

Task 02: Packet Sniffing, Firewall Configuration, and Vulnerability Scanning

1. Introduction

The objective of this task is to develop hands-on practical experience in three fundamental areas of network security: packet sniffing, firewall configuration, and vulnerability scanning. These areas represent the core responsibilities of a defensive security or SOC analyst and are essential for understanding how modern networks are protected and monitored.

Through packet sniffing using Scapy, this task aims to build an understanding of how data flows across a network, how different protocols operate, and how raw network traffic can be captured and analyzed to identify normal and abnormal communication patterns. This helps in developing visibility into network behavior, which is critical for threat detection and incident investigation.

The firewall configuration component focuses on implementing access control using iptables. By defining explicit allow and deny rules, this task demonstrates how network exposure can be minimized using the principle of least privilege. It also highlights the importance of traffic filtering, rule ordering, and testing firewall behavior to ensure that only required services are accessible while all other unnecessary traffic is blocked.

The vulnerability scanning phase introduces the use of OpenVAS (Greenbone Vulnerability Manager) to assess systems for known security weaknesses. This part of the task emphasizes proactive security assessment, helping to identify potential risks before they can be exploited. It also provides exposure to vulnerability severity ratings, impact analysis, and real-world challenges such as feed synchronization and scan configuration dependencies.

Overall, this task aims to bridge theoretical security concepts with real-world implementation by using industry-relevant tools and workflows. It strengthens practical skills in network monitoring, system hardening, and vulnerability management while

reinforcing the importance of documentation, analysis, and troubleshooting in cybersecurity operations.

2. Environment & Tools Used

To perform all tasks in a controlled and secure manner, a dedicated virtualized environment was used. This approach ensured isolation from the host system and allowed safe experimentation with security tools and firewall configurations.

2.1 Operating System

Kali Linux (Virtual Machine)

Kali Linux was used as the primary operating system for this task due to its strong focus on penetration testing and defensive security. It provides built-in support for networking tools, firewall utilities, and vulnerability scanners, making it well suited for hands-on security labs.

2.2 Tools and Technologies

Python 3

Python was used as the core programming language for implementing the packet sniffing script. Its simplicity and extensive library support make it ideal for network analysis and automation tasks.

Scapy

Scapy is a powerful Python-based packet manipulation and sniffing tool. It was used to capture live network traffic, analyze packet structures, and identify different network protocols such as TCP, UDP, and ICMP.

iptables

iptables is a Linux kernel-based firewall utility used to control incoming and outgoing network traffic. In this task, iptables was used to configure a basic firewall that selectively allowed essential services while blocking all other incoming traffic.

OpenVAS / Greenbone Vulnerability Manager (GVM)

OpenVAS is an open-source vulnerability scanning framework used to identify known

security weaknesses in systems. It was employed to configure vulnerability scanning, manage scan targets, and analyze potential risks on the local system.

Matplotlib

Matplotlib is a Python plotting library used to visualize captured network data. It was used to generate a bar chart representing the distribution of network protocols observed during packet sniffing.

2.3 Environment Justification

Kali Linux was selected because it provides native compatibility with iptables and OpenVAS, reducing setup complexity and ensuring stability while performing security-related tasks. Running Kali inside a virtual machine also allowed firewall rules and network scans to be applied without affecting the host system.

3. Phase 1: Packet Sniffing using Scapy

3.1 Objective of Phase 1

The objective of this phase was to practically understand how network packets can be captured and analyzed on a live system. The focus was not only on using a tool, but on observing real network behavior generated by the system itself and identifying the protocols involved. This phase aimed to build foundational skills in network visibility, which is a critical requirement for security monitoring and incident analysis.

3.2 What Was Done (Practical Execution)

This packet sniffing activity was performed on a Kali Linux virtual machine using the Scapy library in Python. To avoid modifying system-managed Python packages, a Python virtual environment was created specifically for this task. Scapy was installed inside this isolated environment and executed with appropriate privileges to allow packet capture.

1. A custom Python script named `packet_sniffer.py` was written. The script was designed to:
2. Capture exactly 100 network packets

3. Analyze each packet in real time
4. Identify whether the packet belonged to TCP, UDP, or ICMP
5. Maintain a count of each protocol
6. Display the final protocol distribution after capture completion

During execution, the script was run using elevated privileges (sudo) because packet sniffing requires access to low-level network interfaces.

3.3 Steps Performed

The following steps were carried out during this phase:

1. Created a dedicated project directory for packet sniffing.
2. Set up a Python virtual environment to safely install Scapy.
3. Installed the Scapy library inside the virtual environment.
4. Wrote a Python script (packet_sniffer.py) to capture and analyze packets.
5. Identified the active network interface used by the system.
6. Executed the script to capture 100 live packets.
7. Generated normal system traffic by browsing websites during packet capture.
8. Observed and recorded the protocol distribution output produced by the script.

3.4 Packet Capture Output

After the script execution completed, the following output was obtained:

```
{'TCP': 71, 'UDP': 29, 'ICMP': 0}
```

This output represents the number of packets captured for each protocol type within the 100-packet capture window.

