# EXPERIMENT – 01

## AIM:

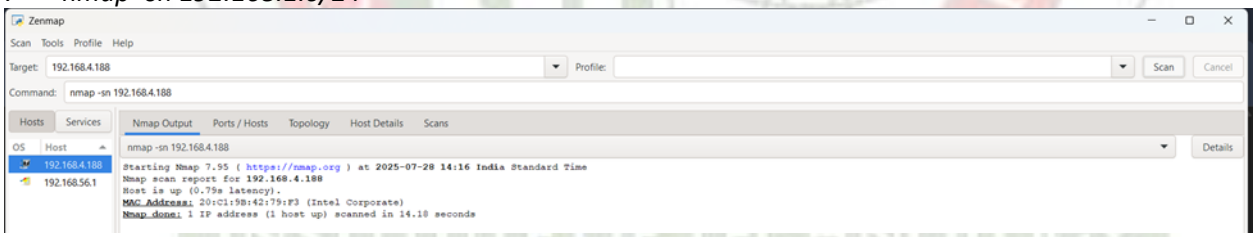
To use Nmap for scanning hosts and identifying services, and to analyze scan behavior using Wireshark to understand TCP flags, ICMP packets, UDP services, and OS detection.

## DESCRIPTION:

Nmap is a powerful tool for network discovery and security auditing. It allows scanning of hosts, services, and operating systems. Wireshark captures and analyzes the network traffic generated by these scans, helping us understand how different scan types work internally.
In this experiment, we perform different Nmap scans (ping sweep, SYN scan, service detection, UDP scan, aggressive scan, all-port scan, and OS detection) and capture the packets with Wireshark. By applying filters, we observe ICMP, TCP flags, UDP queries, and OS detection patterns.

## PROCEDURE:

**Step 1 – Host Discovery (Ping Sweep)**
*.>>> nmap -sn 192.168.1.0/24*



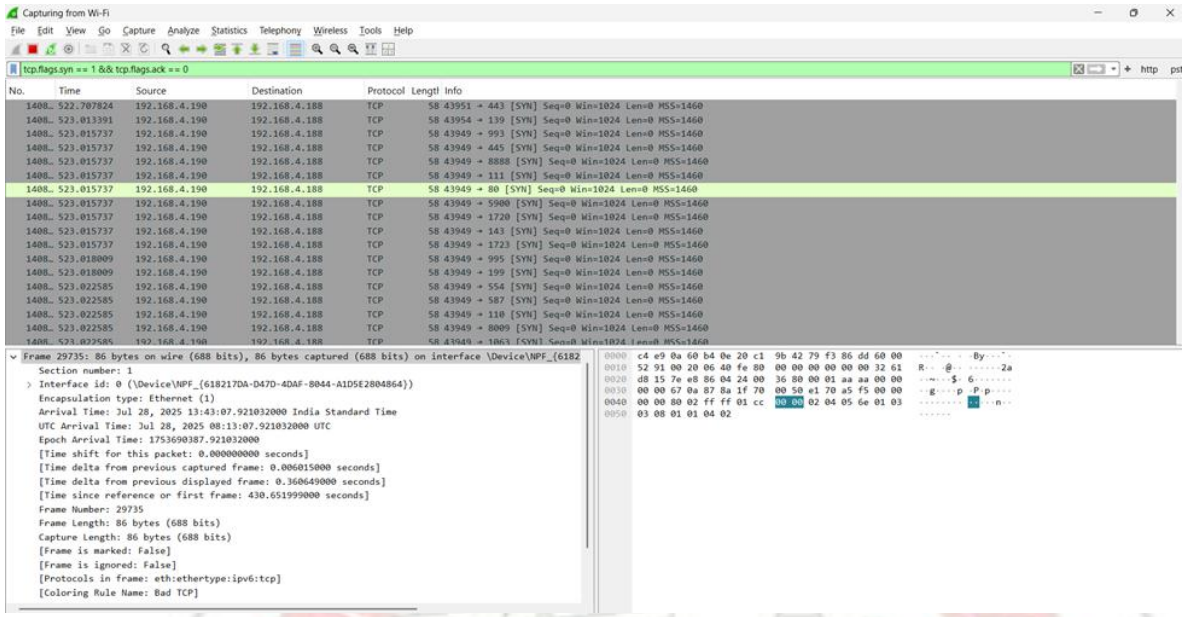

- *Discovers live hosts using ICMP echo requests*
- *Wireshark Filter: icmp*

**Step 2 – Stealth SYN Scan**
**>>>** *nmap -sS <target_ip>*

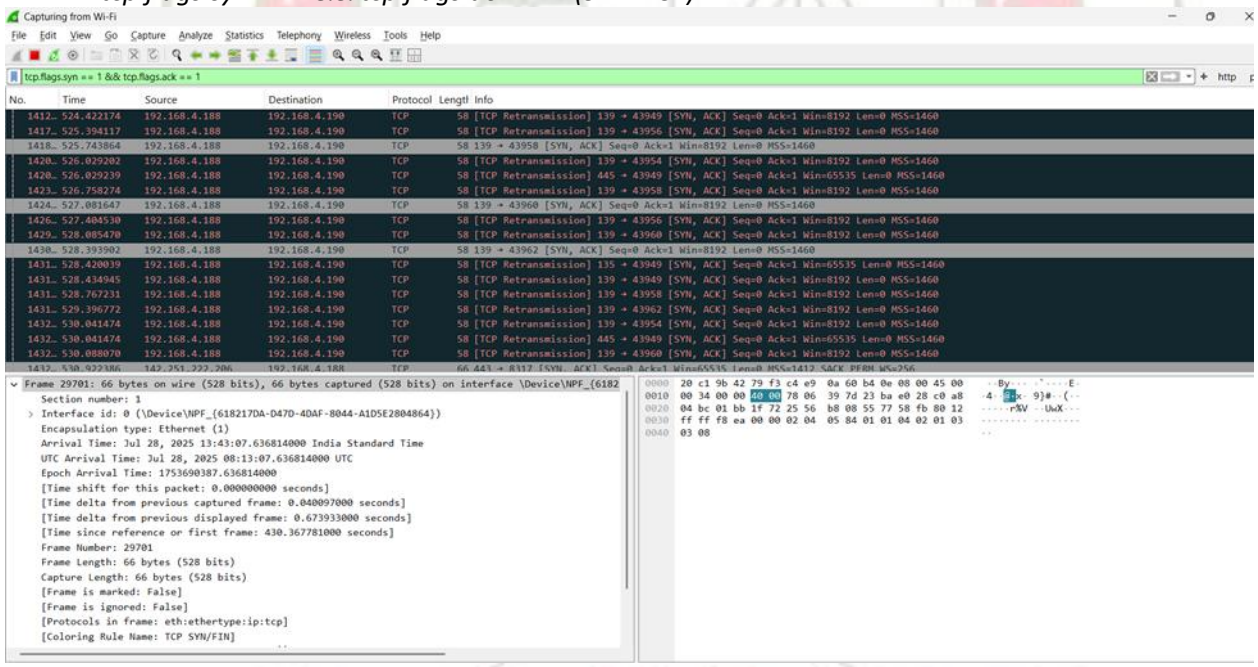
*Sends SYN packets but does not complete the handshake.*

