एस. गोपालन, भा.दू. से.
S. Gopalan, ITS
उप महानिदेशक (सुरक्षा प्रमाणीकरण)
Deputy Director General (Security Certification)

भारत सरकार
Government of India
संचार मंत्रालय
Ministry of Communications
दूरसंचार विभाग
Department of Telecommunications
राष्ट्रीय संचार सुरक्षा केंद्र
National Centre for Communication Security
https://nccs.gov.in/

No. NCCS/SC/2-3/2025-26 Dated 13-04-2026

INTERNSHIP COMPLETION CERTIFICATE

This is to Certify that Mr. Abimanyu A, recommended by Dayananda Sagar College of Engineering, Bangalore, has successfully completed his internship as Category-I Intern with National Centre for Communication Security, Department of Telecommunications, Ministry of Communications, Government of India from 13-02-2026 to 13-04-2026. During the period of internship, he worked under Security Certification Division in the following areas:

1. Developed a framework - ITSAR Compliance Automation Framework (ICAF), that transformed the testing of ITSAR clauses from manual to automatic process. All the test cases pertaining to the ITSAR clause are executed, results collected and collated into final report automatically.
2. ICAF is a centralized system of an intuitive command-line tool to execute, track, and verify security tests efficiently. It automates testing, reporting and validation processes, thereby significantly removing the manual effort required for the same.
3. Demonstrated ICAF tool use for testing of Mutual Authentication ITSAR clause.
4. Created a manual on how to expand the ICAF tool to write code for new ITSAR clauses and test cases.
5. Provided flexibility within the ICAF tool to support multiple OEMs' devices with differing operating systems.

2. He has shown special flair for Python based automation and his performance during the internship is rated as outstanding.

3. I wish him every success in his future endeavours.

S. Gopalan 13/4/26

(S. Gopalan)

एस. गोपालन, भा दू से
S. GOPALAN, ITS
उप महानिदेशक (सु प्र)
Deputy Director General (SC)
राष्ट्रीय संचार सुरक्षा केंद्र
National Centre for Communication Security
दूरसंचार विभाग/Department of Telecommunication
बेंगलूरु/Bengaluru

INTERNSHIP REPORT

ITSAR Compliance Automation Framework (ICAF)

Tester Operation Manual & Documentation

Submitted By:

Abimanyu A
SC Department
NCCS

Submitted To:

Sh. Sumit Singh
ADG (Security Certification-I)

Date: April 13, 2026

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my internship supervisor, **Sh. Sumit Singh**, for their invaluable guidance, continuous support, and constructive feedback throughout this internship. Their expertise in telecom security and test automation has been instrumental in the successful completion of this project.

Special thanks to the Research Associates for clarifying requirements and providing access to test devices and documentation that made this work possible.

Finally, I extend my appreciation to **NCCS** and my team members Gagan Deep, Bhavya Sharma, Ageline Naha and Shiv Kylash for their encouragement and for fostering an environment of technical excellence.

Abimanyu A
13-04-2026

Internship Work Summary

During the course of the internship, significant contributions were made towards the design, development of the **ITSAR Compliance Automation Framework (ICAF)**. The key achievements are summarized below:

1. **Development of ICAF Framework:** Designed and developed the **ITSAR Compliance Automation Framework (ICAF)**, which transformed the testing of ITSAR clauses from a manual process into a fully automated workflow. All test cases related to ITSAR clauses are executed automatically, with results collected and compiled into a final structured report.
2. **Centralized Command-Line Tool:** Built ICAF as a centralized and intuitive command-line tool that enables efficient execution, tracking, and verification of security tests. The framework automates testing, validation, and reporting processes, significantly reducing manual effort.
3. **Demonstration of Use Case:** Successfully demonstrated the usage of the ICAF tool for testing the **Mutual Authentication ITSAR clause (Clause 1.1.1)**, showcasing its practical applicability in real-world telecom security evaluation scenarios.
4. **Extensibility Documentation:** Created a comprehensive manual detailing how to extend the ICAF framework. This includes guidelines for writing new test cases and integrating additional ITSAR clauses into the system.
5. **Multi-OEM Support:** Enhanced the flexibility of the framework to support devices from multiple OEMs with varying operating systems. Implemented a vendor-adapter approach to ensure compatibility across diverse device environments.