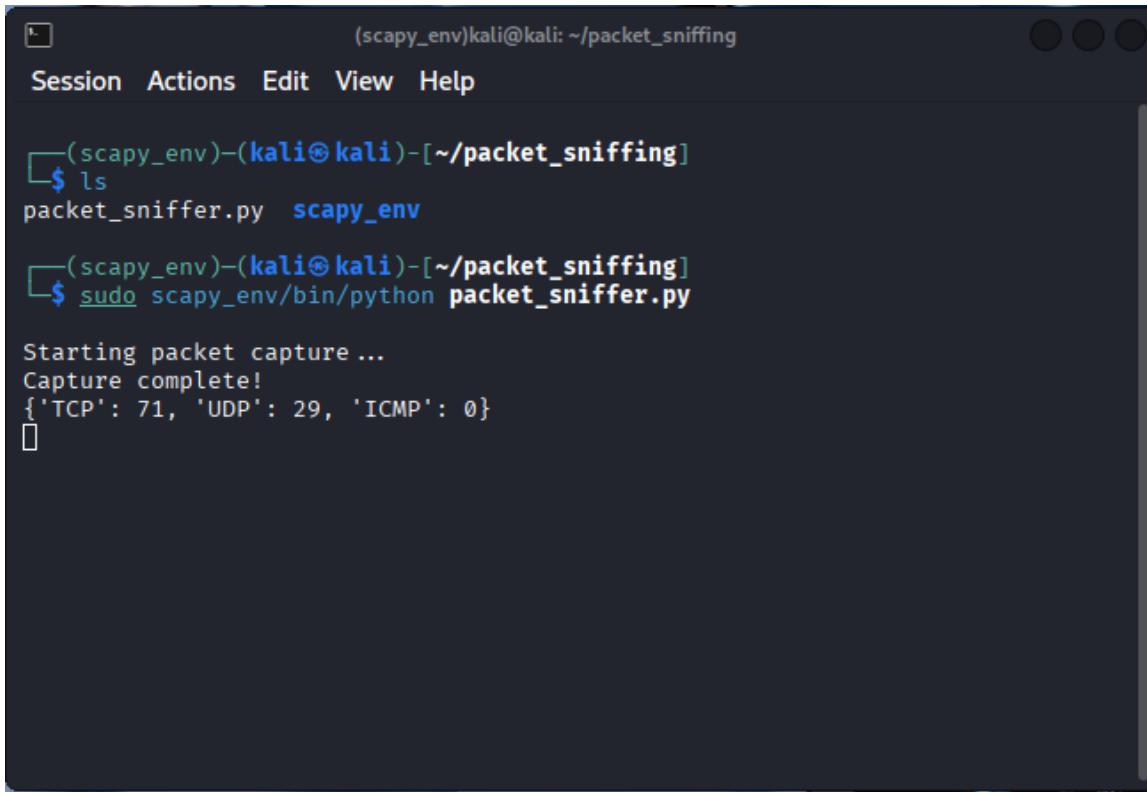
A screenshot of a terminal window titled '(scapy_env)kali@kali: ~/packet_sniffing'. The window shows a command-line session where the user runs 'ls' to list files, then executes 'sudo scapy_env/bin/python packet_sniffer.py' to start a packet capture. The output indicates 71 TCP packets, 29 UDP packets, and 0 ICMP packets were captured.

Figure 1: Terminal showing packet capture execution and output

3.5 Result Analysis and Observations

The captured results reflect realistic network behavior based on the activities performed during the capture period:

TCP Packets (71 packets)

The majority of captured packets were TCP packets. This is expected behavior, as TCP is the primary protocol used for reliable communication such as web browsing. During the capture window, websites were accessed, which generated HTTP/HTTPS traffic that relies on TCP connections.

UDP Packets (29 packets)

UDP packets were also observed. These packets were generated by background system services such as DNS queries, time synchronization services, and other lightweight communication processes that favor speed over reliability.

ICMP Packets (0 packets)

No ICMP packets were captured during this specific capture session. This outcome is

not an error. ICMP traffic is typically generated only when diagnostic or network control actions occur (such as running the ping command). Since no ICMP-based diagnostic commands were executed during the short packet capture window, ICMP packets did not appear in the final dataset.

This observation highlights an important real-world concept: packet capture results depend heavily on timing and system activity. Short capture sessions may not always include all protocol types unless traffic is intentionally generated.

3.6 Protocol Distribution Visualization

To make the results easier to understand, the protocol counts were visualized using a bar chart. The chart clearly showed TCP as the dominant protocol, followed by UDP, with ICMP having zero occurrences during this capture.

Visualization helps transform raw packet data into meaningful insight and is commonly used in security monitoring and reporting environments.