***Wireshark Filters:***

- *tcp.flags.syn == 1 && tcp.flags.ack == 0 (SYN)*



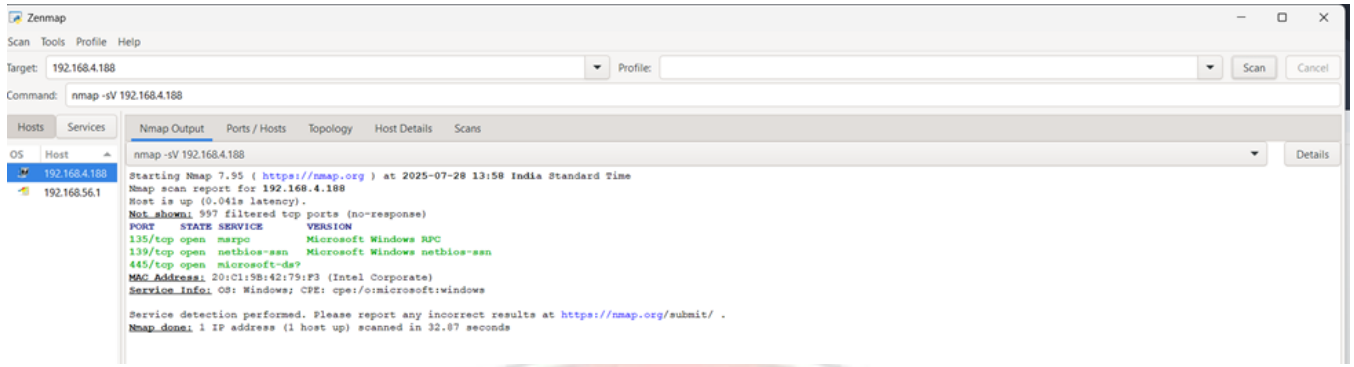

- *tcp.flags.syn == 1 && tcp.flags.ack == 1 (SYN-ACK)*




- *tcp.flags.reset == 1 (RST for closed ports)*

**Step 3 – Service and Version Detection**

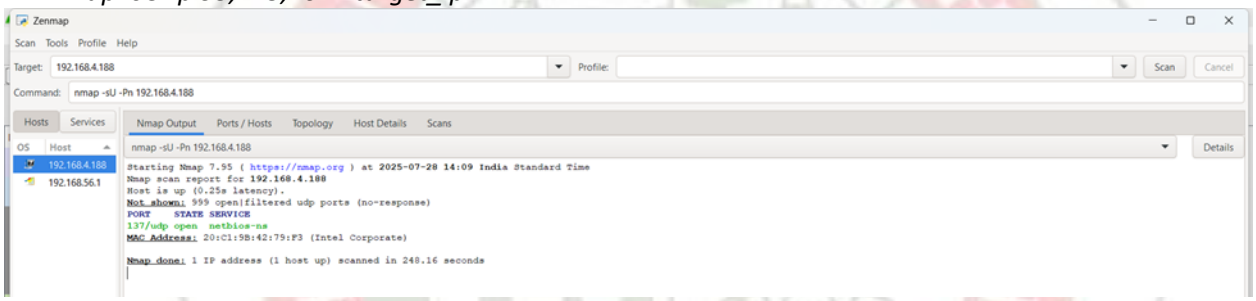
*>>>nmap -sS -sV <target_ip>*



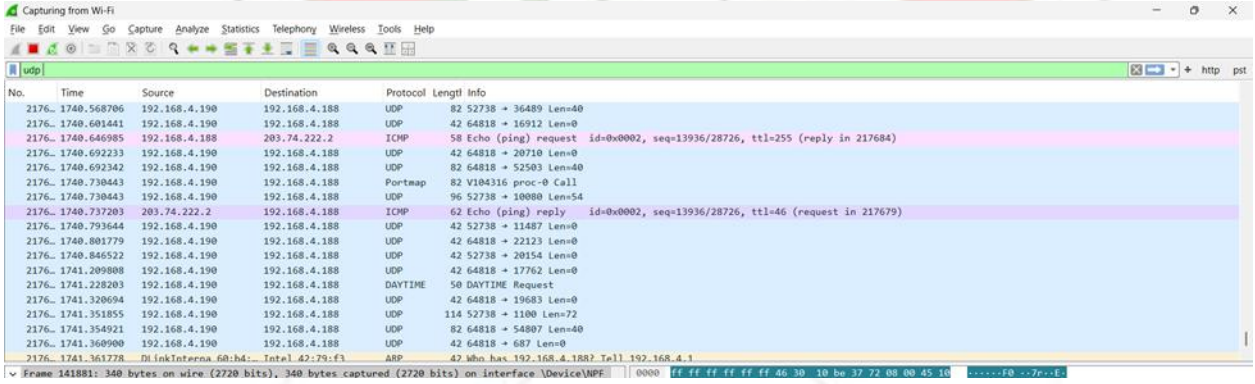

- *Identifies services and versions running on open ports.*
- ***Wireshark Filter:*** *tcp && ip.addr == <target_ip>*