Contact Details

Name:	Abimanyu A
Department:	SC Department, NCCS
Email:	itsactuallymeabi@gmail.com
Phone:	+91-9080116801
Supervisor:	Sh. Sumit Singh
Designation:	ADG (Security Certification-I)
Organization:	National Centre for Communication Security (NCCS)

For any queries or further information regarding this project, please feel free to reach out.

Contents

Abstract

The ITSAR Compliance Automation Framework (ICAF) is a comprehensive security evaluation platform designed to automate telecom device compliance testing based on the Indian Telecom Security Assurance Requirements (ITSAR). This report documents the complete design, implementation, and operational procedures of ICAF, which replaces traditional manual testing workflows with a unified, automated execution system.

The framework currently supports automated validation across two critical ITSAR clauses: Clause 1.1.1 (Management Protocols Entity Mutual Authentication) and Clause 1.6.1 (Cryptographic Based Secure Communication). ICAF employs a modular, layered architecture comprising terminal managers, browser automation, packet capture systems, and evidence management subsystems.

Key features documented in this report include the DUT profiling system for device-specific configuration, the OAM module for protocol verification, the automation step library for constructing testcases, and the report generation engine that produces structured compliance reports with embedded evidence artifacts. All generated evidence—including terminal screenshots, browser screenshots, packet captures, and command outputs—is systematically organized and traceable to specific testcases.

This report serves both as technical documentation for security evaluators and as an operational manual for test execution, covering system requirements, setup procedures, test execution workflows, troubleshooting guidelines, and future enhancement plans.

Keywords: ITSAR Compliance, Telecom Security, Test Automation, Protocol Validation, Security Evaluation, SSH, SNMPv3, TLS, gRPC, Packet Capture

1 Introduction

The **ITSAR Compliance Automation Framework (ICAF)** is a security evaluation platform designed to automate telecom device compliance testing based on **ITSAR (Indian Telecom Security Assurance Requirements)**.

The framework assists testers in executing protocol validation procedures, capturing technical evidence, and generating structured compliance reports suitable for telecom certification audits. ICAF replaces traditional manual testing workflows - involving multiple tools such as SSH clients, packet analyzers, and browser utilities - with a unified, automated execution system.

1.1 Scope of the Internship

This internship involved the design, development, documentation, and testing of the ICAF system. The specific deliverables included:

- Development of automation modules for Clause 1.1.1 and Clause 1.6.1 validation
- Implementation of the step-based execution engine and automation step library
- Design of the DUT profiling system and vendor adapter architecture
- Integration of packet capture and analysis capabilities
- Development of the report generation engine with embedded evidence
- Creation of comprehensive operational documentation
- Testing and validation against multiple DUT types (Linux, OpenWRT, Cisco)

1.2 Supported ITSAR Clauses

ICAF currently supports automated validation across two ITSAR clauses:

- **Clause 1.1.1** - Management Protocols Entity Mutual Authentication (SSH, HTTPS, SNMPv3, gRPC/gNMI)
- **Clause 1.6.1** - Cryptographic Based Secure Communication (cipher suite analysis for SSH, TLS, SNMP)

Generated evidence artifacts include:

- terminal screenshots
- browser screenshots
- packet capture files (.pcapng)
- structured compliance reports (.docx)

2 System Requirements

2.1 Hardware Requirements

2.2 Operating System

ICAF is designed to run on Linux-based systems. Supported distributions:

- Ubuntu 22.04+
- Kali Linux
- Debian 11+

Component	Minimum Requirement
CPU	4 Core Processor
RAM	8 GB
Disk Space	20 GB Free
Network	Ethernet Interface

Table 1: Hardware Requirements

2.3 Required Software Dependencies

2.3.1 Python Environment

```
Python 3.10+
```

Install Python dependencies:

```
pip install -r requirements.txt
```

2.3.2 Network and Security Tools

Tool	Purpose
OpenSSL	TLS protocol testing
Wireshark / tshark	Packet analysis and frame extraction
tcpdump	Packet capture
Nmap	Service discovery and cipher scanning
SNMP utilities	SNMPv1/v2c/v3 protocol testing
grpcurl	gRPC/gNMI mutual authentication testing

