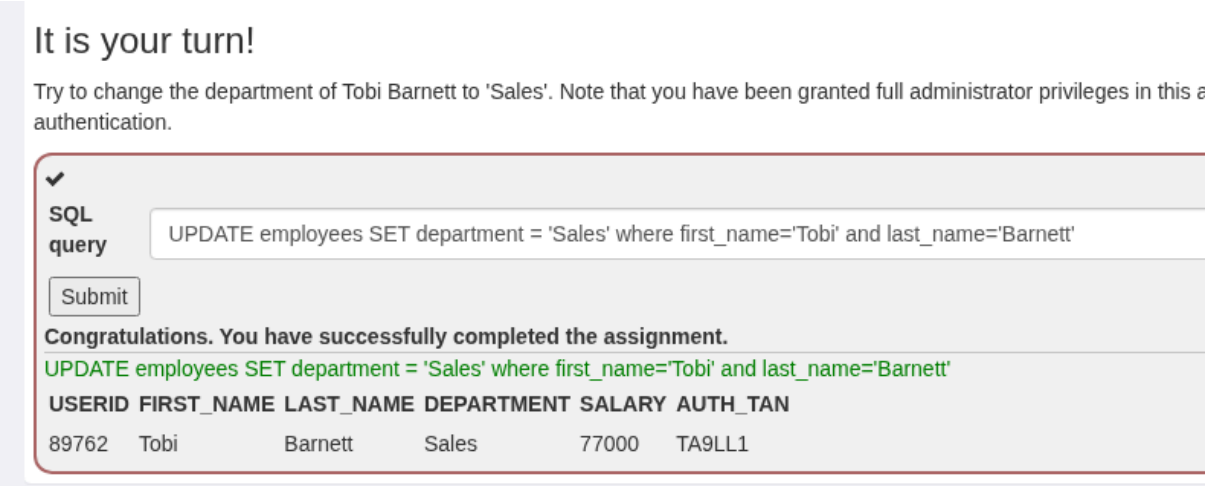
Otra forma de afectar la disponibilidad es eliminar el access_log de la base de datos. El access_log guarda el registro detallado de todas las operaciones y accesos que se ejecutan en la base de datos, de manera que si se elimina, el atacante borraría su rastro de la base de datos.



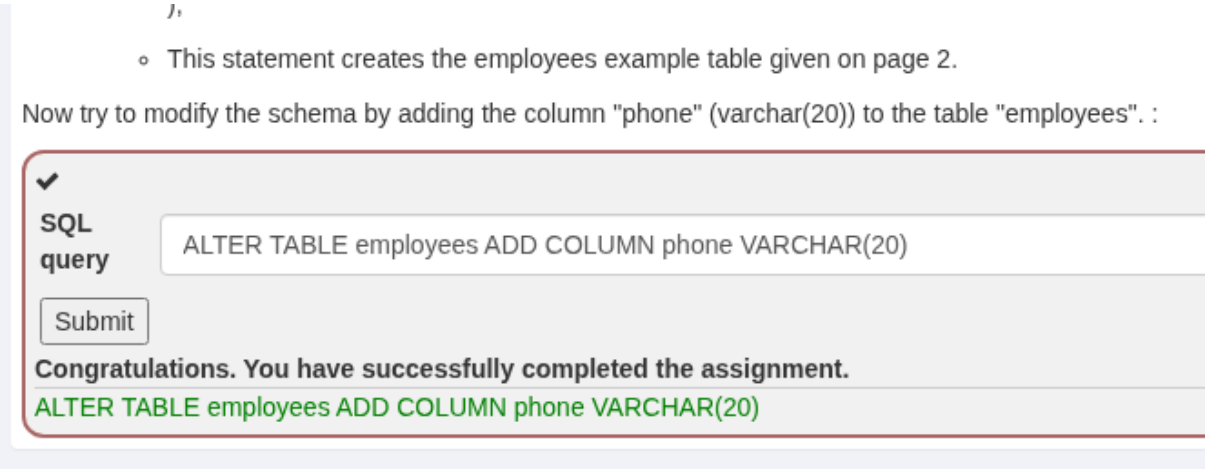
Se ejecuta un `DROP TABLE access_log` afectando la **disponibilidad** de la tabla.

Ataques que perjudican la integridad

Además el usuario al tener privilegios de administrador puede también afectar la integridad de los datos al hacer un UPDATE como vemos en el apartado 3 de la sección de sql injection de webGoat.

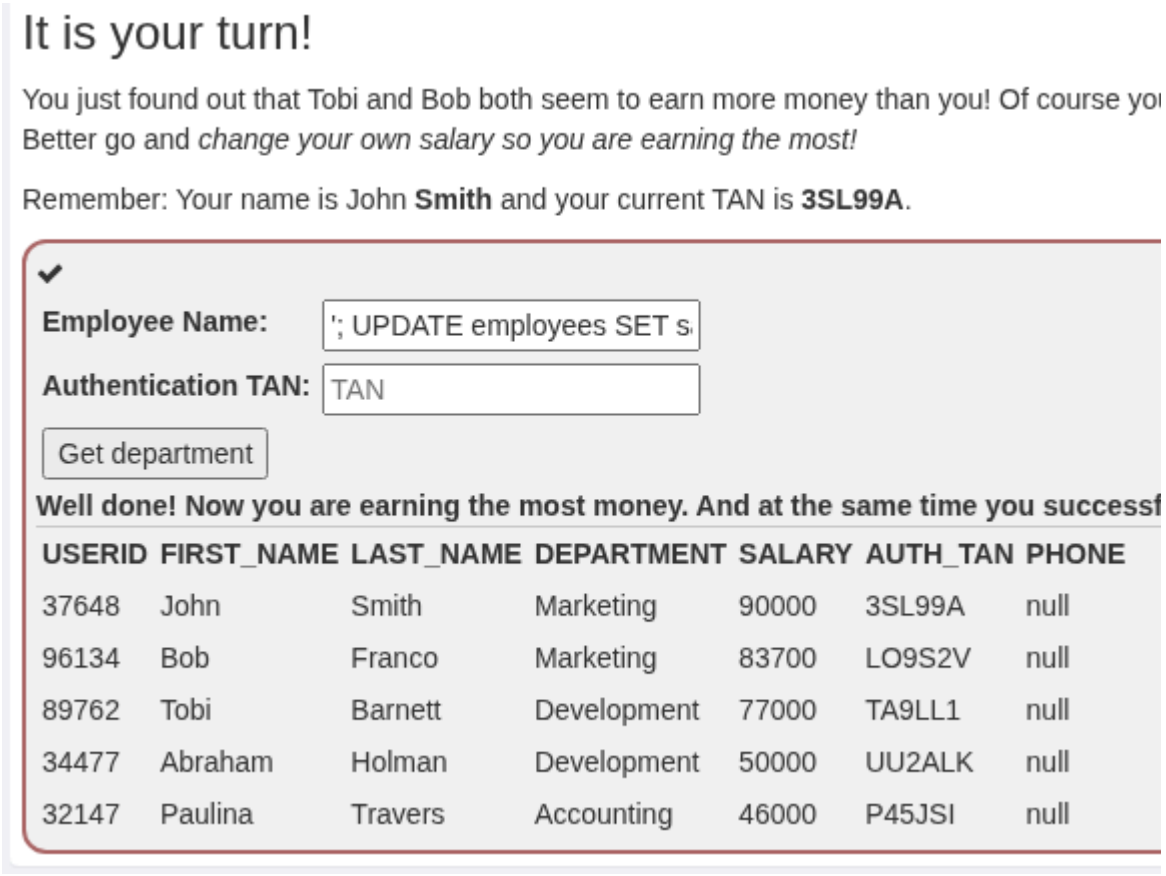


Podría cambiar el departamento e incluso el salario de los empleados, lo cual es muy perjudicial para la empresa.