**Step 4 – UDP Scan**

**>>>***nmap -sU -p 53,123,161 <target_ip>*

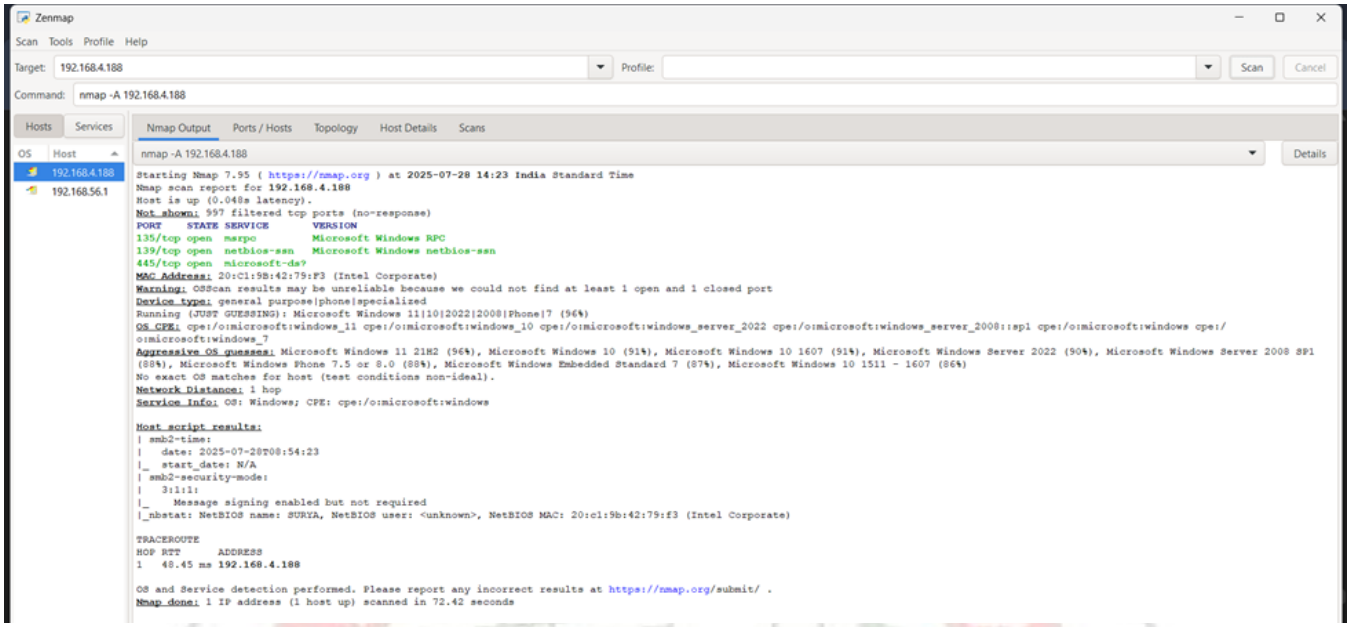



- Tests UDP services (DNS, NTP, SNMP).
- Wireshark Filter: udp



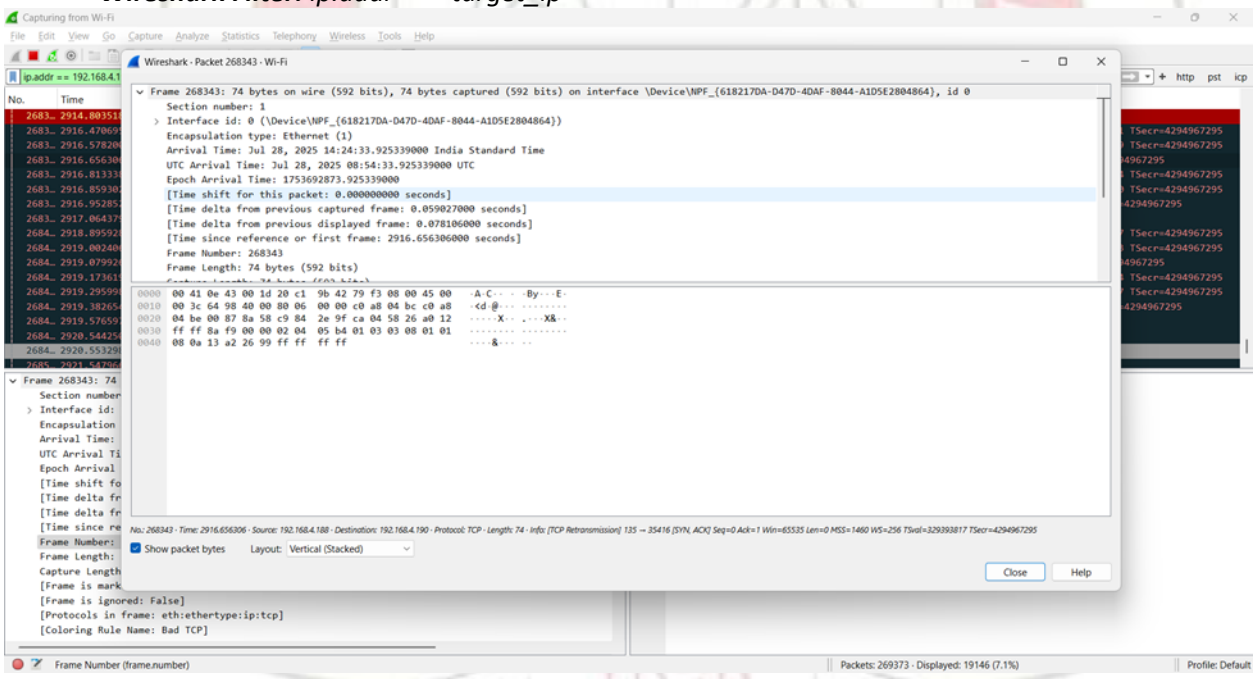

**Step 5 – Aggressive Scan**

**>>>** *nmap -A <target_ip>*




- *Performs OS detection, version detection, script scanning, and traceroute.*

- ***Wireshark Filter:*** *ip.addr == <target_ip>*




***Step 6 – Scan All Ports***

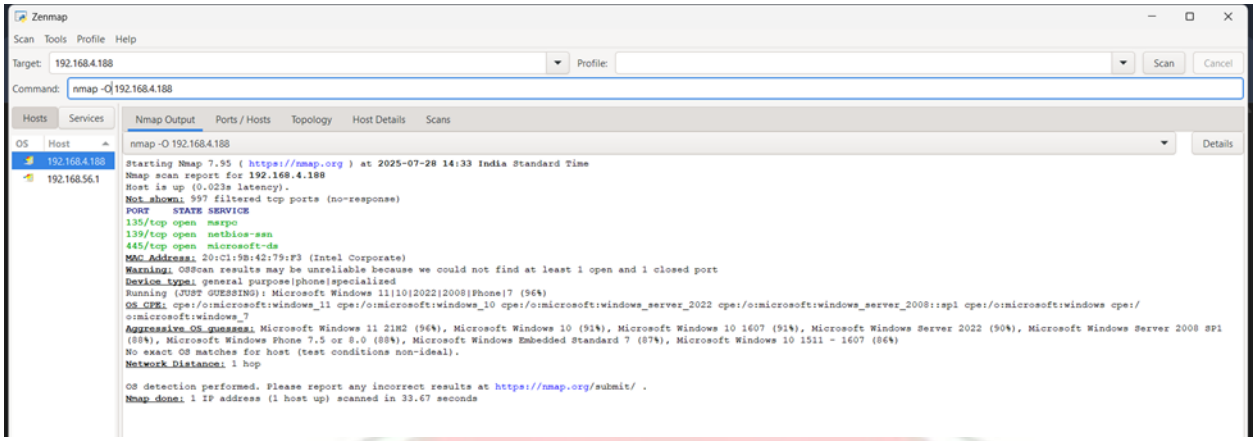
*>>> nmap -p- <target_ip>*

- Scans all 65535 TCP ports.

- **Wireshark Filter:** ip.addr == <target_ip>

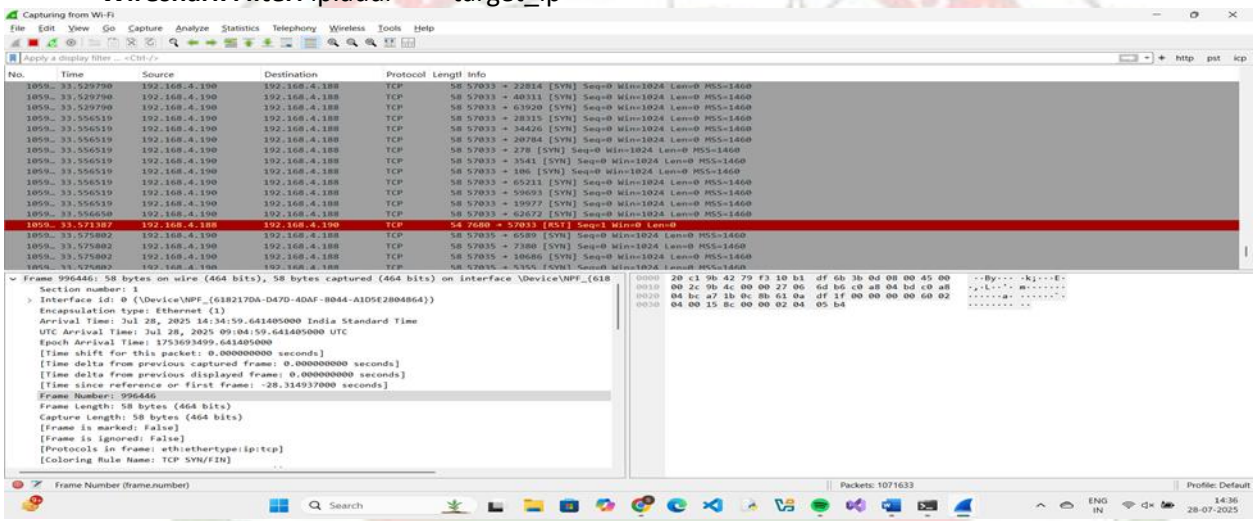
**Step 7 – OS Detection**

*>>>nmap -O <target_ip>*




- Detects the target's operating system.

- **Wireshark Filter:** ip.addr == <target_ip>




## ***OUTPUT ANALYSIS:***

- Ping sweep revealed live hosts in the local subnet using ICMP packets.
- SYN scan showed TCP handshakes (SYN, SYN-ACK, RST) without completing connections.
- Service/version detection identified open ports and software versions.
- UDP scan generated traffic on DNS, NTP, and SNMP ports, showing how UDP differs from TCP.
- Aggressive scan produced detailed OS, service, and traceroute results.
- All-port scan listed every possible TCP port, identifying open/closed states.
- OS detection provided an estimate of the target OS based on packet responses.

Wireshark confirmed these behaviors through packet captures and applied filters, showing how Nmap interacts with target hosts.

**CONCLUSION:**
This experiment demonstrated how to use Nmap for different types of scans and analyze the corresponding network traffic with Wireshark. By observing TCP flags, ICMP responses, UDP traffic, and OS fingerprints, we gained practical insight into how scanning techniques work and how they can be detected or defended against in real-world cybersecurity.

# EXPERIMENT – 02

## *AIM:*

Configure firewall rules with UFW or iptables and test using nmap,
or netcat, or Metasploit.

## *DESCRIPTION:*

In this experiment, we created Windows Firewall rules to allow and block specific TCP ports, then verified their behavior using Ncat (as a listener) and Nmap (as a port scanner). An allow rule was applied to port 5555, and a block rule was applied to port 4444. Ncat successfully listened on port 5555, and an Nmap scan confirmed that port 5555 was open while port 4444 was blocked. This demonstrated how firewall rules can be used to control network access.

## *PROGRAM:*

**Step-1: Open PowerShell as Administrator**

- Required to create firewall rules; otherwise, commands fail with “Access denied.”

