

```

(kali@kali)-[~]
└─$ sudo apt-get install p0f
[sudo] password for kali:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
p0f is already the newest version (3.09b-4).
0 upgraded, 0 newly installed, 0 to remove and 1987 not upgraded.

(kali@kali)-[~]
└─$ p0f -L
p0f 3.09b by Michal Zalewski <lcantuf@coredump.cx>

-- Available interfaces --

0: Name      : eth0
   Description : -
   IP address : 192.168.44.130

1: Name      : any
   Description : Pseudo-device that captures on all interfaces
   IP address : (none)

2: Name      : lo
   Description : -
   IP address : 127.0.0.1

3: Name      : bluetooth-monitor
   Description : Bluetooth Linux Monitor
   IP address : (none)

```

The first screenshot demonstrates the installation and use of the p0f tool for passive fingerprinting. The `sudo apt-get install p0f` command confirms that p0f is already installed and up-to-date. Next, the `p0f -L` command is used to list all available network interfaces, showing `eth0` as the primary interface with an IP address of 192.168.44.130. Running `sudo p0f` starts the fingerprinting process, loading 322 signatures from its default configuration. However, no packets were processed during the session, possibly due to a lack of traffic on the interface. This step sets up the foundation for passively analyzing network packets and identifying OS configurations.

```

└─$ sudo p0f
p0f 3.09b by Michal Zalewski <lcantuf@coredump.cx>

[-] Closed 1 file descriptor.
[-] Loaded 322 signatures from '/etc/p0f/p0f.fp'.
[-] Intercepting traffic on default interface 'eth0'.
[-] Default packet filtering configured [v4v6].
[-] Entered main event loop.

^C[!] WARNING: User-initiated shutdown.
All done. Processed 0 packets.

(kali@kali)-[~]
└─$ sudo arp-scan 172.16.145.1/24
Interface: eth0, type: EN10MB, MAC: 00:0c:29:bd:78:9b, IPv4: 192.168.44.130
WARNING: Cannot open MAC/Vendor file ieee-oui.txt: Permission denied
WARNING: Cannot open MAC/Vendor file mac-vendor.txt: Permission denied
WARNING: host part of 172.16.145.1/24 is non-zero
Starting arp-scan 1.10.0 with 256 hosts (https://github.com/royhills/arp-scan)
0 packets received by filter, 0 packets dropped by kernel
Ending arp-scan 1.10.0: 256 hosts scanned in 2.266 seconds (112.97 hosts/sec). 0 responded

(kali@kali)-[~]
└─$ nmap -sn 172.16.145.1/24
Starting Nmap 7.80SN (https://nmap.org) at 2024-11-26 16:27 CST
Nmap done: 256 IP addresses (0 hosts up) scanned in 104.26 seconds

```

The second screenshot corresponds to Part II, where `arp-scan` and `nmap` are used to analyze the local network. The `sudo arp-scan` command attempts to map the network but encounters warnings about permissions (Cannot open MAC/Vendor file `ieee-oui.txt`) and an unreachable host. Similarly, the `nmap -sn` command performs a ping sweep across the 172.16.145.1/24 subnet, but no hosts respond, suggesting restricted connectivity or network misconfiguration. An additional `nmap -sU` UDP scan also shows Network unreachable errors. These steps reveal potential barriers in scanning and emphasize the importance of proper network and firewall configurations during forensic analysis.

The third screenshot aligns with the firewall configuration step in Part II, Step 9, where `ufw` (Uncomplicated Firewall) is configured to allow UDP port 53 traffic. Initially, the firewall is inactive, as confirmed by `sudo ufw status verbose`. Using `sudo ufw enable` activates it, and rules are applied to allow incoming UDP packets on port 53. The final status output verifies the firewall is active, logging is enabled, and the new rules are in place. This step is crucial for permitting specific traffic during port scans and preparing the environment for further investigation.

```

(kali@kali)-[~]
└─$ sudo systemctl start ufw

(kali@kali)-[~]
└─$ sudo ufw allow 53/udp
Rules updated
Rules updated (v6)

(kali@kali)-[~]
└─$ sudo ufw status verbose
Status: inactive

(kali@kali)-[~]
└─$ sudo ufw enable
Firewall is active and enabled on system startup

(kali@kali)-[~]
└─$ sudo ufw status verbose
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip

To Action From
--
53/udp ALLOW IN Anywhere
53/udp (v6) ALLOW IN Anywhere (v6)