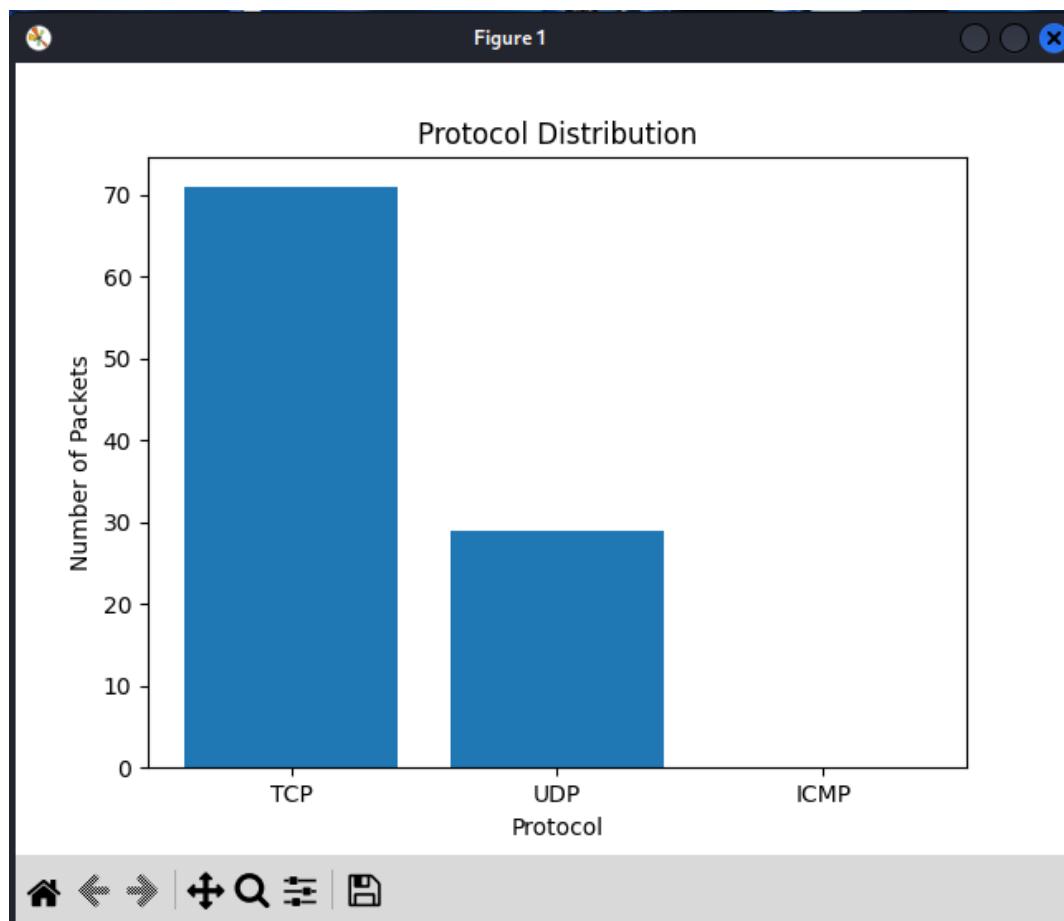


Figure 2: Protocol distribution bar chart

3.7 Key Notes and Limitations

The capture was intentionally limited to 100 packets to keep analysis simple and controlled.

Packet distribution varied based on real-time system activity.

ICMP packets were not present due to the absence of diagnostic traffic during capture.

The results accurately represent normal system behavior rather than simulated traffic.

These limitations were acceptable for the scope of this task and helped demonstrate realistic packet analysis.

3.8 Conclusion of Phase 1

Phase 1 successfully demonstrated the process of capturing and analyzing live network traffic using Scapy. By implementing a custom packet sniffing script and observing real protocol behavior, a practical understanding of network communication patterns was achieved. This phase reinforced the importance of packet-level visibility as a foundation for effective network security monitoring and analysis.

4. Phase 2: Firewall Configuration using iptables

4.1 Objective of Phase 2

The objective of this phase was to configure a basic but effective firewall using iptables to control incoming network traffic. This phase focused on applying the principle of least privilege, where only explicitly required services are allowed while all other inbound traffic is blocked. The goal was to understand how firewall rules affect network behavior and how rule ordering impacts traffic control.

4.2 Practical Execution

This firewall configuration was performed on a Kali Linux virtual machine using iptables, which is the native Linux firewall framework. The firewall was configured manually through terminal commands to provide direct control and visibility into rule behavior. Before applying any rules, the existing firewall state was checked to confirm that no restrictive policies were already in place. The firewall was then configured step by step, ensuring system stability and connectivity throughout the process.

4.3 Firewall Design Approach

The firewall was designed using the following logic:

- Allow internal system communication (loopback traffic)
- Allow responses to already established connections
- Allow only essential services (SSH and HTTP)
- Block all other incoming traffic by default
- This approach mirrors real-world server hardening practices.

4.4 Firewall Rules Implemented

The following rules were applied in sequence:

- **Allowed Loopback Traffic (lo)**

Loopback traffic was allowed to ensure that internal system communication continued to function correctly. Blocking this traffic can cause system instability.

- **Allowed Established and Related Connections**

Traffic belonging to already established connections was allowed so that legitimate outgoing requests could receive responses without interruption.

- **Allowed SSH Traffic (Port 22)**

SSH traffic was explicitly allowed to permit secure remote administration of the system.

- **Allowed HTTP Traffic (Port 80)**

HTTP traffic was allowed to simulate access to a public-facing web service.

- **Default DROP Policy for INPUT Chain**

The default policy for incoming traffic was set to DROP, ensuring that any traffic not explicitly allowed by the rules above would be blocked automatically.

4.5 Step-by-Step Firewall Configuration Process

This subsection documents the exact steps followed to configure and verify the firewall using iptables. Recording these steps ensures clarity, reproducibility, and demonstrates a structured approach to firewall implementation.