**Step-2: Create Firewall Allow Rule for Port 5555**

*>>> New-NetFirewallRule -DisplayName "Allow TCP 5555" -Direction Inbound -Protocol TCP -LocalPort 5555 -Action Allow*

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\WINDOWS\system32> New-NetFirewallRule -DisplayName "Allow TCP 5555" -Direction Inbound -Protocol TCP -LocalPort 5555 -Action Allow
>>


Name                          : {2007e1f3-cced-441f-9a90-78c54b653449}
DisplayName                   : Allow TCP 5555
Description                   :
DisplayGroup                  :
Group                         :
Enabled                       : True
Profile                       : Any
Platform                      : {}
Direction                     : Inbound
Action                        : Allow
EdgeTraversalPolicy           : Block
LooseSourceMapping            : False
LocalOnlyMapping              : False
Owner                         :
PrimaryStatus                 : OK
Status                        : The rule was parsed successfully from the store. (65536)
EnforcementStatus             : NotApplicable
PolicyStoreSource             : PersistentStore
PolicyStoreSourceType         : Local
RemoteDynamicKeywordAddresses : {}
PolicyAppId                   :
PackageFamilyName             :
```

- Allows inbound TCP connections on port 5555.

**Step-3: Create Firewall Block Rule for Port 4444**

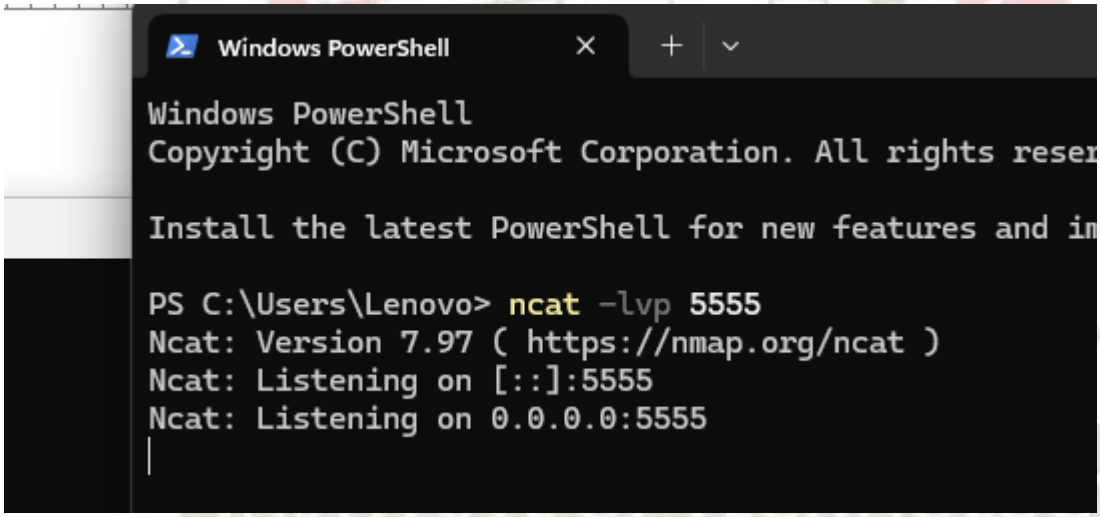
*>>> New-NetFirewallRule -DisplayName "Block TCP 4444" -Direction Inbound -Protocol TCP -LocalPort 4444 -Action Block*

```
PS C:\WINDOWS\system32> New-NetFirewallRule -DisplayName "Block TCP 4444" -Direction Inbound -Protocol TCP -LocalPort 4444 -Action Block
>>


Name                            : {89a7cb41-65c8-414e-a26a-b23492bcf3da}
DisplayName                     : Block TCP 4444
Description                     :
DisplayGroup                    :
Group                           :
Enabled                         : True
Profile                         : Any
Platform                        : {}
Direction                       : Inbound
Action                          : Block
EdgeTraversalPolicy             : Block
LooseSourceMapping              : False
LocalOnlyMapping                : False
Owner                           :
PrimaryStatus                   : OK
Status                          : The rule was parsed successfully from the store. (65536)
EnforcementStatus               : NotApplicable
PolicyStoreSource               : PersistentStore
PolicyStoreSourceType           : Local
RemoteDynamicKeywordAddresses   : {}
PolicyAppId                     :
PackageFamilyName               :
```

- Blocks inbound TCP connections on port 4444.

**Step-4: Start Listener on Port 5555 with Ncat**

**>>>***ncat -lvp 5555*




- Ncat successfully listens for incoming connections on port 5555.

**Step-5: Scan Ports with Nmap**

**>>>***nmap -p 5555,4444 127.0.0.1*

```
PS C:\WINDOWS\system32>
PS C:\WINDOWS\system32>
PS C:\WINDOWS\system32> nmap -p 5555,4444 127.0.0.1
>>
Starting Nmap 7.97 ( https://nmap.org ) at 2025-08-18 14:27 +0530
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00s latency).