```

The third screenshot aligns with the firewall configuration step in Part II, Step 9, where `ufw` (Uncomplicated Firewall) is configured to allow UDP port 53 traffic. Initially, the firewall is inactive, as confirmed by `sudo ufw status verbose`. Using `sudo ufw enable` activates it, and rules are applied to allow incoming UDP packets on port 53. The final status output verifies the firewall is active, logging is enabled, and the new rules are in place. This step is crucial for permitting specific traffic during port scans and preparing the environment for further investigation.

preparing the environment for further investigation.

```

(kali@kali) ~
$ sudo nmap -sU 172.16.145.137
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-11-26 16:52 CST
sendto in send_ip_packet.sd: sendto(4, packet, 68, 0, 172.16.145.137, 16) => Network is unreachable
Offending packet: UDP 192.168.44.138:52364 > 172.16.145.137:40198 ttl=61 id=27168 iplen=68
sendto in send_ip_packet.sd: sendto(4, packet, 68, 0, 172.16.145.137, 16) => Network is unreachable
Offending packet: UDP 192.168.44.138:52362 > 172.16.145.137:64481 ttl=53 id=31508 iplen=68
sendto in send_ip_packet.sd: sendto(4, packet, 68, 0, 172.16.145.137, 16) => Network is unreachable
Offending packet: UDP 192.168.44.138:52364 > 172.16.145.137:40198 ttl=61 id=27168 iplen=68
sendto in send_ip_packet.sd: sendto(4, packet, 28, 0, 172.16.145.137, 16) => Network is unreachable
Offending packet: UDP 192.168.44.138:52362 > 172.16.145.137:6050 ttl=50 id=25110 iplen=28
sendto in send_ip_packet.sd: sendto(4, packet, 28, 0, 172.16.145.137, 16) => Network is unreachable
Offending packet: UDP 192.168.44.138:52364 > 172.16.145.137:1835 ttl=46 id=52736 iplen=100
sendto in send_ip_packet.sd: sendto(4, packet, 100, 0, 172.16.145.137, 16) => Network is unreachable
Offending packet: UDP 192.168.44.138:52362 > 172.16.145.137:1035 ttl=42 id=12856 iplen=100
sendto in send_ip_packet.sd: sendto(4, packet, 100, 0, 172.16.145.137, 16) => Network is unreachable
Offending packet: UDP 192.168.44.138:52364 > 172.16.145.137:1835 ttl=46 id=52736 iplen=100
sendto in send_ip_packet.sd: sendto(4, packet, 28, 0, 172.16.145.137, 16) => Network is unreachable
Offending packet: UDP 192.168.44.138:52364 > 172.16.145.137:16974 ttl=48 id=32915 iplen=28
sendto in send_ip_packet.sd: sendto(4, packet, 28, 0, 172.16.145.137, 16) => Network is unreachable
Offending packet: UDP 192.168.44.138:52362 > 172.16.145.137:20 ttl=57 id=6562 iplen=28
sendto in send_ip_packet.sd: sendto(4, packet, 28, 0, 172.16.145.137, 16) => Network is unreachable
Offending packet: UDP 192.168.44.138:52364 > 172.16.145.137:20 ttl=57 id=6562 iplen=28
Omitting future Sendto error messages now that 10 have been shown. Use -d2 if you really want to see them.
Nmap scan report for 172.16.145.137
Host is up (0.001s latency).
All 1000 scanned ports on 172.16.145.137 are in ignored states.
Not shown: 1000 open/filtered udp ports (no-response)

```

The fourth screenshot shows the results of a UDP port scan using nmap, aligned with Part II, Step 9. After configuring the firewall, nmap -sU is run against the target host (172.16.145.137). Despite allowing UDP port 53 in the firewall, the scan results still indicate Network unreachable. This could be due to other network-level restrictions or incorrect configurations on the

target host. The result highlights challenges in network exploration and the importance of verifying connectivity and target readiness.

```

Cloning into 'swap_digger'...
remote: Enumerating objects: 147, done.
remote: Counting objects: 100% (30/30), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 147 (delta 15), reused 21 (delta 11), pack-reused 117 (from 1)
Receiving objects: 100% (147/147), 357.52 KiB | 1.02 MiB/s, done.
Resolving deltas: 100% (69/69), done.