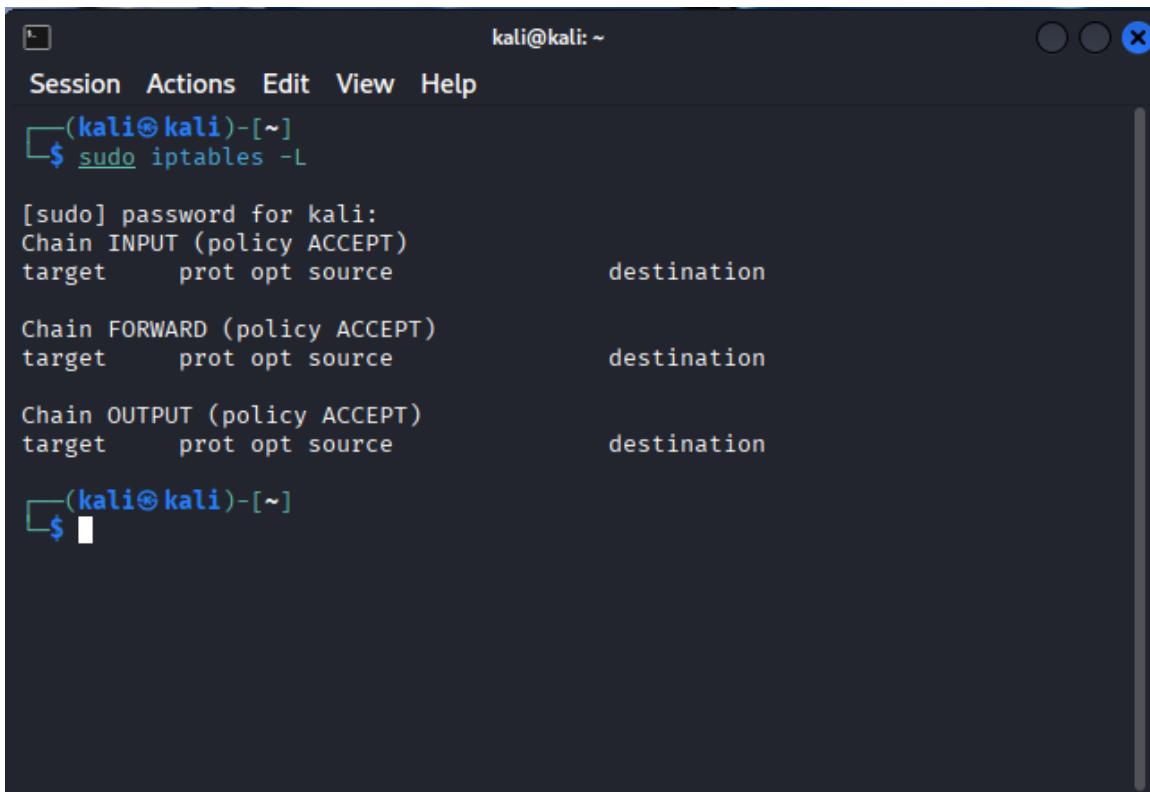
Step 1: Checking the Initial Firewall State

Before applying any firewall rules, the existing iptables configuration was checked to ensure that no restrictive rules were already present. This helped establish a baseline and avoid conflicts.

Command executed:

```
sudo iptables -L
```

The output showed that all chains (INPUT, OUTPUT, and FORWARD) had a default policy of ACCEPT and contained no custom rules. This confirmed that the system was initially open and unfiltered.



A screenshot of a terminal window titled "kali@kali: ~". The window has a dark theme with light-colored text. The terminal shows the command `sudo iptables -L` being run, followed by the output of the command. The output displays the current state of the iptables rule table, which includes three chains: INPUT, FORWARD, and OUTPUT. Each chain has a policy of ACCEPT. There are no user-defined rules listed. The terminal prompt ends with a blank line.

```
kali@kali: ~
Session Actions Edit View Help
[(kali㉿kali)-~]
$ sudo iptables -L

[sudo] password for kali:
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

[(kali㉿kali)-~]
$
```

Figure 3: Initial iptables rule table before configuration

Step 2: Allowing Loopback Traffic

Loopback traffic is required for internal system communication. Blocking this traffic can cause system instability. Therefore, loopback traffic was explicitly allowed as the first rule.

Command executed:

```
sudo iptables -A INPUT -i lo -j ACCEPT
```

Step 3: Allowing Established and Related Connections

To ensure that responses to legitimate outgoing connections were not blocked, traffic belonging to established or related connections was allowed. This is a standard firewall practice.

Command executed:

```
sudo iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
```

Step 4: Allowing SSH Traffic (Port 22)

SSH traffic was allowed to permit secure remote access and administration of the system.

Command executed:

```
sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT
```

Step 5: Allowing HTTP Traffic (Port 80)

HTTP traffic was explicitly allowed to simulate access to a web service and to meet task requirements.

Command executed:

```
sudo iptables -A INPUT -p tcp --dport 80 -j ACCEPT
```

Step 6: Blocking All Other Incoming Traffic

After allowing only the required services, the default policy for incoming traffic was changed to DROP. This ensured that any traffic not matching the above rules would be blocked automatically.