PORT     STATE  SERVICE
4444/tcp closed krb524
5555/tcp open   freeciv

Nmap done: 1 IP address (1 host up) scanned in 0.17 seconds
PS C:\WINDOWS\system32>
```

***Expected results:***

- Port 5555 → Open (allowed).
- Port 4444 → Closed/Filtered (blocked).

## ***OUTPUT ANALYSIS:***

The Nmap scan showed port 5555 open, proving the "Allow TCP 5555" rule worked. Port 4444 appeared closed, confirming the "Block TCP 4444" rule. This verified that Windows Firewall rules were applied correctly and controlled inbound traffic as expected.

## ***CONCLUSION:***

The experiment successfully demonstrated how to configure firewall rules in Windows using PowerShell and verify them with Ncat and Nmap. By allowing one port and blocking another, we confirmed that firewall rules can effectively manage and restrict network access..

# EXPERIMENT – 03

## *AIM:*

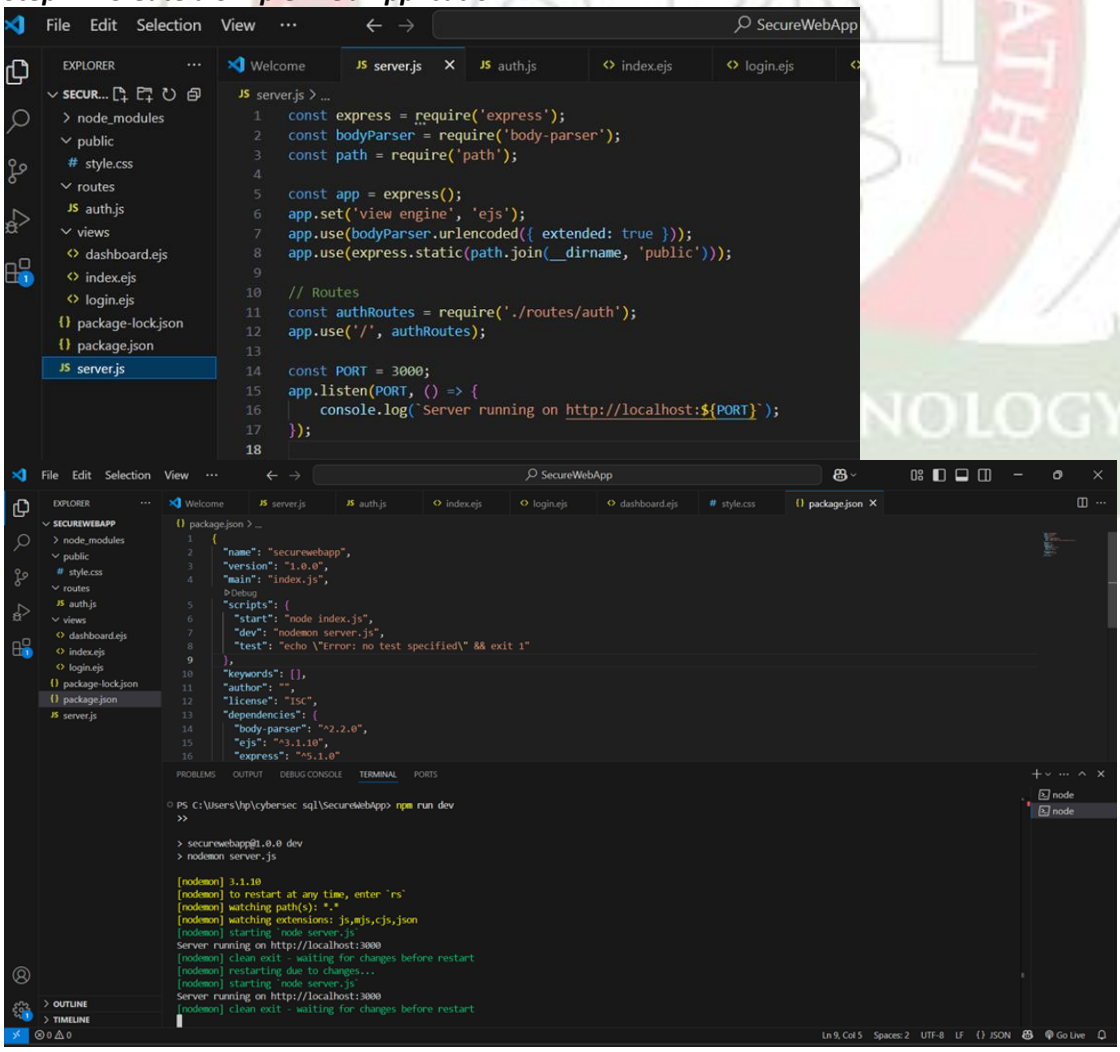
To develop a simple web application, analyze it using OWASP ZAP or Burp Suite, and identify common web vulnerabilities such as SQL injection, Cross-Site Scripting (XSS), or insecure authentication.

## *DESCRIPTION:*

Web applications are frequently targeted by attackers due to insecure coding practices and misconfigurations. Tools like OWASP ZAP and Burp Suite are widely used for penetration testing and vulnerability assessment.
In this experiment, we first build a simple test web app and then run scans using OWASP ZAP or Burp Suite. The goal is to detect issues such as SQL injection points, reflected XSS, missing security headers, and broken authentication.

## *PROCEDURE:*

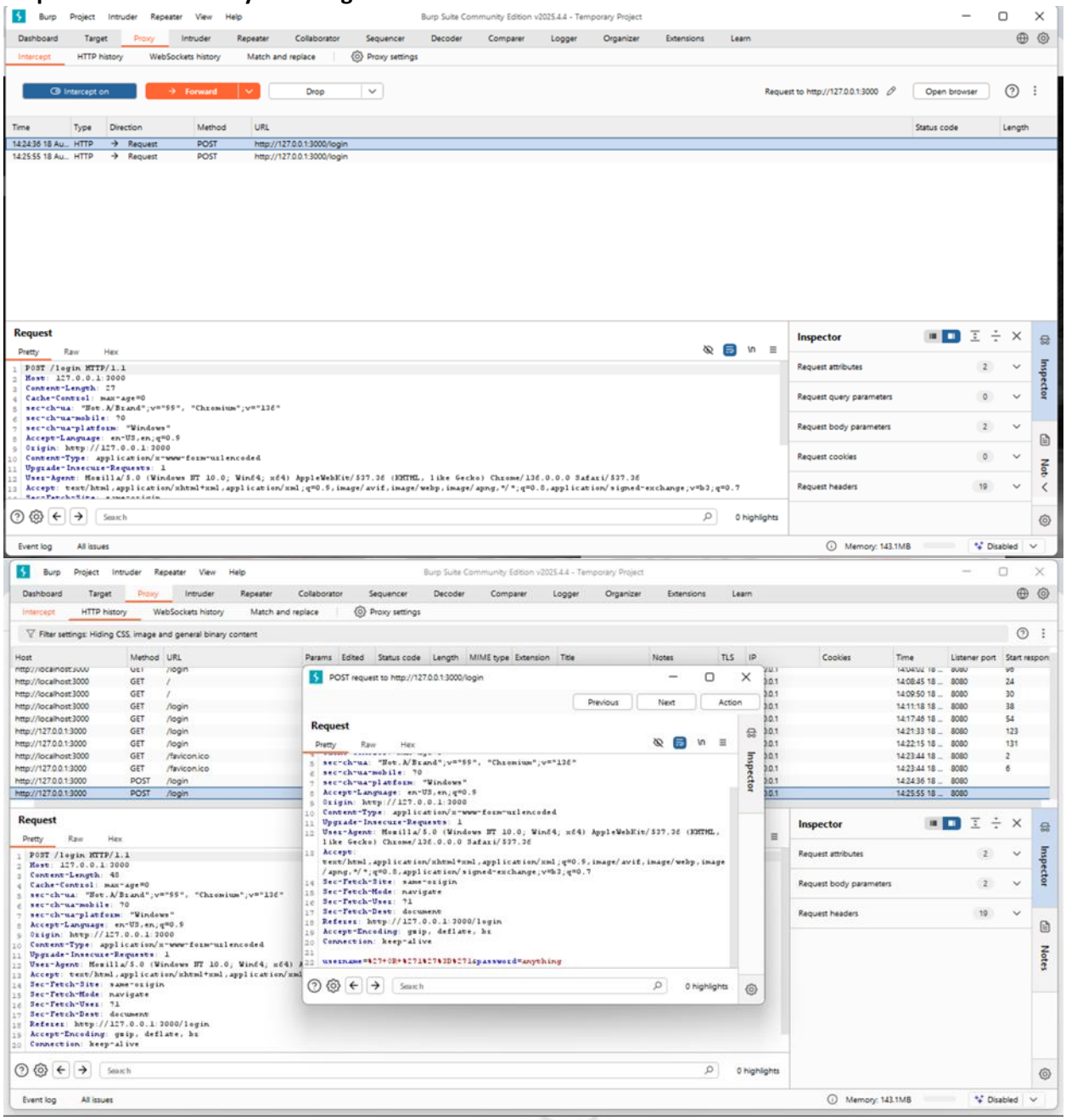
### *Step 1 – Create a Simple Web Application*



**Step 2: Setup Burp Suite**

- Install Burp Suite (community edition).
- Configure the browser to use the proxy provided by the tool to intercept and analyze traffic.

**Step 3: Perform Security Scanning**



**Step 4: Analyze and Understand Vulnerabilities**

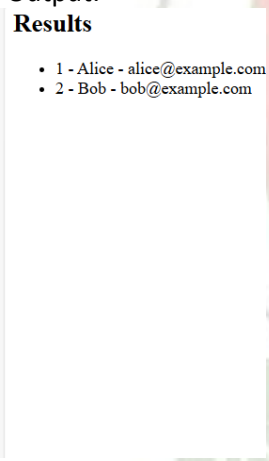
- Examine the vulnerabilities detected (e.g., input fields vulnerable to injection or missing headers).
- Understand the root causes and potential risks associated with each issue.

**Step 5: Implement Secure Coding Practices**

- Fix the issues by validating and sanitizing user inputs.
- Use prepared statements for database queries.
- Implement proper authentication and session management.

**Vulnerable:**


Output:

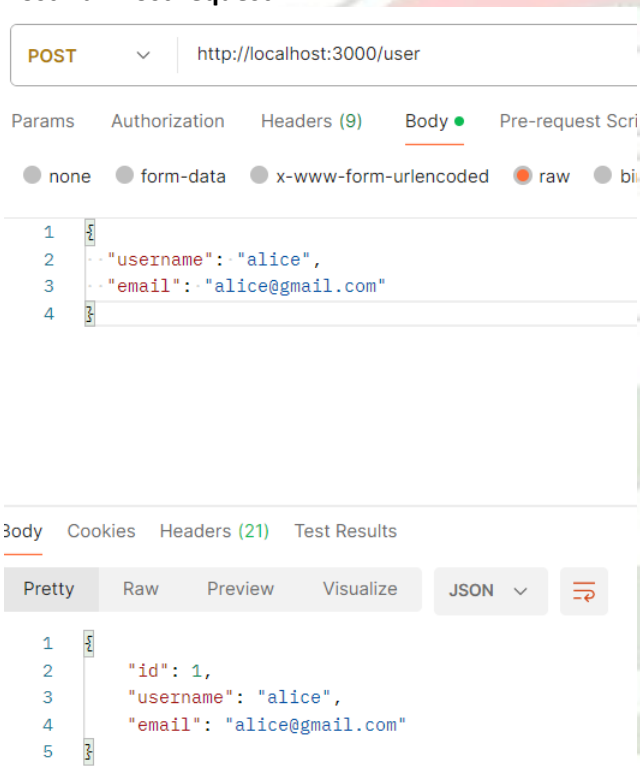



**Not Vulnerable:**


Output:

## Results

**Postman Post request:**




**Prototype Pollution**

Trick the app into adding dangerous properties globally.