En este caso se está ejecutando una consulta DDL afectando la **integridad**, ya que es un ALTER que modifica la estructura de las tablas, pero bien puede ser un DROP como se ejecutó anteriormente y perjudicar la **disponibilidad**.

Otra manera de afectar la integridad de los datos es si un empleado busca aumentarse el salario a sí mismo o disminuir el salario a sus colegas. Usando un UPDATE podría hacer eso y como el SQL se concatena podría usar ; y -- para poder ejecutar un UPDATE por completo como si lo estuviera haciendo directamente en la consola del motor de la base de datos.



El SQLi utilizado fue este:

`'; UPDATE employees SET salary = 90000 WHERE auth_tan = '3SL99A';--`

Se utilizó ‘; para culminar la primera sentencia SQL y luego del UPDATE se utilizó -- para que no se ejecute la última sentencia SQL.

De esta manera se ve comprometida la **integridad** de los datos.

c. Post-explotación

Una vez confirmado el acceso a la base de datos se procedió a realizar las siguientes acciones:

1. Filtración de datos sensibles(confidencialidad): usando SQLi se pudo visualizar información de los empleados de la empresa ficticia, esto incluía el salario de los empleados y los departamentos a los que pertenecían.
2. Eliminación de datos sensibles(disponibilidad): fue posible eliminar tablas y entre ellas el access_log que permitió borrar todo rastro de los SQLi ejecutados.
3. Escalada de privilegios(confidencialidad): utilizando DCL se puede dar privilegios de administrador a un usuario desconocido y además se pueden revocar privilegios a los usuarios que ya son administradores.

4. Manipulación de datos sensibles(integridad): usando queries UPDATE se pueden modificar los datos de los empleados, como se puede apreciar en la anterior fase, es posible aumentar el salario de un empleado.

d. Posibles mitigaciones

1. Reducir permisos y privilegios del usuario: el principal factor para evitar que manipulen la base de datos, es asegurarse que los clientes de la plataforma no tengan acceso de administrador y reservarle solo los permisos para las tablas a las que deben acceder.
2. Usar prepared statements o ORMs: esta es la solución primordial. Aplicando estas opciones se evita que el posible código malicioso que envíe un cliente sea ejecutado directamente como una concatenación del query ya establecido en el backend. Esto sucede debido al funcionamiento mismo de los prepared statements y los ORMs, en ambos casos, se toma el texto del usuario como literales en lugar de código.
3. Filtrar palabras clave y símbolos como ; y --: puede ser una capa adicional de seguridad para evitar que intenten ejecutar código aunque aplicando prepared statements o ORMs no sería necesario aplicar este método, pero de aplicarlo, se debería implementar en el backend y no en el frontend.

e. Resultado

En conclusión, según las pruebas realizadas se obtiene una puntuación de **7.8** y un vector:

`CVSS:3.1/AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H`

Siendo una vulnerabilidad de riesgo: **high**

3.2. Cross Site scripting XSS (A05:2025 injection)

a. Reconocimiento o information gathering

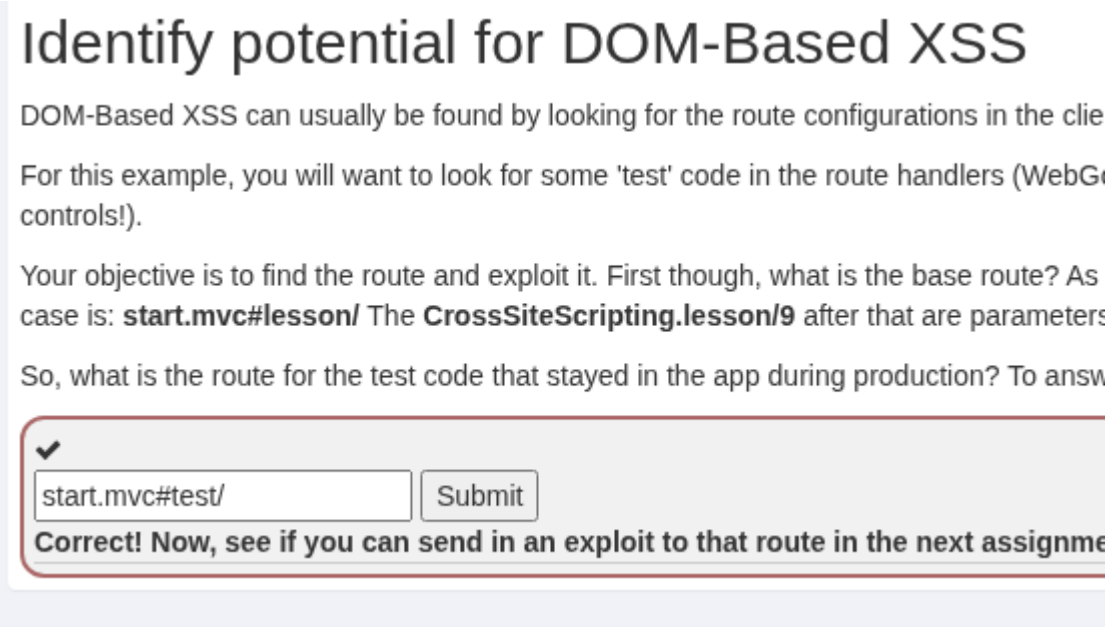
En el apartado 7 de cross site scripting se analizó los campos de texto y se puede insertar scripts de javascript que son ejecutados por el navegador y podría usarse para XSS.

`<script>console.log(document);</script>`

Es lo que se ha utilizado para comprobar que webGoat es vulnerable a XSS.

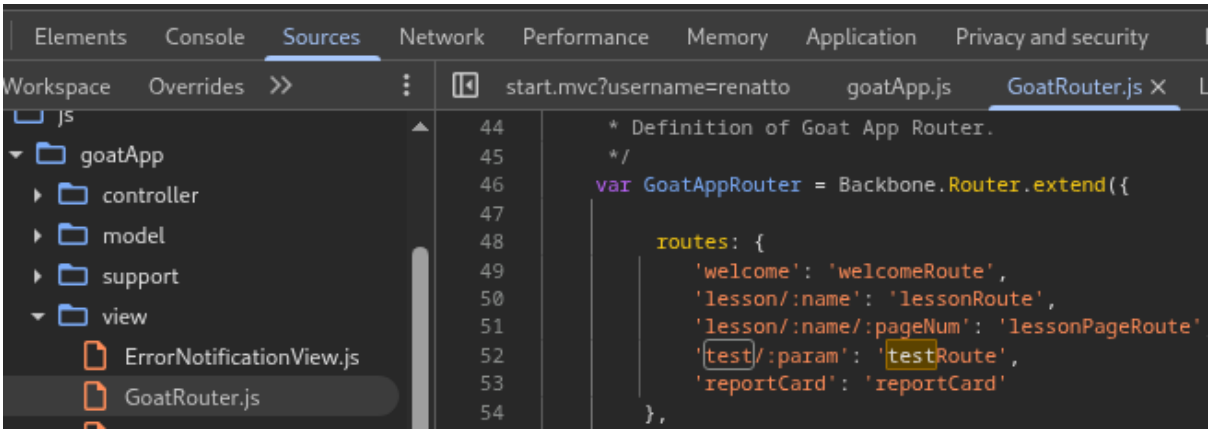
b. Explotación de la vulnerabilidad

Ahora que se conoce que es posible un XSS reflejado, se puede explotar un DOM-based XSS. En el apartado 10 de la sección se especifica que hay una ruta /test que está siendo expuesta en producción y toma los parámetros como código para ser ejecutado. Si algún atacante malintencionado conoce esta ruta podría usarlo para enviar el link modificado a alguna víctima y obtener credenciales importantes o su cookie de sesión, información muy delicada.