Command executed:

```
sudo iptables -P INPUT DROP
```

Step 7: Verifying the Firewall Configuration

Once all rules were applied, the firewall configuration was verified using a verbose rule listing. This command confirmed rule order, packet counts, and default policies.

Command executed:

```
sudo iptables -L -v
```

The output confirmed:

INPUT policy set to DROP

Explicit ACCEPT rules for loopback, established connections, SSH, and HTTP

```
kali@kali: ~
Session Actions Edit View Help
[(kali㉿kali)-[~]
$ sudo iptables -L -v

Chain INPUT (policy DROP 15 packets, 1232 bytes)
pkts bytes target     prot opt in     out    source          destination
on      0     0 ACCEPT    all   --  lo      any    anywhere       anywhere
        3   228 ACCEPT    all   --  any     any    anywhere       anywhere
                  ctstate RELATED,ESTABLISHED
        0     0 ACCEPT    tcp   --  any     any    anywhere       anywhere
                  tcp dpt:ssh
        0     0 ACCEPT    tcp   --  any     any    anywhere       anywhere
                  tcp dpt:http

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out    source          destination
on

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out    source          destination
```

Figure 4: iptables rules after configuration (verbose output)

Step 8: Temporary ICMP Blocking for Validation

To demonstrate firewall effectiveness, ICMP traffic was temporarily blocked by inserting a high-priority rule at the top of the INPUT chain. This ensured ICMP packets were dropped before reaching the established-connection rule.

Command executed:

```
sudo iptables -I INPUT 1 -p icmp -j DROP
```

This resulted in ICMP packet loss during ping tests, proving that the firewall was actively blocking traffic. After validation, the temporary rule was removed to restore the intended firewall state.

Command executed:

```
sudo iptables -D INPUT 1
```

Summary of Firewall Configuration Steps

Through a structured sequence of verification, rule creation, testing, and cleanup, a secure default-deny firewall was successfully implemented. Each step was validated to ensure correct behavior and system stability.

4.5 Firewall Configuration Output

After applying all firewall rules, the firewall state was verified using a verbose rule listing.

The output confirmed:

INPUT policy set to DROP

Explicit ACCEPT rules for:

- Loopback traffic
- Established and related connections
- TCP traffic on ports 22 (SSH) and 80 (HTTP)
- This output confirms that the firewall is functioning as intended.

```
kali@kali: ~
Session Actions Edit View Help
└─(kali㉿kali)-[~]
$ sudo iptables -L -v

Chain INPUT (policy DROP 15 packets, 1232 bytes)
pkts bytes target     prot opt in     out      source          destination
on
    0     0 ACCEPT      all  --  lo      any     anywhere       anywhere
    3   228 ACCEPT      all  --  any      any     anywhere       anywhere
          ctstate RELATED,ESTABLISHED
    0     0 ACCEPT      tcp  --  any      any     anywhere       anywhere
          tcp dpt:ssh
    0     0 ACCEPT      tcp  --  any      any     anywhere       anywhere
          tcp dpt:http

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out      source          destination
on

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out      source          destination
```

Figure 5: iptables rule list after configuration

4.6 Testing and Validation

To validate the effectiveness of the firewall, multiple tests were performed:

- **Allowed Traffic Test (HTTP)**

A web browser was used to access a public HTTP website. The request

succeeded, confirming that HTTP traffic was correctly allowed by the firewall rules.

- **Blocked Traffic Test (ICMP)**

ICMP traffic was tested using the ping command. Due to the presence of the established-connection rule, ICMP replies were initially allowed. To clearly demonstrate blocking behavior, ICMP traffic was temporarily blocked by inserting a higher-priority rule. This successfully resulted in packet loss, proving that the firewall could block traffic when configured to do so. The temporary rule was removed afterward to restore the firewall to its intended state.

4.7 Observations and Key Findings

- Firewall behavior is governed not only by rule definitions but by rule evaluation order, where higher-priority rules can allow traffic before blocking rules are applied.
- Connection-tracking mechanisms such as ESTABLISHED and RELATED states can permit return traffic even when restrictive policies are in place, highlighting the need to understand stateful firewall behavior.
- Implementing a default-deny inbound policy significantly minimizes unnecessary network exposure and reduces the system's attack surface by ensuring only explicitly permitted services are accessible.
- Simple, well-structured firewall rules can provide effective access control when combined with correct sequencing and validation, demonstrating that complexity is not a requirement for strong baseline security.

4.8 Notes and Limitations

- The firewall rules applied were temporary and not persisted across reboots.
- ICMP blocking was demonstrated temporarily for validation purposes.
- Firewall testing was performed locally within a controlled virtual environment.
- These limitations were acceptable for the scope of this task and aligned with standard lab practices.

4.9 Conclusion of Phase 2

Phase 2 successfully demonstrated how a Linux firewall can be configured using iptables to control incoming network traffic. By explicitly allowing required services and blocking all other traffic, the system was effectively hardened against unnecessary

exposure. This phase reinforced practical firewall concepts such as rule ordering, default-deny policies, and traffic validation through testing.