```
PS C:\WINDOWS\system32> Invoke-WebRequest -Uri http://localhost:3000/user -Method POST -ContentType "application/json" -
Body '{"username":"hacker","email":"hack@example.com","__proto__":{"isAdmin":true}}'


StatusCode        : 201
StatusDescription : Created
Content           : {"id":13,"username":"hacker","email":"hack@example.com"}
RawContent        : HTTP/1.1 201 Created
                    Content-Security-Policy: default-src 'self';base-uri 'self';font-src 'self' https:
                    data:;form-action 'self';frame-ancestors 'self';img-src 'self' data:;object-src 'none';script-s...
Forms             : {}
Headers           : {[Content-Security-Policy, default-src 'self';base-uri 'self';font-src 'self' https:
                    data:;form-action 'self';frame-ancestors 'self';img-src 'self' data:;object-src 'none';script-src
                    'self';script-src-attr 'none';style-src 'self' https: 'unsafe-inline';upgrade-insecure-requests],
                    [Cross-Origin-Opener-Policy, same-origin], [Cross-Origin-Resource-Policy, same-origin],
                    [Origin-Agent-Cluster, ?1]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 56
```

**Denial of Service (DoS) – Large JSON Payload**
This will try to overwhelm your server with a huge request.

```
PS C:\WINDOWS\system32> Invoke-WebRequest -Uri http://localhost:3000/user -Method POST -ContentType "application/json" -
Body (@{username=("A"*10000000); email="test@test.com"} | ConvertTo-Json -Compress)
Invoke-WebRequest : PayloadTooLargeError: request entity too large    at readStream (C:\Users\Krishna
Tarun\webapp\node_modules\raw-body\index.js:163:17)    at getRawBody (C:\Users\Krishna
Tarun\webapp\node_modules\raw-body\index.js:116:12)    at read (C:\Users\Krishna
Tarun\webapp\node_modules\body-parser\lib\read.js:74:3)    at jsonParser (C:\Users\Krishna
Tarun\webapp\node_modules\body-parser\lib\types\json.js:125:5)    at Layer.handleRequest (C:\Users\Krishna
Tarun\webapp\node_modules\router\lib\layer.js:152:17)    at trimPrefix (C:\Users\Krishna
Tarun\webapp\node_modules\router\index.js:342:13)    at C:\Users\Krishna
Tarun\webapp\node_modules\router\index.js:297:9    at processParams (C:\Users\Krishna
Tarun\webapp\node_modules\router\index.js:582:12)    at next (C:\Users\Krishna
Tarun\webapp\node_modules\router\index.js:291:5)    at Function.handle (C:\Users\Krishna
Tarun\webapp\node_modules\router\index.js:186:3)
At line:1 char:1
+ Invoke-WebRequest -Uri http://localhost:3000/user -Method POST -Conte ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (System.Net.HttpWebRequest:HttpWebRequest) [Invoke-WebRequest], WebExc
   eption
    + FullyQualifiedErrorId : WebCmdletWebResponseException,Microsoft.PowerShell.Commands.InvokeWebRequestCommand
```

## ***OUTPUT ANALYSIS:***

The security scan using OWASP ZAP/Burp Suite detected common vulnerabilities such as SQL injection points, reflected XSS, missing security headers, and weak authentication mechanisms in the test web application. By exploiting these issues, it was possible to demonstrate risks like unauthorized data access, code injection, and session hijacking. After applying secure coding practices (input validation, sanitization, prepared statements, and strong authentication), the vulnerabilities were no longer exploitable, showing the effectiveness of remediation.

## ***CONCLUSION:***

This experiment highlighted the importance of identifying and mitigating web application vulnerabilities. Tools like OWASP ZAP and Burp Suite effectively revealed security flaws, while secure coding practices ensured that the application became resilient to attacks. It reinforced that continuous testing, validation, and adherence to security best practices are essential for building secure web applications.

# EXPERIMENT – 07

## AIM:

Use OpenSSL for Encryption/Decryption and Configure SSL/TLS on a Web Server.

## DESCRIPTION:

This experiment demonstrates both the cryptographic and secure communication capabilities of OpenSSL. In the first part, OpenSSL is used to perform symmetric encryption and decryption of files using the AES-256-CBC algorithm with PBKDF2 key derivation, ensuring data confidentiality and integrity. In the second part, OpenSSL is used to generate a private key and a self-signed X.509 certificate, which are then applied to configure SSL/TLS on a local web server (using either Python or Apache with XAMPP). The configured server is accessed via HTTPS, where the secure channel and certificate details can be verified in a browser. This practical exercise reinforces key concepts in cryptography, SSL/TLS, and secure web communication.

- **Encryption/Decryption:** Converts plaintext → ciphertext (encryption) and back (decryption) using keys/algorithms.
- **OpenSSL:** Toolkit for crypto, X.509 certificates, and SSL/TLS utilities.
- **SSL/TLS:** Protocols providing confidentiality, integrity, and authentication for web traffic.
- **Digital Certificate:** Binds a public key to an identity (domain), enabling trust in HTTPS.

## PROCEDURE:

**Pre-Lab Setup (Windows)**

1. **Create a working folder** (all commands are in PowerShell):
   mkdir C:\lab\exp7
   cd C:\lab\exp7
2. **Confirm OpenSSL is in PATH**:
   openssl version
   If not found, install OpenSSL for Windows and add its bin path (e.g., C:\Program Files\OpenSSL-Win64\bin) to **System PATH**, then reopen PowerShell.

**Part A — File Encryption/Decryption with OpenSSL (AES-256)**

We will use modern, safer key derivation with -pbkdf2 and a high iteration count.

1. **Create a plaintext file:**
   "This is a secret message" | Out-File -NoNewline -Encoding ascii secret.txt
   (You can also open it with Notepad and edit: notepad .\secret.txt)

```
PS C:\WINDOWS\system32> openssl version
>>
OpenSSL 3.5.2 5 Aug 2025 (Library: OpenSSL 3.5.2 5 Aug 2025)
PS C:\WINDOWS\system32> cd C:\lab\exp7
PS C:\lab\exp7> "This is a secret message" | Out-File -NoNewline -Encoding ascii secret.txt
```

2. **Encrypt the file (AES-256-CBC + PBKDF2):**
   openssl enc -aes-256-cbc -pbkdf2 -iter 100000 -salt -in secret.txt -out secret.enc
   - When prompted, **enter a strong passphrase** and confirm it.

```
PS C:\lab\exp7> openssl enc -aes-256-cbc -pbkdf2 -iter 100000 -salt -in secret.txt -out secret.enc
enter AES-256-CBC encryption password:

Verifying - enter AES-256-CBC encryption password:
```

3. **Decrypt back to plaintext:**
   openssl enc -d -aes-256-cbc -pbkdf2 -iter 100000 -in secret.enc -out secret_decrypted.txt
   - Use the **same passphrase** you set during encryption.

```
PS C:\lab\exp7> openssl enc -d -aes-256-cbc -pbkdf2 -iter 100000 -in secret.enc -out secret_decrypted.txt
>>
enter AES-256-CBC decryption password:
```

4. **Verify integrity (compare files):**
   cmd /c fc secret.txt secret_decrypted.txt
   - Expected: FC: no differences encountered

```
PS C:\lab\exp7> cmd /c fc secret.txt secret_decrypted.txt
>>
Comparing files secret.txt and SECRET_DECRYPTED.TXT
FC: no differences encountered
```

**Part B — Configure SSL/TLS on a Web Server**

**Step 1 — Generate Private Key & Self-Signed Certificate (with SAN)**
In C:\lab\exp7, run:
openssl req -x509 -newkey rsa:2048 -nodes -keyout key.pem -out cert.pem -days 365

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]:Telangana
Locality Name (eg, city) []:Hyderabad
Organization Name (eg, company) [Internet Widgits Pty Ltd]:CBIT
Organizational Unit Name (eg, section) []:CSE
Common Name (e.g. server FQDN or YOUR name) []:Sai Praveen
Email Address []:chinta.saipraveen14@gmail.com
PS C:\lab\exp7>
```

Explanation:

- req -x509 → generate an X.509 certificate.
- -newkey rsa:2048 → create a new RSA key (2048 bits).
- -nodes → don't encrypt the private key (so the server can use it without asking for password).
- -keyout key.pem → save private key here.
- -out cert.pem → save certificate here.
- -days 365 → valid for 1 year.