Table 2: Required External Tools

Example installation:

```
sudo apt install wireshark tshark tcpdump nmap snmp openssl
```

2.3.3 Browser Automation

ICAF uses Selenium-based browser automation to validate HTTPS management interfaces. Required components:

- Firefox browser
- GeckoDriver

Installation example:

```
sudo apt install firefox-esr
```

3 Initial Setup

3.1 Clone the Repository

```
git clone <repository-url>
cd icaf
```

3.2 Create Python Virtual Environment

```
python3 -m venv myenv
source myenv/bin/activate
pip install -r requirements.txt
```

3.3 Configure the Environment File

ICAF reads all credentials and connection parameters from a `.env` file in the project root. No interactive credential prompts are used during execution. Create and populate the `.env` file before running any tests:

```
# DUT SSH credentials
SSH_USER=admin
DUT_IP=10.99.214.214
SSH_PASSWORD=your_password

# SNMPv3 credentials
SNMP_USER=snmpuser
SNMP_AUTH_PASS=authpassword
SNMP_PRIV_PASS=privpassword
SNMP_COMMUNITY=public

# Web interface credentials
WEB_LOGIN_URL=https://10.99.214.214
WEB_USERNAME=admin
WEB_PASSWORD=your_password

# Network interface used for packet capture
INTERFACE=eth0
```

3.4 Verify Tool Installation

```
openssl version
tshark --version
nmap --version
snmpwalk --version
```

4 Testbed Setup

4.1 Device Under Test (DUT)

The DUT should be a telecom device supporting remote management protocols. Examples:

- Routers
- Access Points
- Network Controllers
- Customer Premises Equipment (CPE)

4.2 Network Configuration

The tester machine must have network connectivity to the DUT management interface.

```
Tester Machine
      |
      |  Management Network (Ethernet)
      |
Device Under Test (DUT)
```

Verify reachability before starting:

```
ping <dut_ip>
```

5 Running ICAF

ICAF is invoked through its CLI. The primary command is `icaf run`, which accepts the following options:

Option	Description
`--clause`	Execute a specific ITSAR clause (e.g., `1.1.1` or `1.6.1`)
`--profile`	Select a DUT profile by name (default: `default`)
`--oam`	Path to an OAM Excel file for protocol configuration data
`--section`	Execute all clauses within a given ITSAR section

Table 3: CLI Options

Example commands:

```
# Run Clause 1.1.1 using the default DUT profile
icaf run --clause 1.1.1

# Run Clause 1.1.1 using a specific DUT profile
icaf run --clause 1.1.1 --profile alpine

# Run Clause 1.1.1 with an OAM Excel file
icaf run --clause 1.1.1 --oam /path/to/oam_document.xlsx

# Run Clause 1.6.1
icaf run --clause 1.6.1
```

All credentials are read automatically from the `.env` file at startup. The engine validates that `SSH_USER`, `DUT_IP`, and `SSH_PASSWORD` are present before proceeding.

6 DUT Profiling System

ICAF uses a **DUT profile** to capture device-specific configuration before testing begins. Profiles define the commands, prompts, selectors, and protocol parameters that the framework uses when interacting with the DUT.

Profiles are stored in `icaf/profile/` and are available in two formats:

- **YAML** - human-readable configuration used by all testcases
- **Excel (.xlsx)** - an alternate format that mirrors the YAML structure for OAM compatibility

Example profile directory:

```
icaf/profile/
    default.yaml
    default.xlsx
    alpine.yaml
    alpine.xlsx
    metasploitable.yaml
```

A DUT profile captures SSH connection options, prompt patterns, SNMP parameters, web interface selectors, and gRPC configuration. An abbreviated example:

```
ssh:
  base: ssh
  target: "{user}@{ip}"
  connect_options: []
  password_prompt:
    - "password"
    - "Password"
  success_prompt:
    - "$"
    - "#"
    - ">"
  failure_prompt:
    - "Permission␣denied"
    - "Connection␣refused"

snmp:
  target: "10.99.214.214"
  user: "snmpuser"
  auth_pass: "authpassword"
  priv_pass: "privpassword"
  community: "public"

web:
  login_url: "https://10.99.214.214"
  username_field: "#username"
  password_field: "#password"
  submit_selector: "#login-btn"

grpc:
```

```
  port: 50051
  tls_cert: "certs/client.crt"
  tls_key:  "certs/client.key"
  ca_cert:  "certs/ca.crt"
```

The active profile is selected at runtime using the `--profile` flag. When a DUT does not support a particular protocol (e.g., gRPC), the corresponding testcase is either skipped automatically or disabled in the clause definition.