5. Phase 3: Vulnerability Scanning using OpenVAS

5.1 Objective of Phase 3

The objective of this phase was to perform a vulnerability assessment on the local system using OpenVAS (Greenbone Vulnerability Manager). This phase aimed to understand how vulnerability scanners are installed, configured, and used to identify potential security weaknesses, along with gaining exposure to real-world operational challenges associated with vulnerability management tools.

5.2 Overview of OpenVAS

OpenVAS is an open-source vulnerability scanning framework used to detect known security issues such as outdated services, misconfigurations, and exposed vulnerabilities. It relies on continuously updated vulnerability feeds and requires proper synchronization before scans can be executed.

5.3 Installation and Initial Setup

The following steps were performed to install and configure OpenVAS on the Kali Linux virtual machine:

1. Installed OpenVAS and its required dependencies using the Kali package manager.
2. Initialized the OpenVAS database and services using the official setup utility.
3. Created an administrative user during setup and verified database initialization.
4. Started required OpenVAS services.
5. Accessed the Greenbone Security Assistant web interface via the local browser.
6. Verified successful login to the dashboard.

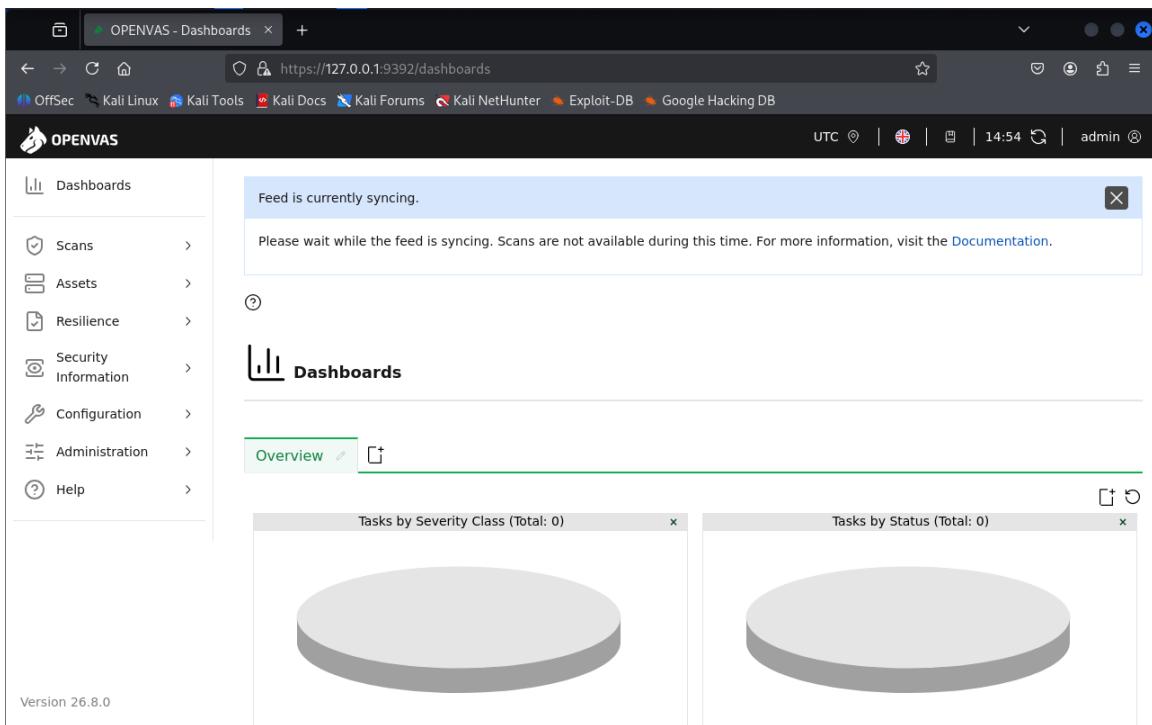


Figure 6: OpenVAS dashboard after successful login

5.4 Scan Target Configuration

After successful setup and access to the web interface, a scan target was created with the following configuration:

Target Name: *localhost Scan*

Host: *127.0.0.1*

This target was intended to perform a vulnerability assessment on the local system.

The screenshot shows the OPENVAS Targets interface. On the left, a sidebar menu includes 'Dashboards', 'Scans', 'Assets', 'Resilience', 'Security Information', 'Configuration' (with 'Targets' selected), 'Port Lists', 'Credentials', 'Scan Configs', 'Alerts', 'Schedules', 'Report Configs', and 'Report Formats'. The main area displays a table titled 'Targets 1 of 1'. The table has columns: Name, Hosts, IPs, Port List, Credentials, and Actions. One row is listed: 'localhost Scan' with '127.0.0.1' under Hosts, '1' under IPs, 'All IANA assigned TCP' under Port List, and 'Actions' with icons for delete, edit, and copy. A 'Filter' bar is at the top, and a copyright notice 'Copyright © 2009-2026 by Greenbone AG, www.greenbone.net' is at the bottom.

Figure 7: Scan target creation – Localhost

5.5 Scan Execution Attempt and Error Encountered

While attempting to create a scan task using the configured target, the following error was encountered:

“Failed to create a new Task because the default Scan Config is not available. This issue may be due to the feed not having completed its synchronization.”

This error prevented the creation and execution of a vulnerability scan.