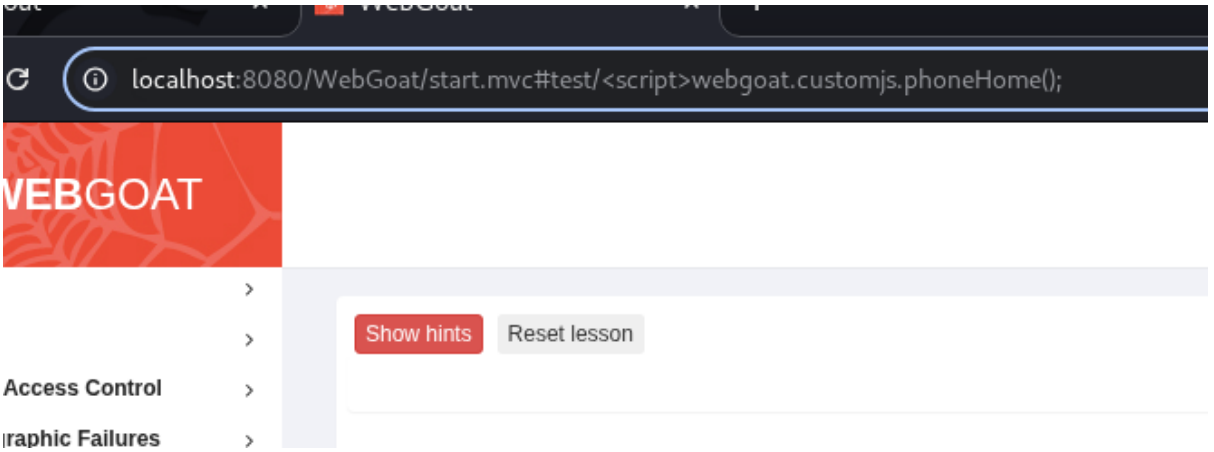


Se obtuvo la ruta test al verificar el source de la aplicación, la cual está disponible públicamente para cualquier persona que acceda a la página.

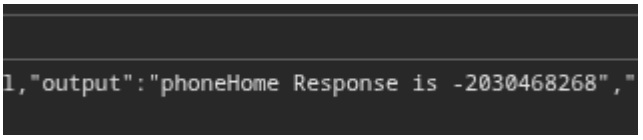
Y por la configuración de la ruta, los parámetros son ejecutados como código. Esto se puede apreciar revisando la dev tools del navegador.



Esto quiere decir que si abro un nuevo tab con la ruta encontrada y un código malicioso como parámetro, este se va ejecutar sin problemas.



En este caso por ser un ejercicio de webGoat obtenemos un número que simula ser un teléfono:



Pero un ataque XSS podría usarse para obtener una cookie de sesión o incluso para enviar información sensible sin que los usuarios de la página lo noten. Esto afecta la **confidencialidad** de los datos y también la **disponibilidad**, porque dependiendo de la información obtenida por el atacante, podría eliminar datos sensibles.

c. Post-explotación

En esta prueba realizada con webGoat y simulando que es una empresa ficticia hemos podido realizar las siguientes acciones:

1. Exposición de datos sensibles: se puede obtener cookies de sesión, teléfonos y datos personales y más grave aún datos financieros.

2. Peligro potencial de escalar a reverse shell: aunque el XSS solo afecta al cliente, si la información obtenida es de un administrador por ejemplo, un atacante más preparado podria usar esa sesión para insertar algun script de RCE de forma remota, y por los privilegios del usuario como administrador, el servidor tomará este script como válido y de esa forma el atacante tiene acceso total al servidor.

d. Posibles mitigaciones

1. No exponer innecesariamente rutas o campos de texto ocultos: en muchas páginas se exponen campos o rutas que no se usan directamente en producción, y estos pueden ser una puerta de entrada para un ataque XSS.
2. Escapar variables js: para que no se ejecute el javascript enviado, se debe tomar como texto literal y al escapar las variables que manejan esta data, se evita que se ejecute como código javascript.

3. Agregar validaciones en el frontend: hay librerías disponibles según el lenguaje de programación que ayudan a validar la data ingresada. Validando se asegura de que aunque inserten código malicioso, éste no proceda a ejecutarse. Por ejemplo si insertan un email, el validador se asegurará de que sea un email en lugar de un tag <script> con contenido malicioso. Así si alguien llega a cambiar la URL de la página, al ser un XSS reflejado, el código no se ejecutará.
4. Logs de acceso y escritura: esto es para el caso de los XSS stored, ayudarán a detectar si alguien desconocido insertó código que no estaba previsto para la aplicación. Es una ayuda adicional, ya que con los puntos anteriores, ya se debió cubrir que algún campo o la url misma pueda dejar código directamente en el servidor o en la base de datos.

e. Resultado

Para un ataque XSS reflejado la puntuación es de **6.1**
`CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N`

Riesgo: medio

Para un ataque XSS almacenado la puntuación es de **7.2**

`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:L/I:L/A:N`

Riesgo: alto

La principal diferencia radica en que el XSS almacenado ya vive en el servidor y por lo tanto se ejecutara masivamente y automáticamente, lo que lo hace más peligroso que el reflejado.

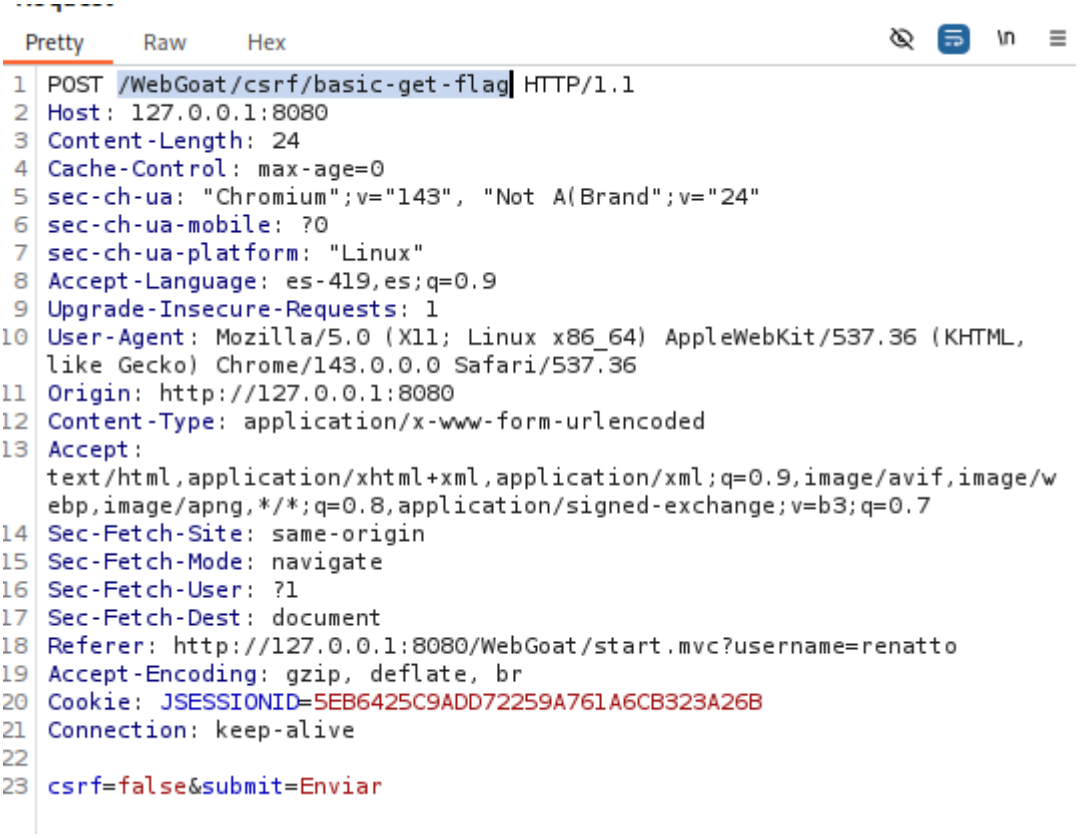
3.3. Security Misconfiguration (A02:2025 Security Misconfiguration)

a. Reconocimiento de la vulnerabilidad

Una vulnerabilidad de security misconfiguration sucede cuando la aplicación web no ha pasado por un proceso de hardening. Es posible que las librerías utilizadas tengan vulnerabilidades, que los logs de errores den información excesiva, que se usan passwords por defecto o que hayan flags desactivados de seguridad, etc. Todo aquello que implique una configuración y que no haya tenido en cuenta las mejores prácticas de seguridad, son una posible vulnerabilidad de security misconfiguration.