6.1 Vendor Adapters

ICAF includes a **vendor adapter system** that translates generic profile operations into device-specific commands. Available adapters:

- `LinuxAdapter` - generic Linux-based devices
- `CiscoAdapter` - Cisco IOS / IOS-XE devices
- `OpenWRTAdapter` - OpenWRT-based access points

The `AdapterFactory` selects the correct adapter based on the detected device type, allowing testcases to remain device-agnostic while the adapter handles vendor-specific command differences.

7 OAM Module

The **OAM (Operations, Administration, Maintenance) module** processes Excel documents supplied by the OEM that describe the management protocols configured on the DUT. When an OAM file is provided via `--oam`, ICAF:

1. Parses the Excel file to extract the list of declared management protocols (`raw_protocols`).
2. Runs a live network scan against the DUT to verify which protocols are actually active (`verified_protocols`).
3. Passes the combined context (`oam_context`) to the report generator.

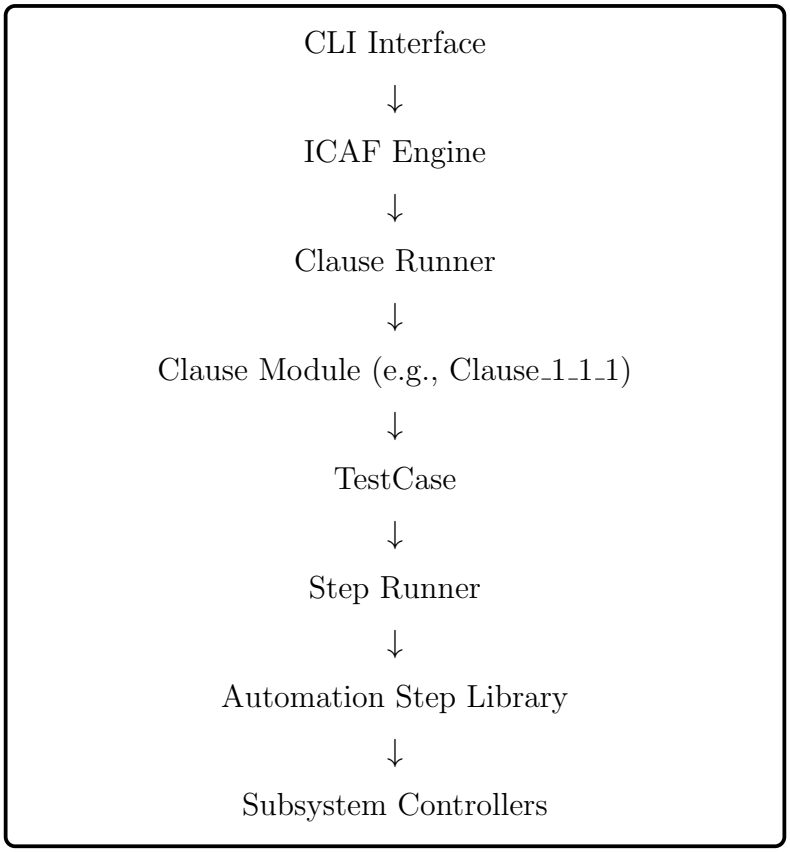
The report uses this data to produce a protocol verification table showing:

- protocols that are both configured and detected (Verified)
- protocols that are configured but not detected (Not Detected)
- protocols that are detected but not declared (Unexpected)

This verification table forms part of the precondition evidence in the compliance report.