You should now have **key.pem** and **cert.pem** in C:\lab\exp7.

**Step 2A — Quick HTTPS Test with Python (recommended for demo)**

https_server.py > ...

```python
import http.server, ssl

httpd = http.server.HTTPServer(('localhost', 4443), http.server.SimpleHTTPRequestHandler)

context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain(certfile="cert.pem", keyfile="key.pem")
httpd.socket = context.wrap_socket(httpd.socket, server_side=True)

print("Serving on https://localhost:4443")
httpd.serve_forever()
```

Run it:
python .\https_server.py

**Step 2B — HTTPS with Apache (XAMPP)**

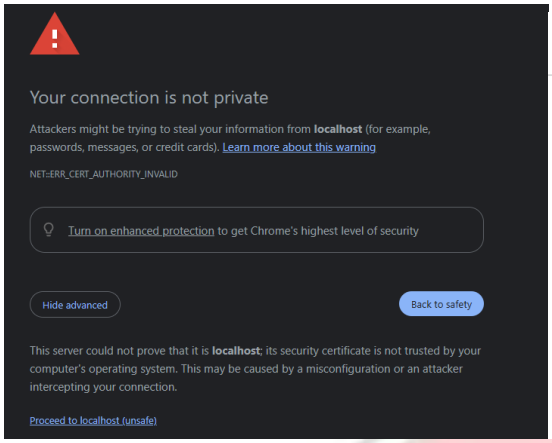
1. **Place files**:
   - Copy server.crt → C:\xampp\apache\conf\ssl.crt\server.crt
   - Copy server.key → C:\xampp\apache\conf\ssl.key\server.key (Create folders if they do not exist.)
2. **Enable SSL module & include SSL config:**
   - Open C:\xampp\apache\conf\httpd.conf in Notepad.
   - Ensure these lines are **uncommented** (no leading #):
     LoadModule ssl_module modules/mod_ssl.so
     Include conf/extra/httpd-ssl.conf
3. **Edit SSL virtual host file:**
   - Open C:\xampp\apache\conf\extra\httpd-ssl.conf.
   - Ensure a virtual host like below (adjust paths):

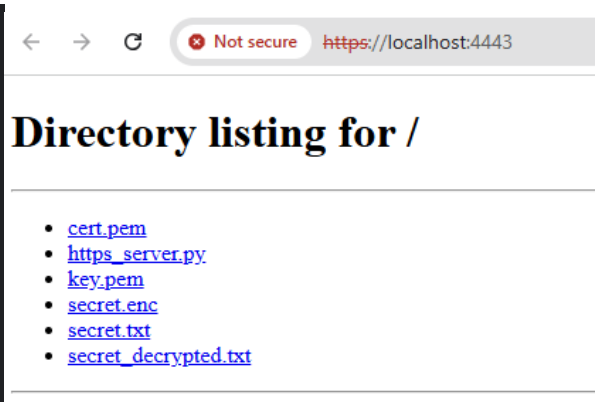
```
<VirtualHost _default_:443>
 DocumentRoot "C:/xampp/htdocs"
 ServerName localhost:443

 SSLEngine on
 SSLCertificateFile "C:/xampp/apache/conf/ssl.crt/server.crt"
 SSLCertificateKeyFile "C:/xampp/apache/conf/ssl.key/server.key"

 <Directory "C:/xampp/htdocs">
  Options Indexes FollowSymLinks
  AllowOverride All
  Require all granted
 </Directory>
</VirtualHost>
```

4. **Restart Apache** using **XAMPP Control Panel** → **Stop** then **Start** Apache.

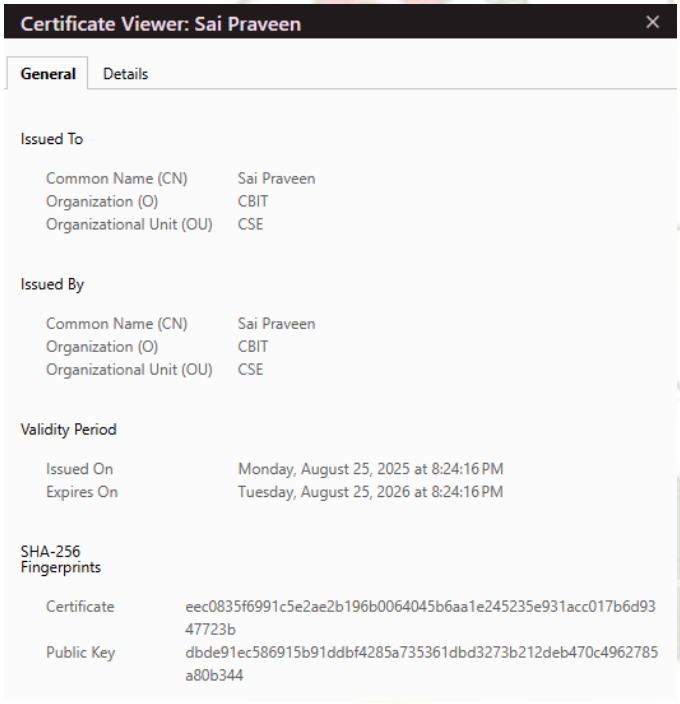
**Step 3 — Test in Browser**

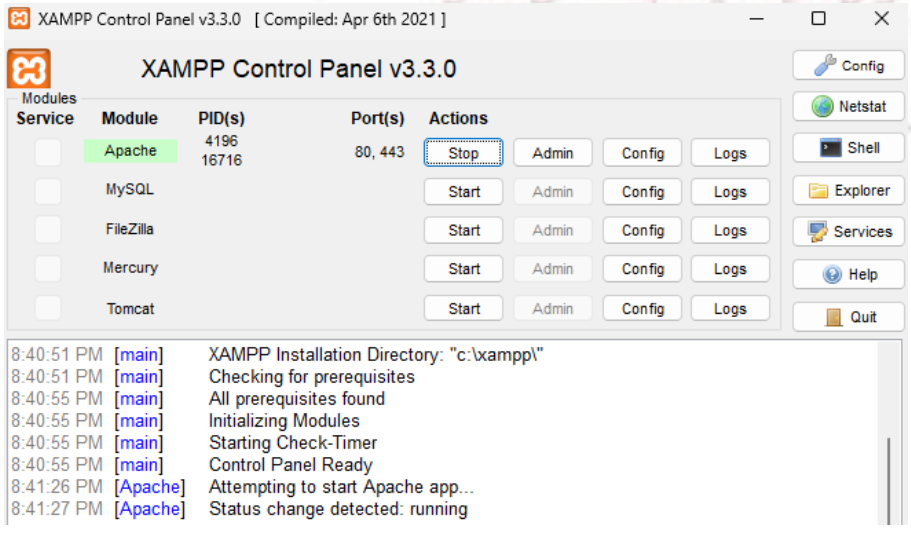
1. For Python server: open https://127.0.0.1:4443/ (or https://localhost:4443/).
2. For XAMPP Apache: open https://localhost/.
3. Because this is **self-signed**, the browser may show a warning:
   - Click **Advanced** → **Proceed to localhost** (for testing only).




4. Click the **lock icon → Connection is secure → Certificate** to view details (issuer, validity, SAN, etc.).



**Observation**

| Protocol | Observation in Browser | Security Status |
|---|---|---|
| HTTP (http://localhost) | Page loads without lock symbol | Not Secure |
| HTTPS (https://localhost) | Padlock or a warning (self-signed) | Secure (Encrypted) |

## ***OUTPUT ANALYSIS:***

- secret.enc is unreadable binary. Decrypting with the correct passphrase reproduces the original plaintext **exactly** (verified by fc or SHA256 hashes).
- The browser uses **TLS 1.2/1.3** to connect to the local server. Certificate details display **CN=localhost** and **SAN: DNS:localhost, IP:127.0.0.1**.
- Self-signed certificates are fine for **lab/testing** but **not for production**. In production, obtain a certificate from a trusted CA (e.g., Let's Encrypt).

## ***CONCLUSION:***

The experiment successfully demonstrated:

1. Symmetric file encryption and decryption using **OpenSSL (AES-256-CBC + PBKDF2)** on Windows.
2. Generating a private key and a **self-signed X.509 certificate** with SAN for localhost.
3. Hosting a site over **HTTPS** using **Python** or **Apache (XAMPP)** and verifying secure communication in the browser.