En webGoat hay un apartado de ataques CSRF. En este tipo de ataques el hacker se aprovecha de la confianza que una página tiene por los usuarios ya logueados.

Usando burp suite podemos interceptar las peticiones del usuario, donde se visualiza la cookie de sesión y se puede usar para, externamente, hacer consultas como si fueras el mismo usuario.

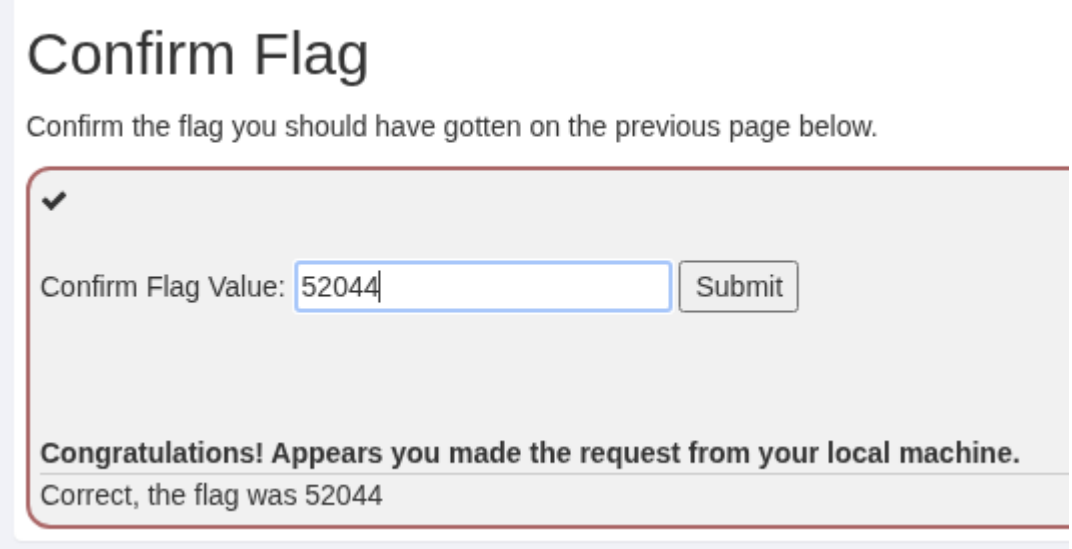


WebGoat nos pide hacer la misma consulta pero desde una fuente externa y debido a que no han configurado un bloqueo adicional para consultar si el usuario que intenta realizar una petición es realmente quien dice ser o que la petición proviene del mismo origen y no de una fuente externa, es posible el ataque CSRF.

Usando curl y los datos obtenidos en burp suite, podemos reproducir el request.

```
┌──(renattomin㉿kali)-[~]
└─$ curl -X POST "http://127.0.0.1:8080/WebGoat/csrf/basic-get-flag" \
    -H "Accept-Encoding: gzip, deflate, br" \
    -H "Cookie: JSESSIONID=5EB6425C9ADD72259A761A6CB323A26B" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    -d "csrf=false&submit=Enviar"
{
  "flag" : 52044,
  "success" : true,
  "message" : "Congratulations! Appears you made the request from a separate
host."
}
```

En el ejemplo de webGoat obtenemos un flag, pero si fuera una aplicación real, podríamos obtener datos sensibles afectando la **confidencialidad**. Y dependiendo de los endpoints y la configuración de seguridad en ellos, el hacker podría modificar datos o incluso eliminar data importante, afectando la **integridad** y **disponibilidad**.



b. Explotación de la vulnerabilidad

En el apartado 4 tenemos una sección de posteo de comentarios. El ataque CSRF se ejecuta de forma similar a lo explicado en el reconocimiento de la vulnerabilidad. Esto expone la facilidad con la que un atacante podría afectar la integridad de los datos en una página web, ya que, se espera que no hayan ese tipo de comentarios desde fuentes externas.

Pude realizar el ataque observando los datos interceptados desde burp suite y ejecutando el request desde curl.

```
POST /WebGoat/csrf/review HTTP/1.1
Host: 127.0.0.1:8080
Content-Length: 80
sec-ch-ua-platform: "Linux"
Accept-Language: es-419,es;q=0.9
sec-ch-ua: "Chromium";v="143", "Not A(Brand";v="24"
sec-ch-ua-mobile: ?0
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/
like Gecko) Chrome/143.0.0.0 Safari/537.36
Accept: */*
Content-Type: application/x-www-form-urlencoded; charset
Origin: http://127.0.0.1:8080
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://127.0.0.1:8080/WebGoat/start.mvc?usernam
Accept-Encoding: gzip, deflate, br
Cookie: JSESSIONID=5EB6425C9ADD72259A761A6CB323A26B
Connection: keep-alive

reviewText=Hola+soy+renatto&stars=1&validateReq=
2aa14227b9a13d0bede0388a7fba9aa9
```

El primer response indica que estoy ejecutando el request desde el mismo sitio.


```
{
  "lessonCompleted":false,
  "feedback":
  "It appears your request is coming from the same host you are submitti
  ng to.",
  "feedbackArgs":null,
  "output":null,
  "outputArgs":null,
  "assignment":"ForgedReviews",
  "attemptWasMade":true
}
```


Pero una vez realizado desde curl se postea el comentario sin problemas.

```
┌──(renattomin㉿kali)-[~]
└─$ curl -X POST "http://127.0.0.1:8080/WebGoat/csrf/review" \
    -H "Accept-Encoding: gzip, deflate, br" \
    -H "Cookie: JSESSIONID=5EB6425C9ADD72259A761A6CB323A26B" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    -d "reviewText=Hola+soy+renatto&stars=1&validateReq=2aa14227b9a13d0bede0
388a7fba9aa9"
{
 "lessonCompleted" : true,
 "feedback" : "It appears you have submitted correctly from another site. Go
reload and see if your post is there.",
 "feedbackArgs" : null,
 "output" : null,
 "outputArgs" : null,
 "assignment" : "ForgedReviews",
 "attemptWasMade" : true
}
```

En el ejemplo de webGoat solo se trata de postear comentarios, pero un ataque CSRF puede incluir operaciones graves como transferencias de dinero.

c. Post-explotación

En esta prueba realizada con webGoat hemos podido realizar las siguientes acciones:

1. Transferencias de dinero: si un usuario está logueado y tiene los permisos suficientes se puede aprovechar del security misconfiguration de la aplicación para subir archivos o enviar emails que contengan código malicioso. Al no tener filtros de seguridad, la aplicación confiara ciegamente en el usuario y este atacante podría robar dinero de otros usuarios.
2. Robo de identidad: el atacante podría robar las credenciales de otro usuario y ejecutar acciones como si fuera otra persona.
3. Escalada de privilegios: un atacante podría aprovechar la sesión de un administrador para darse a sí mismo privilegios de administrador haciendo un request al endpoint respectivo.

d. Posibles mitigaciones

1. Tokens desde el servidor anti csrf: el servidor podrá enviar un token único por cada sesión al usuario. El atacante no podrá leerlo por la política de los navegadores del SOP (same-origin policy), el token anti csrf permanece oculto para el atacante, así que aunque pueda visualizar la cookie de sesión, no podrá ejecutar alguna acción, porque el servidor verifica que el request contenga el token anti-csrf.
2. Captchas: en cada request que se hace de forma externa, las plataformas no exigen alguna acción adicional para evitar que se ejecute un script tan fácilmente. Si se coloca captcha como requisito para evitar operaciones no deseadas, el atacante necesariamente tendría que tener acceso dentro de la misma página para poder dar click en dicho captcha, pero al ser un ataque CSRF generalmente no tiene el

acceso, por lo que el request no procederá ya que el servidor verifica que el captcha no fue aprobado.