8 Core Architecture

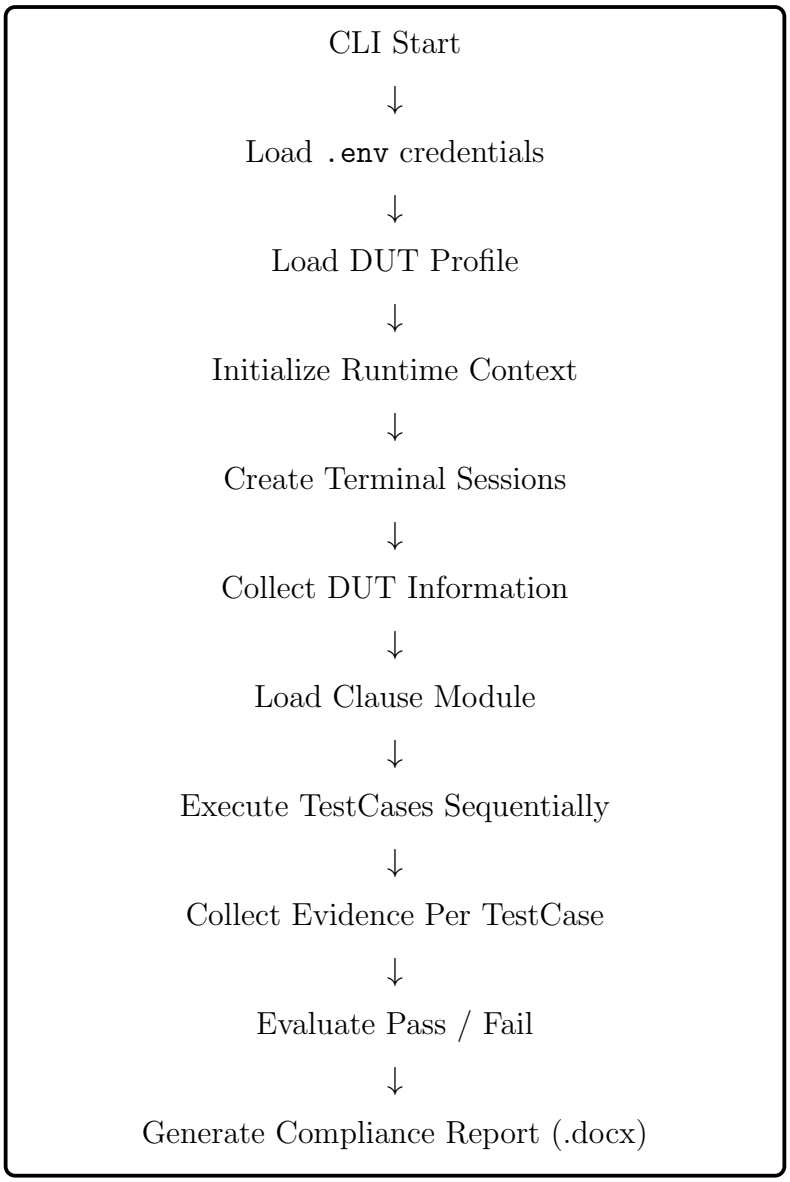
ICAF follows a modular, layered architecture. Each layer has a clearly defined responsibility:



Subsystem controllers include:

- **Terminal Manager** - controls visible terminal windows
- **Browser Manager** - controls the Selenium-driven Firefox browser
- **Packet Capture Controller** - manages tcpdump processes
- **Evidence Manager** - allocates and tracks per-testcase output directories
- **Device Detector** - identifies the DUT operating system via SSH
- **Adapter Factory** - selects the vendor-specific command adapter
- **Report Manager** - triggers clause-specific report generation

9 Execution Flow



10 Runtime Context

The `RuntimeContext` object is created once per execution and passed to every testcase and step. It carries all shared state throughout the run, including:

- SSH credentials and connection parameters
- SNMP and web interface credentials
- References to the terminal manager, browser manager, and adapter
- `current_testcase` - a pointer to the testcase currently executing, used by evidence collection
- `evidence` - the `EvidenceManager` instance
- `pcap_process` and `pcap_file` - state for the active packet capture
- DUT metadata (name, firmware version, OS hash, config hash)
- OAM context (if an OAM file was provided)