# EXPERIMENT – 09

## *AIM:*

Secure a database (MySQL/PostgreSQL) and test for SQL injection with SQLMap.

## *DESCRIPTION:*

This project focuses on securing a relational database (MySQL/PostgreSQL) against SQL injection attacks and validating the security using automated tools. An intentionally vulnerable database setup is first created with insecure queries to demonstrate successful SQL injection using SQLMap, such as data extraction and authentication bypass. Secure coding practices are then applied, including parameterized queries, least-privilege access, and input validation. The database is retested with SQLMap, which confirms that injection attempts fail, ensuring the system is resilient against common database exploitation techniques.

## *PROCEDURE:*

**Step 1: Install Database Server**

- Install and start a PostgreSQL service to prepare the environment.
- Confirm the database server is running and accessible.

**Step 2: Download SQLMap Software**

- Clone the SQLMap repository from GitHub.
- Add SQLMap to the system PATH so it can be executed using CLI.

**Step 3: Create a Sample Table and Insert Data**

- Create a test database and a users table with sample data.
- Verify the data insertion to ensure a valid target for SQLi testing.

**Step 4: Host a Test Web App Using Flask with a Vulnerable Query**

- Develop a simple web app that constructs SQL queries directly from user input (without sanitization).
- Verify the app returns user data correctly but is susceptible to SQLi.

**Step 5: Run SQLMap to Test for SQL Injection**

- Use SQLMap against the Flask application's endpoint to detect injectable parameters.

Enumerate the database and confirm successful data extraction, proving the vulnerability.

**Vulnerable Code:**

```python
# --- VULNERABLE ENDPOINT ---
@app.route('/user_vulnerable')
def get_user_vulnerable():
    """This endpoint is vulnerable to SQL injection."""
    user_id = request.args.get('id')

    if not user_id:
        return jsonify({"error": "Please provide an 'id' parameter."}), 400

    conn = get_db_connection()
    cur = conn.cursor()

    # VULNERABLE: Directly formatting user input into the query string.
    query = f"SELECT id, username, email FROM users WHERE id = {user_id}"

    try:
        cur.execute(query)
        user = cur.fetchone()
    except Exception as e:
        return jsonify({"error": str(e)}), 500
    finally:
        cur.close()
        conn.close()

    if user:
        return jsonify({"id": user[0], "username": user[1], "email": user[2]})
    else:
        return jsonify({"error": "User not found"}), 404
```

**Secure Code:**

```python
# --- SECURE ENDPOINT ---
@app.route('/user_secure')
def get_user_secure():
    """This endpoint is secure against SQL injection."""
    user_id = request.args.get('id')

    if not user_id:
        return jsonify({"error": "Please provide an 'id' parameter."}), 400

    conn = get_db_connection()
    cur = conn.cursor()

    # SECURE: Using a parameterized query to separate code from data.
    query = "SELECT id, username, email FROM users WHERE id = %s"

    try:
        # The user_id is passed as a separate tuple, handled safely by the driver.
        cur.execute(query, (user_id,))
        user = cur.fetchone()
    except Exception as e:
        return jsonify({"error": str(e)}), 500
    finally:
        cur.close()
        conn.close()

    if user:
        return jsonify({"id": user[0], "username": user[1], "email": user[2]})
    else:
        return jsonify({"error": "User not found"}), 404
```

**OUTPUT:**

Query statistics

Reset Statistics

| Query | Rows | Calls | Min (ms) | Mean (ms) | Max (ms) |
|---|---|---|---|---|---|
| SELECT pg_catalog.pg_current_wal_lsn… | 9 | 9 | 0.004714 | 0.008186 | 0.026197 |
| SELECT id, username, email FROM user… | 6 | 7 | 0.018219 | 0.039067 | 0.049465 |
| SELECT id, username, email FROM user… | 3 | 3 | 0.055788 | 3335.808594 | 5004.344727 |
| SELECT id, username, email FROM user… | 1 | 3 | 0.006186 | 0.017922 | 0.040794 |
| SELECT PG_SLEEP($1)-- | 3 | 3 | 0.019419 | 3335.552734 | 5005.164063 |
| SELECT id, username, email FROM user… | 3 | 2 | 0.039362 | 0.051177 | 0.062992 |
| INSERT INTO users (username, email) … | 2 | 2 | 0.071944 | 0.154886 | 0.237827 |
| SELECT id, username, email FROM user… | 2 | 2 | 0.039309 | 0.046932 | 0.054554 |
| INSERT INTO users (username, email) … | 2 | 2 | 0.397053 | 4.305604 | 8.214154 |
| SELECT id, username, email FROM user… | 2 | 2 | 0.034972 | 0.045796 | 0.056621 |

PostgreSQL query stats showing normal and SQL injection queries.

```
PS D:\TestCode> cd sqlmap-dev
PS D:\TestCode\sqlmap-dev> python sqlmap.py -u "http://127.0.0.1:5000/user_vulnerable?id=1" --dbs --batch
        ___
       __H__
 ___ ___[*]_____ ___ ___  {1.9.8.8#dev}
|_ -| . [*]     | .'| . |
|___|_  [*]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

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

[*] starting @ 10:26:55 /2025-08-25/

[10:26:56] [INFO] testing connection to the target URL
[10:26:58] [INFO] checking if the target is protected by some kind of WAF/IPS
[10:27:00] [INFO] testing if the target URL content is stable
[10:27:03] [INFO] target URL content is stable
[10:27:03] [INFO] testing if GET parameter 'id' is dynamic
[10:27:05] [WARNING] GET parameter 'id' does not appear to be dynamic
[10:27:07] [WARNING] heuristic (basic) test shows that GET parameter 'id' might not be injectable
[10:27:09] [INFO] testing for SQL injection on GET parameter 'id'
[10:27:09] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[10:27:28] [INFO] GET parameter 'id' appears to be 'AND boolean-based blind - WHERE or HAVING clause' injectable (with --code=200)
[10:28:17] [INFO] heuristic (extended) test shows that the back-end DBMS could be 'PostgreSQL'
it looks like the back-end DBMS is 'PostgreSQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] Y
for the remaining tests, do you want to include all tests for 'PostgreSQL' extending provided level (1) and risk (1) values? [Y/n] Y
[10:28:17] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'
[10:28:19] [INFO] GET parameter 'id' is 'PostgreSQL AND error-based - WHERE or HAVING clause' injectable
[10:28:19] [INFO] testing 'Generic inline queries'
[10:28:21] [INFO] testing 'PostgreSQL inline queries'
[10:28:23] [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'
[10:28:40] [INFO] GET parameter 'id' appears to be 'PostgreSQL > 8.1 stacked queries (comment)' injectable
[10:28:40] [INFO] testing 'PostgreSQL > 8.1 AND time-based blind'
[10:28:57] [INFO] GET parameter 'id' appears to be 'PostgreSQL > 8.1 AND time-based blind' injectable
```

SQLMap successfully detected SQLi in the vulnerable PostgreSQL web app parameter.

```
back-end DBMS: PostgreSQL
[10:29:50] [WARNING] schema names are going to be used on PostgreSQL for enumeration as the counterpart to database names on other DBMSes
[10:29:50] [INFO] fetching database (schema) names
[10:29:54] [WARNING] the SQL query provided does not return any output
[10:29:58] [INFO] retrieved: 'public'
[10:30:01] [INFO] retrieved: 'pg_catalog'
[10:30:03] [INFO] retrieved: 'information_schema'
```

PostgreSQL schemas: `public`, `pg_catalog`, `information_schema`.

## ***OUTPUT ANALYSIS & CONCLUSION:***

SQLMap testing revealed that the PostgreSQL database is vulnerable to SQL injection, as it successfully enumerated schemas like public, pg_catalog, and information_schema. This indicates improper input sanitization and weak query handling, allowing attackers to extract critical metadata. To mitigate such risks, secure coding practices such as parameterized queries, strict input validation, least-privilege access, and regular security testing must be enforced.

CHAITANYA BHARATHI
INSTITUTE OF TECHNOLOGY
స్వయం తేజస్వినే భవ
1979