3. Doble verificación de operaciones: si en caso un atacante pueda, de alguna forma, enviar código malicioso al servidor para ejecutar alguna operación para su beneficio, el servidor podría implementar una verificación de permisos antes de ejecutar toda su lógica de negocio. Se podría usar emails de verificación, mensajes de texto, o generar OTP antes de ejecutar operaciones muy delicadas, como transferencias indeseadas.
4. Atributo samesite=strict: el navegador no enviará la cookie de sesión a no ser que la petición ocurra desde el mismo origen y no desde una fuente externa. Esto evita que el atacante pueda visualizar las cookies de sesión de los usuarios.

e. Resultado

Para un ataque CSRF la puntuación es de **6.5**
CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:H/A:N

Riesgo: medio

3.4. Vuln & outdated components (A03:2025 Software supply chain failures)

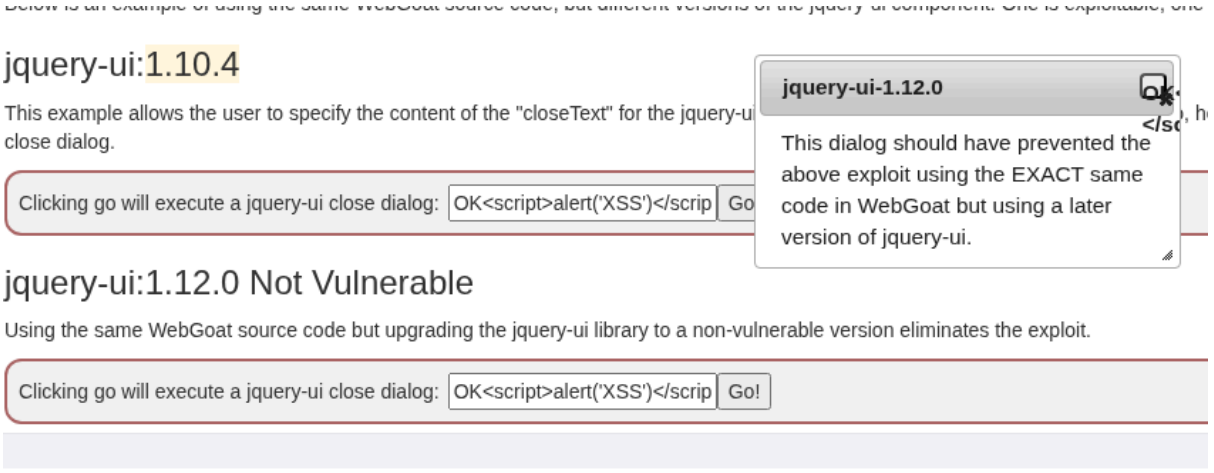
a. Reconocimiento de la vulnerabilidad

En OWASP nos hablan de la importancia de realizar un tracking constante a las dependencias y código que conforman la aplicación. Muchas veces las versiones de las dependencias que utilizan los desarrolladores son antiguas, lo que quiere decir que tienen vulnerabilidades no parcheadas; otras veces pueden tener dependencias actualizadas pero las sub-dependencias que utilizan dichas dependencias si tienen vulnerabilidades.

Las vulnerabilidades en la cadena de suministro no solo se basan en dependencias, otros componentes como el IDE que utiliza el desarrollador o la tecnología pipeline CI/CD y sus respectivas configuraciones pueden dejar expuesta otra vulnerabilidad.

En el caso de webGoat, en el apartado 5, nos expone una vulnerabilidad conocida de jquery-ui en su versión 1.10.4 (CVE-2016-7103) donde la propiedad closeText que es el texto que se muestra cuando se pasa el puntero sobre la “X”, ejecuta el código JS directo, por lo que se pueden ejecutar ataques XSS.

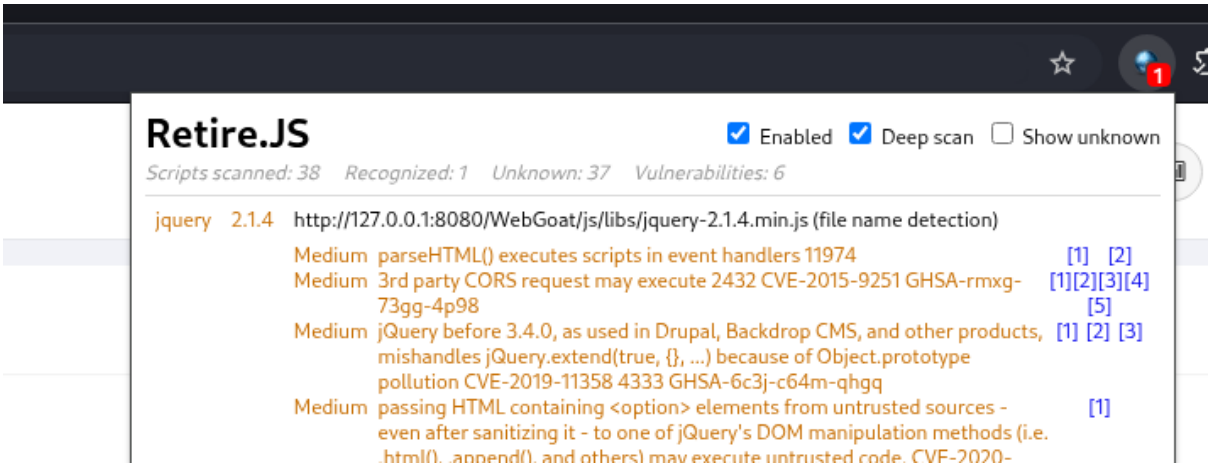
Esto se parcho en las versiones superiores a 1.12 de jquery-ui. A partir de esa versión el contenido del closeText es literal, tomado solo como texto plano, por lo que aunque escriban código malicioso, no se va ejecutar como pasaba en la versión 1.10.4



Esta vulnerabilidad afecta principalmente la **integridad** de los datos, porque el atacante podría modificar parte del DOM y robar credenciales de la víctima.

b. Explotación de la vulnerabilidad

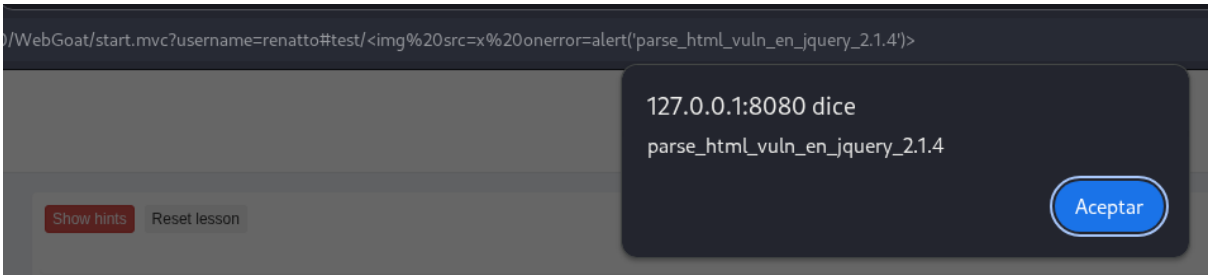
Utilizando herramientas para reconocimiento de dependencias, usamos retire.js y vemos que webGoat está usando jquery 2.1.4 la cual tiene una vulnerabilidad clara que permite un ataque DOM-based-XSS.