11 Evidence Storage

ICAF automatically organises all evidence artifacts into a structured output directory. The **EvidenceManager** is initialised at the start of each run and creates the root run directory using a timestamp and the clause identifier:

`output/runs/{YYYY-MM-DD_HH-MM-SS}-{clause}/`

For each testcase that executes, the EvidenceManager creates a dedicated subdirectory containing three sub-folders:

```
output/runs/{timestamp}-1.1.1/
    1.1.1/
        TC1_SNMPV3_POSITIVE/
            screenshots/
            logs/
            pcap/
        TC2_SNMPV3_INVALID_CREDENTIALS/
            screenshots/
            logs/
            pcap/
        TC3_SSH_MUTUAL_AUTH/
            screenshots/
            logs/
            pcap/
        TC4_SSH_CORRECT_PUBLIC_KEY/
            screenshots/
            logs/
            pcap/
        TC6_HTTPS_VALID_LOGIN/
            screenshots/
            logs/
            pcap/
        TC7_HTTPS_INVALID_LOGIN/
            screenshots/
            logs/
            pcap/
```

Each sub-folder holds a distinct evidence type:

Folder	Contents
`screenshots/`	PNG images captured from terminal windows and the browser. Filenames encode the testcase name, terminal identifier, and a timestamp (e.g., `TC3_SSH_MUTUAL_AUTH_tester_143012_000123.png`).
`logs/`	Command output text files captured during step execution.
`pcap/`	Network packet captures in `.pcapng` format recorded by tcpdump during protocol interactions.

Table 4: Evidence Sub-folder Types

11.1 Evidence Registration

As each artifact is saved to disk, it is **registered against the active testcase**. The `RuntimeContext` maintains a `current_testcase` pointer that is set at the start of each testcase and updated as execution advances. When a step produces an artifact (screenshot, PCAP, command output), it calls `testcase.add_evidence()` to attach a record containing:

- `command` - the command that was executed (if applicable)
- `output` - the terminal output captured (if applicable)
- `screenshot` - the absolute file path of the saved screenshot
- `caption` - a human-readable description of what the screenshot shows

This registration ensures that every artifact is traceable to the exact testcase and step that produced it.

12 Report Generation

At the end of execution, ICAF generates a structured **Word document (.docx)** compliance report. The report is produced by the `ReportManager`, which delegates to a clause-specific report class selected by `ReportFactory`.

12.1 Report Configuration Sources

The report generator draws from three sources:

1. `reporting/specs/clause_X_X_X.yaml` - the primary content specification. This YAML file defines the ITSAR requirement text, DUT configuration notes, preconditions, test objective, test plan, and per-testcase observations and conclusions. Each testcase carries three variants of its observation, conclusion, and remarks fields - one for each possible outcome: `_pass`, `_fail`, and `_not_run`. The generator selects the correct variant automatically based on the runtime status of each testcase.
2. `reporting/report_config.yaml` - document-level metadata that does not change between runs: reviewer names, organisation name, revision history entries, and prepared-for text.
3. `RuntimeContext` - per-run values including DUT name, firmware version, OS hash, config hash, SSH IP, start time, and the evidence run directory path.

12.2 Evidence in the Report

Evidence collected during test execution is embedded directly into the report document. For each testcase, the report generator iterates over `testcase.evidence` - the list of registered evidence records described in the previous section. For each record:

- If a `command` is present, it is rendered as a monospace terminal block showing the exact command that was executed.
- If an `output` string is present, it is rendered as a monospace output block.
- If a `screenshot` path is present, the image file is read from disk and embedded as an inline figure with its associated `caption` printed beneath it.

Packet capture evidence is also referenced by path in the relevant testcase sections. Be-

cause all screenshot paths stored in the evidence records are absolute file paths within the run directory, the report generator does not need to search for files - it reads exactly the files that were registered during execution.

12.3 AI-Enriched Observations

The report generator will soon include an optional AI-powered enrichment feature that enhances pre-authored YAML observation text with specific details derived from actual command outputs and screenshot captions collected at runtime. Instead of relying on generic templated text, this feature will produce observations that reference the exact responses observed during testing. Enrichment will only be applied when real evidence exists for a given testcase; testcases marked as `NOT_RUN` will continue to use the `_not_run` YAML variant without any enrichment. This capability will be powered entirely by a local AI model, with no external API dependencies.