```

```

(kali@kali) ~/Downloads
$ cd swap_digger

(kali@kali) ~/Downloads/swap_digger
$ sudo chmod +x swap_digger.sh
[sudo] password for kali:

(kali@kali) ~/Downloads/swap_digger
$ sudo ./swap_digger.sh -S

- SWAP Digger -

[+] Current swap file:
  -> /dev/sda5
[+] /etc/fstab swap files:
  -> /dev/sda5
[+] Looking for all available swap device files (will take some time):
  -> /dev/sda5

SWAP Digger end, byebye!

/home/kali/Downloads/swap_digger

```

These screenshots showcase the installation and execution of swap_digger, as part of Part III. In the beginning, the git clone command fails due to outdated authentication methods, which require personal access tokens instead of passwords. After resolving this, the repository is successfully cloned, the script is made executable using

chmod +x, and it is run with ./swap_digger.sh -S. The output confirms the active swap file as /dev/sda5, providing insight into potential data remnants stored in swap space. This step is critical for recovering sensitive information like credentials, session data, and other forensic artifacts stored temporarily in memory.

sudo ./swap_digger.sh -a: The command is used within the context of analyzing and extracting data from the system's swap space, which contains sensitive remnants of information temporarily written from memory. By running the script, the swap_digger tool gains access to the system-level swap file for forensic analysis. The -a option performs a sweep of the swap

partition, removing various types of potentially valuable artifacts, including web-entered passwords, emails, XML-based credentials, and structured data. It also searches for WiFi access points, frequently accessed HTTP/HTTPS URLs, FTP domains, opened files, and traces of IP addresses, though some results, such as IP addresses, may include false positives. The script also attempts to locate cryptographic hashes like MD5, SHA1, and SHA256. This analysis provides forensic investigators with insights into the system's activity, recovering data that could include usernames, passwords, session details, or activity traces left in swap memory. The output, such as human-readable strings and artifact details, is typically saved in a file (e.g., /tmp/swap_dig/swap_dump.txt) for further analysis. This functionality is critical in digital forensic investigations for reconstructing user activity, recovering lost credentials, or identifying potential security breaches and misuse of the system, making the -a option an essential part of the artifact recovery process.

In Step 14, the command rkhunter is used to scan the Linux system for rootkits, hidden files, and suspicious configurations. After installing the tool using `sudo apt-get install rkhunter`, the `sudo rkhunter --check -rwo` command initiates the analysis. This command checks for files or scripts that deviate from known safe versions stored in the rkhunter database. In Step 15, the command chkrootkit serves as another layer of defense for rootkit detection. After installing it with `sudo apt-get install chkrootkit`, running `sudo chkrootkit` scans for known rootkit signatures. It examines system binaries and critical configurations, verifying whether they are infected or intact. Each scanned element, like cron or date, is checked against a list of typical rootkit alterations, and the tool provides clear results on whether these elements are safe or compromised. The `ascii` command in step 16 is useful for displaying ASCII character information in various formats. The command `ascii -s hello` converts the string "hello" into its corresponding ASCII values, displaying each character's decimal, hexadecimal, and binary representations. When running `ascii -x`, the command outputs an extended ASCII table, listing all characters alongside their respective codes. In step 17 we use the `xxd` command to generate hexadecimal dumps of files, facilitating the identification of file signatures. For instance, `xxd -g 1 0zapftis.rar | head` outputs the first ten lines of the hexadecimal representation of a .rar file, including its signature 52 61 72, which corresponds to the Roshal Archive format. In Step 18, the `strings` command is utilized to extract human-readable text from binary files. By combining it with `grep`, we can search for specific patterns, such as jpg, within a file. For example, `strings -t x terry-work-usb-2009-12-11.E01 | grep -i jpg` reveals occurrences of jpg in the binary file along with their offsets, which are displayed in hexadecimal. In Step 19, the Sleuth Kit, a suite of tools for examining disk images and recovering files, is introduced. The `fsstat` command provides

detailed filesystem info, such as type, volume info, and cluster sizes. For example, on a FAT16 image, it shows sector size and FAT structure, helping analysts understand data organization. The `fls` command lists files and directories in a disk image, showing names and metadata, which is crucial for identifying valuable or deleted files. Similarly, the `ils` command lists inode info, including creation and modification timestamps, vital for forensic timelines. The `img_stat` command examines the disk image structure, showing type and size, ensuring image integrity for analysis, while `img_cat` extracts raw data from disk images for further investigation. In Step 20, the `fiwalk` command is for analyzing filesystems within disk images, providing detailed file info, including metadata and hash values, which helps verify file integrity and identify tampering.