En secciones anteriores había una ruta adicional agregada a producción llamada /test. Por defecto backbone está agregando el sufijo .lesson cuando se trata de rutas que son lecciones pero cuando no es asi, solo envia el request al servidor y la respuesta obtenida debe agregarse al DOM usando parseHTML, ya que se está usando jquery 2.1.4.

Usando las dev tools, en el DOM se puede observar que el div principal es lesson-content. Si se revisa la documentación de jquery y de lo que hace parseHTML hay una posibilidad de que se esté inyectando la respuesta del servidor dentro de un parseHTML que está a su vez dentro de este div lesson-content.

La forma de verificar que la vulnerabilidad expone este posible ataque XSS es la siguiente. Como backbone envía los parámetros de /test/:param al servidor y este los incluye en la respuesta al frontend, inyectamos una etiqueta html con un alert. Si el funcionamiento es como se está describiendo, entonces el alert se ejecutará.



Al poner el código malicioso en la URL de /test/:param y en todas aquellas rutas en las que backbone esté enviando parámetros del tipo /esta_ruta/:estos_param, un atacante podría escribir un script más elaborado y usar acortadores de URL o hacer un encoding para que la víctima no sospeche de que es lo que está ejecutando y de esta manera poder robar información **confidencial** o incluso reemplazar un formulario oficial de la página que está visitando para enviar dinero de una compra a su cuenta personal.

c. Post-explotación

Con vulnerabilidades en la cadena de suministro, un atacante podría realizar las siguientes acciones:

1. Robo de credenciales: podría crear un formulario y enviarlo a una víctima como si fuera un formulario oficial de la plataforma.
2. Capturar eventos de teclado (keylogger): podría montar un servidor que capture los eventos del teclado de algún usuario y obtener así, no solo credenciales, sino información sensible de la empresa o de otros usuarios, como emails, teléfonos, etc.
3. Redireccionar rutas a otro servidor propio del atacante.
4. XSS almacenado: hay la posibilidad de que se convierta en un XSS almacenado si consideramos que el servidor de la base de datos no valida los datos enviados por el usuario y confían en que son buenos usuarios, como se vio en la sección de ataques CSRF. Si hay algún evento de inserción en tabla de datos y una tabla que muestra los datos insertados en frontend, jquery 2.1.4 renderiza las filas usando parseHTML leyendo la información de la base de datos, esto sería catastrófico porque se ejecutaría masivamente para todos los usuarios y de manera automática.

d. Posibles mitigaciones

1. Actualizar las dependencias a sus últimas versiones: así se cubren parches de seguridad que las versiones antiguas no tenían.
2. En caso de que las últimas versiones también presenten alguna vulnerabilidad, usar una dependencia alternativa o crear una versión personalizada que evite ejecutar dicha vulnerabilidad o cubrirla.
3. Configurar los pipeline de CI/CD con las mejores prácticas de seguridad e incluir los flags más seguros recomendados por las mismas herramientas.
4. Eliminar dependencias en desuso o código legacy, solo incluir las dependencias en uso y verificar si se pueden reducir el número de dependencias, pues muchas veces hay funciones nativas de los mismos lenguajes de programación o funciones ya incluidas en librerías ya instaladas, que hace que otras dependencias se vuelvan innecesarias. Menos dependencias, mayor eficiencia de la plataforma y menos posibles vulnerabilidades expuestas.
5. Al momento de instalar nuevas dependencias, verificar que sean de fuentes oficiales y usar versiones específicas. Por ejemplo en docker se utilizan los hashes para especificar una versión en específico de una imagen, de esta forma, se evita usar una imagen manipulada.
6. Revisar constantemente fuentes como la NVD, OSV y CVE: con el fin de evitar usar código con vulnerabilidades conocidas. Por ejemplo, el uso de eval, es una vulnerabilidad muy conocida, que se debe evitar en cualquier proyecto.
7. Se puede usar herramientas de software o extensiones de navegador(retire.js) que analicen las vulnerabilidades existentes de las dependencias utilizadas.

e. Resultado

Para ataques XSS basados en fallas de la cadena de suministro, se obtiene una puntuación: **6.1**

CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N

Riesgo: medio

3.5. Secure Passwords (A07:2025 Authentication Failures)

a. Reconocimiento de la vulnerabilidad

NIST (Nacional institute of standards and technology) tiene una guia para crear passwords seguras e inducir a los usuarios a crear una.

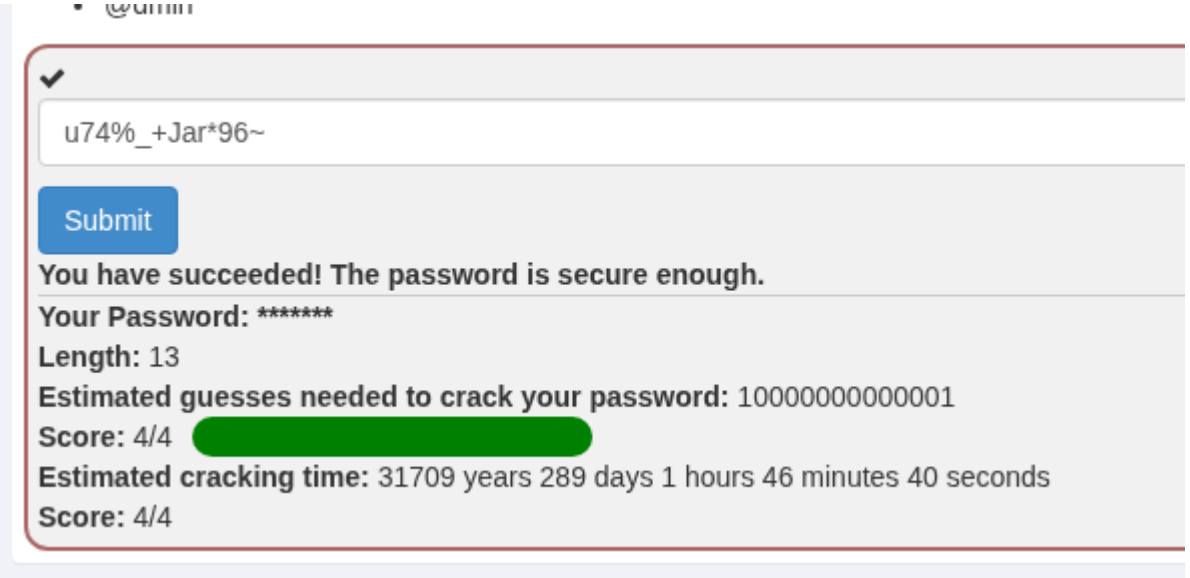
En webGoat tenemos credenciales de usuario al momento del login y es importante que los usuarios sigan los lineamientos del NIST, y para eso la plataforma en cuestión debe darle la experiencia de usuario necesaria para cubrir los lineamientos y evitar malas prácticas.