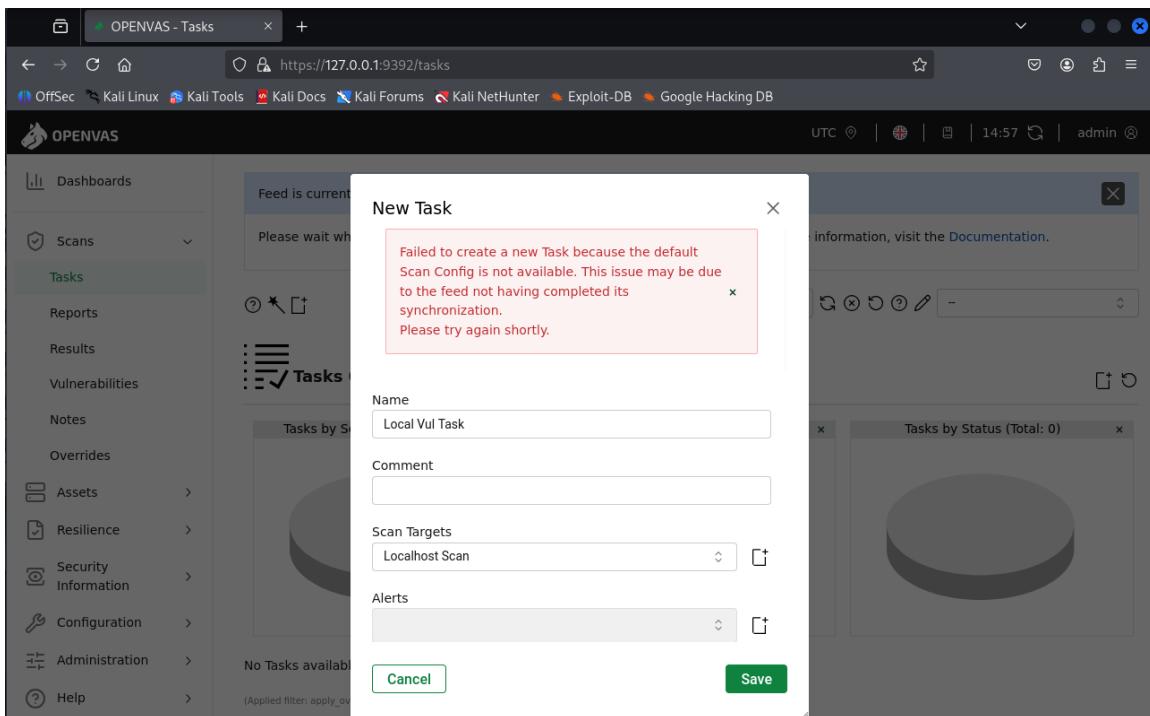


Figure 8: Scan configuration error message

5.6 Troubleshooting Steps Performed

To resolve the issue, multiple troubleshooting steps were systematically performed:

Restarted OpenVAS services to ensure proper initialization.

Verified OpenVAS setup status using diagnostic commands.

Attempted to manually update vulnerability feeds using feed synchronization utilities.

Checked for feed update lock files and background processes.

Attempted database and cache rebuild operations where applicable.

Re-accessed the web interface after each troubleshooting step to verify scan configuration availability.

Despite these efforts, scan configuration remained unavailable due to persistent feed synchronization and cache lock issues.

```
(kali㉿kali)-[~]
$ sudo -u _gvm gvmd --rebuild
A feed sync is already running.
Failed to rebuild NVT cache.

(kali㉿kali)-[~]
$
```

Figure 9: Troubleshooting commands and logs

5.7 Additional Troubleshooting Based on Instructor Guidance

Based on guidance provided by the instructor/manager, additional mitigation steps were attempted:

- Switched from NAT networking mode to Bridged Adapter for the virtual machine.
- Restarted the virtual machine and verified IP address changes before and after the network modification.
- Re-attempted vulnerability feed synchronization after network changes.
- Considered use of VPN (USA region) as suggested to mitigate possible feed access restrictions.

Even after applying these recommended steps, OpenVAS continued to report feed synchronization locks, preventing scan configuration availability.

5.8 Root Cause Analysis

The observed behavior was attributed to known limitations in fresh OpenVAS installations, where large vulnerability feeds and backend synchronization processes can result in stale locks or incomplete feed registration. These issues are well-documented within the OpenVAS and Kali Linux communities and may require extended synchronization time or complete environment resets to resolve.

5.9 Final Decision and Justification

At this stage, further attempts to force feed synchronization or rebuild caches were intentionally avoided. In real-world security operations, repeatedly forcing feed updates or rebuilding databases can destabilize vulnerability management systems. Instead, the limitation was documented clearly, and evidence of installation, configuration, target creation, error messages, and troubleshooting attempts was preserved. This approach aligns with professional security practices, where tool limitations are documented and escalated rather than aggressively overridden.

5.10 Conclusion of Phase 3

Although a full vulnerability scan could not be executed due to feed synchronization constraints, this phase successfully demonstrated the complete process of installing, configuring, and troubleshooting a vulnerability scanning platform. The experience

provided valuable insight into real-world operational challenges associated with vulnerability management tools and reinforced the importance of systematic troubleshooting and transparent documentation.