En la sección de secure passwords, webGoat nos reta a crear una password segura. Se debe considerar que el password cumpla con lo siguiente:

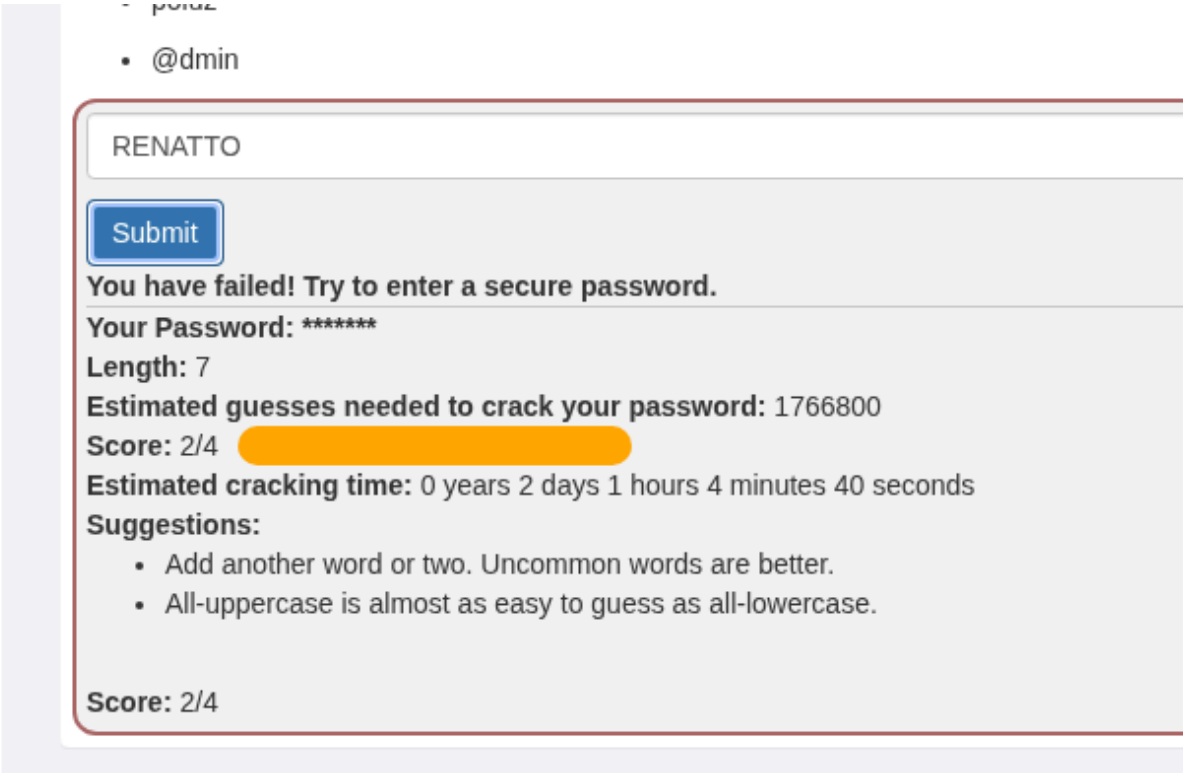
- Evitar patrones repetitivos o predecibles. Ej: aaa1, abcd1234, aB12cD, etc
- El password debe tener una longitud mínima de 8 caracteres.
- Debe contener caracteres UNICODE, los espacios en blanco no son excluyentes. Ej: ~#23 J%*a
- Evitar usar información personal y pública, como el nombre del servicio, la empresa, el nombre de usuario o de algún familiar, la edad, etc.

Para consultar todos los lineamientos del NIST se puede visitar la guia del NIST 800-63b's.

En el ejercicio de webGoat, cumplimos con una password segura, siguiendo los lineamientos NIST:



Pero qué pasa cuando usamos el nombre de usuario como password:



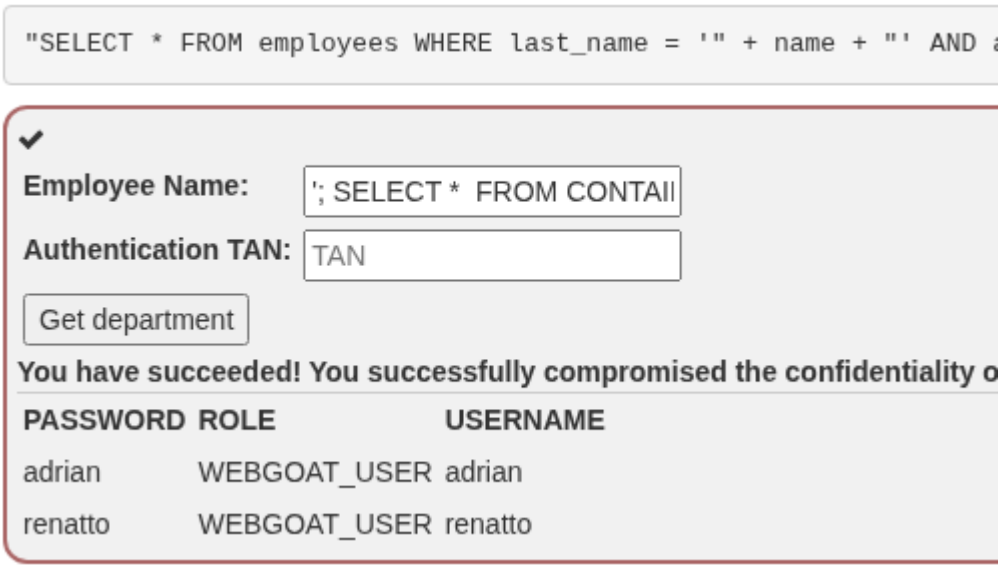
La vulnerabilidad queda expuesta, no por el código, no por las dependencias, sino por mala práctica del usuario y por la falta de experiencia de usuario que ofrece la plataforma al registrar usuarios. Esta vulnerabilidad compromete la confidencialidad de los datos de las víctimas y la integridad de sus datos dentro de la plataforma.

b. Explotación de la vulnerabilidad

Existen herramientas como John the ripper, que usando un comando puede crackear un password fácilmente. Pero para hacer esto deberíamos tener el hash del password y esto se logró en secciones anteriores con SQLi. Se comprobó que los usuarios tienen acceso a la base de datos y por lo tanto pueden visualizar la tabla con la lista de usuarios en webGoat.

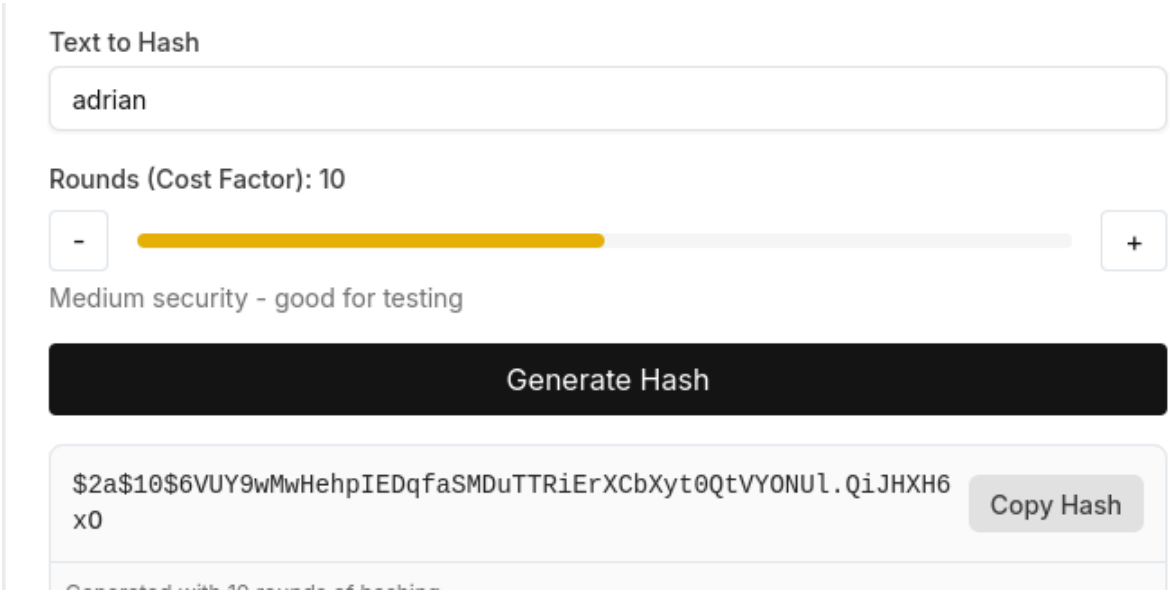
`'; SELECT *  FROM CONTAINER.WEB_GOAT_USER–`

Con esta consulta, podemos ver lo siguiente:



Como vemos webGoat, no hashea el password, por lo que cualquier usuario podría obtener las credenciales de los demás usuarios. Pero para probar la vulnerabilidad del secure passwords, supondremos que están hasheados en formato bcrypt.

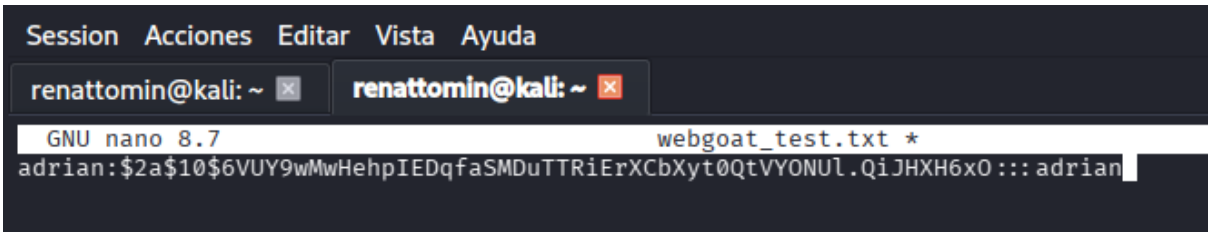
Con una herramienta online bcrypt-generator: https://bcrypt-generator.com/



El password adrian pasa a ser $2a$10$6VUY9wMwHehpIEDqfaSMDuTTRiErXCbXyt0QtVYONUl.QiJHXH6xO en la base de datos webGoat.

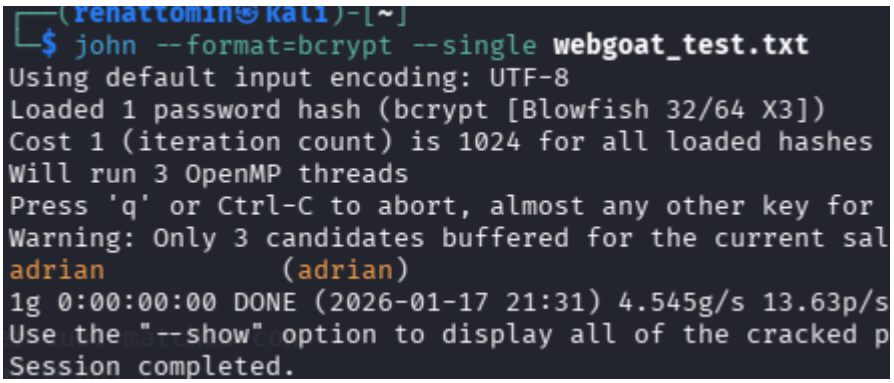
Obteniendo esta información podemos usar Jhon the ripper para crackear el password. Usando otra herramienta online podemos determinar que tipo de hash es el que tiene la base de datos. https://hashes.com/en/tools/hash_identifier

Y como se puede observar, identifican que el hash es de tipo bcrypt. John the ripper requiere un archivo de texto en formato GECOS, donde colocas primero el usuario y luego el hash.

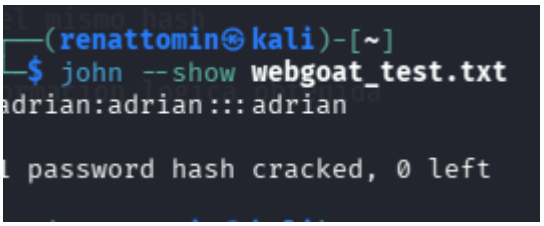


usuario:hash:uid:gid:nombre:email

Ahora corremos john the ripper para que intente comprobar diferentes combinaciones con la información brindada y ver si alguna combinación hace match con el hash en formato bcrypt.



Y como se puede apreciar, fue sencillo de obtener el password porque el usuario adrian usó su mismo nombre como password. Hubiera sido el mismo resultado si hubiera agregado adrian123 o adrianABC123, etc. Podemos comprobar que el password fue crackeado con el parámetro –show de john the ripper.



En este caso, se facilitó el ataque por el SQLi previamente expuesto, pero pudo haber sido un ataque MITM que intercepte el tráfico y obtenga data importante para crear diccionarios y luego ejecutarlos con otras herramientas. El resultado hubiera sido el mismo por el nivel de seguridad del password. Incluso una persona que haga footprinting en su navegador acerca de una víctima, podría armar igual un diccionario y ejecutar un ataque de fuerza bruta en las plataformas que usa la víctima. El resultado será el mismo y otra vez, por el nivel de seguridad de las passwords utilizadas.

Esto compromete la **confidencialidad** en un nivel muy alto y la **integridad**. Incluso la **disponibilidad** se ve comprometida, pues el atacante podría eliminar las cuentas de la víctima si lo desea, aunque hay medidas de seguridad para esto puede darse el caso de que sean medidas débiles o el atacante también haya obtenido el password del correo de la víctima.

c. Post-explotación

El atacante podría realizar las siguientes acciones:

1. Ejecutar operaciones haciéndose pasar por la víctima.
2. Vender credenciales y datos personales de las víctimas
3. Robar la identidad de la víctima para cometer actos fraudulentos.
4. Vaciar cuentas bancarias anexadas a las plataformas a las que tiene acceso.
5. Manipular información sensible en beneficio propio. Por ejemplo, en una universidad podría cambiar notas, si obtiene el password de algún profesor.

d. Posibles mitigaciones

1. Cambiar la experiencia de usuario de la plataforma para guiar al usuario a definir passwords más seguros.
2. Agregar captchas al inicio de sesión para evitar que los atacantes usen ataques automatizados.
3. Limitar el número de veces de intento de login.
4. Implementar logs y monitoreo para identificar desde donde intentan hacer ataques y bloquear el acceso a los posibles atacantes. En caso de ser usuarios oficiales, usarán los medios oficiales de la plataforma para contactarse con soporte y/o recuperar el password.
5. Multi factor authentication: en el peor de los casos, que el atacante obtenga las credenciales de una víctima, no podrá acceder sin pasar la doble autenticación.

e. Resultado

Un ataque por la vulnerabilidad de insecure passwords obtiene una puntuación de: **9.0**

`CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:C/C:H/I:H/A:H`

Riesgo: crítico

4. Herramientas utilizadas

1. Burp Suite: se usó como proxy para interceptar los requests y responses mientras se navegaba en webGoat.
2. Sqlmap: se usó como information gathering para detectar si webGoat era vulnerable a sql injection y obtener además algunos datos de su base de datos.
3. Curl: ejecutó requests desde una fuente externa diferente al mismo origen de webGoat, probando que era vulnerable a ataques CSRF.
4. DevTools: permitió visualizar el código fuente de webGoat e identificar las rutas de backbone.js.
5. Retire.js: extensión del navegador que utilizamos para identificar dependencias con vulnerabilidades.
6. John the ripper: herramienta de comandos para crackear passwords según los hashes generados en bases de datos.
7. bcrypt-generator: herramienta online que genera un hash bcrypt según un texto plano.
8. hash-identifier: aplicación online que identifica el formato de un hash.
9. CVSS(Common vulnerability scoring system version 3.1): calculadora para clasificar vulnerabilidades.

5. Referencias

bcryp-generator: https://bcrypt.online/

hash-identifier: https://hashes.com/es/tools/hash_identifier

CVSS: https://www.first.org/cvss/calculator/3.1

OWASP: https://owasp.org/Top10/2025/

CVE (Common vulnerabilities and exposure): https://www.cve.org/

Jquery docs: https://api.jquery.com/