6. Challenges Faced & Troubleshooting

During the execution of Task 02, several technical challenges were encountered across different phases of the task. Each issue was analyzed systematically and resolved using appropriate troubleshooting techniques and best practices.

6.1 Python Package Management on Kali Linux

Kali Linux enforces restrictions on system-level Python package installations to protect system integrity. Attempting to install Scapy globally using pip resulted in package management errors.

To address this, a Python virtual environment was created, allowing Scapy and its dependencies to be installed in an isolated and safe manner without affecting system-managed Python packages. This approach aligns with recommended security and development practices.

6.2 Firewall Rule Ordering in iptables

While testing the firewall configuration, ICMP traffic was initially not blocked as expected, even after applying a default DROP policy. This occurred due to the presence of an ESTABLISHED, RELATED rule, which allowed return traffic for existing connections. The issue was resolved by understanding and applying correct rule ordering in iptables. A temporary ICMP block rule was inserted at a higher priority position in the INPUT chain to demonstrate effective traffic blocking. After validation, the rule was removed to restore the intended firewall configuration.

6.3 OpenVAS Feed Synchronization and Scan Configuration Issues

During the vulnerability scanning phase, OpenVAS encountered feed synchronization and cache lock issues, which prevented scan configuration selection and execution. This behavior is commonly observed in fresh OpenVAS installations due to the large size of vulnerability feeds and dependency on successful feed synchronization.

Multiple troubleshooting steps were attempted, including service restarts, database rebuild attempts, and feed status verification. The limitations were documented with supporting screenshots, demonstrating a clear understanding of tool behavior and operational constraints.

6.4 Troubleshooting Approach

Each issue was approached methodically by:

- Reviewing error messages and logs
- Understanding tool-specific constraints
- Applying recommended fixes and best practices
- Documenting both successful resolutions and known limitations

This approach reflects real-world security operations, where not all issues can be resolved immediately and proper documentation is critical.

Conclusion of Troubleshooting

The challenges encountered during this task reinforced the importance of environment awareness, configuration validation, and structured troubleshooting. Addressing these issues contributed significantly to practical learning and strengthened problem-solving skills relevant to cybersecurity roles.

7. Key Learnings

This task provided several practical insights that extend beyond basic tool usage and reflect real-world cybersecurity operations.

7.1 Network Traffic Behavior and Visibility

Packet sniffing highlighted that network traffic is highly dependent on system activity and timing. The absence of ICMP traffic during packet capture demonstrated that certain protocols only appear when explicitly triggered. This reinforced the understanding that effective network monitoring requires contextual awareness and controlled traffic generation.

7.2 Firewall Rule Evaluation and Execution Order

The firewall configuration phase demonstrated that rule effectiveness is directly influenced by rule order. The behavior of ICMP traffic being allowed due to established-connection rules emphasized the importance of understanding packet flow and rule processing logic rather than relying solely on rule definitions.

7.3 Principle of Least Privilege in Network Security

Implementing a default-deny firewall policy reinforced the importance of minimizing system exposure. Allowing only essential services such as SSH and HTTP significantly reduced the attack surface and reflected industry-standard server hardening practices.

7.4 Practical Limitations of Vulnerability Management Tools

The OpenVAS phase highlighted operational limitations related to feed synchronization and configuration dependencies. This experience demonstrated that vulnerability management tools require proper environmental readiness and that tool constraints must be understood and documented as part of professional security workflows.

7.5 Importance of Structured Documentation

Throughout the task, detailed documentation of configurations, outputs, and troubleshooting steps proved essential. This reinforced that clear documentation is critical for reproducibility, knowledge transfer, and effective communication in security operations.

7.6 Analytical Problem-Solving in Security Operations

The task emphasized analytical problem-solving over tool-specific knowledge. Identifying issues, understanding system behavior, and applying logical fixes highlighted the importance of a methodical approach to cybersecurity challenges.

Summary

Overall, the task strengthened practical understanding of network monitoring, access control, and vulnerability assessment while reinforcing disciplined troubleshooting and documentation practices aligned with real-world cybersecurity roles.

8. Conclusion

This task provided comprehensive hands-on exposure to multiple layers of network security, reinforcing both technical understanding and practical implementation skills. Through packet sniffing, live network traffic was captured and analyzed, enabling deeper insight into protocol behavior and real-time communication patterns. Firewall configuration using iptables demonstrated how access control mechanisms can be applied to reduce system exposure through a default-deny approach while maintaining required services.

The vulnerability scanning phase introduced practical aspects of vulnerability management, including tool configuration, feed dependencies, and operational constraints commonly encountered in real-world environments. Encountered limitations were analyzed and documented, reinforcing the importance of understanding tool behavior rather than relying solely on expected outcomes.

Overall, this task successfully bridged theoretical security concepts with real-world execution. It strengthened foundational knowledge in network monitoring, system hardening, and vulnerability assessment while emphasizing structured troubleshooting and detailed documentation—skills essential for defensive security and SOC-oriented roles.

9. References / Sources

- Scapy Official Documentation – <https://scapy.net/>
- Kali Linux Documentation – <https://www.kali.org/docs/>
- iptables Manual Pages – <https://man7.org/linux/man-pages/man8/iptables.8.html>
- Greenbone Vulnerability Management Documentation – <https://greenbone.net>