12.4 Report Contents

The generated report includes the following sections:

- Cover page with DUT name, version, and tester organisation details
- Revision history
- ITSAR requirement reference and description
- DUT configuration summary (management protocols and OAM verification table if applicable)
- Preconditions
- Test objective
- Test plan (tools used, scope, applicable ITSAR references)
- Test execution section with one entry per testcase, each containing: testcase description, step-by-step execution narrative, embedded screenshots, command blocks, packet capture references, pass/fail status, and conclusion
- Result summary table across all testcases
- Overall compliance conclusion

The report is saved to the run directory as:

```
output/runs/{timestamp}-{clause}/tcaf_report.docx
```

13 Terminal Manager

The terminal manager controls multiple visible terminal sessions throughout test execution. Two persistent terminals are created at startup and shared across all testcases:

- **tester** - used for commands executed from the tester machine (SSH, OpenSSL, snmpwalk, grpcurl, nmap, etc.)
- **dut** - available for direct DUT interaction when required

Using visible terminals is an intentional design choice: testers can observe command execution in real time, screenshots capture the actual terminal state, and the environment closely resembles manual testing. Each terminal is assigned a logical name and accessed through `context.terminal_manager.get_terminal(name)`.

14 Packet Capture System

Network traffic is captured using **tcpdump** for all protocol interactions that require network-level verification. The packet capture lifecycle within a testcase follows this pattern:

```
PcapStartStep  - launches tcpdump on the configured interface
                 (writes to testcase pcap/ directory)
                 [protocol interaction steps]
PcapStopStep   - sends SIGINT to tcpdump, waits for graceful flush,
                 escalates to SIGTERM/SIGKILL if needed,
                 confirms file bytes written before returning
AnalyzePcapStep - runs tshark with a display filter, extracts
                  per-packet fields (frame number, source/dest IP,
                  protocol, TLS version, cipher suite, SSH fields,
                  SNMP version), stores results on context
WiresharkPacketScreenshotStep - opens the pcap in Wireshark, navigates
                                to the relevant frame, captures screenshot
                                as evidence
```

Captured packets are stored in the testcase's `pcap/` subdirectory and are referenced in the compliance report as supporting evidence for protocol behaviour verification.

15 Implemented Clauses and Testcases

15.1 Clause 1.1.1 - Management Protocols Entity Mutual Authentication

This clause verifies that each management protocol supported by the DUT enforces mutual authentication. All testcases are composed from the automation step library.

15.1.1 TC1 - SNMPv3 Positive Authentication

Objective: Verify that SNMPv3 with correct credentials succeeds, and that legacy SNMPv1 and SNMPv2c are disabled.

The testcase first attempts `snmpget` using SNMPv1 and SNMPv2c and verifies that both return a timeout (no response), confirming they are disabled. It then executes an SNMPv3 `snmpget` using valid credentials and verifies a successful response is returned. Packet captures are taken for each interaction and Wireshark screenshots are attached.

15.1.2 TC2 - SNMPv3 Invalid Credentials

Objective: Verify that SNMPv3 rejects authentication attempts using incorrect credentials.

An SNMPv3 `snmpget` is attempted using a deliberately invalid username or wrong authentication password. The testcase verifies that no valid response is returned and that the authentication failure is observable in the packet capture.

15.1.3 TC3 - SSH Mutual Authentication (Password)

Objective: Verify that SSH enforces authentication using a password, accepts correct credentials, and rejects incorrect credentials.

The testcase first connects using the correct password and verifies a shell prompt is received. It then connects using an incorrect password and verifies a `Permission denied` response. Packet captures are taken for both interactions. A terminal screenshot is captured after each significant step.

15.1.4 TC4 - SSH Public Key Authentication (Correct Key)

Objective: Verify that SSH successfully authenticates when a valid public key is presented.

The testcase connects to the DUT using an SSH public/private key pair that has been pre-authorised on the device. It verifies that authentication succeeds and a shell prompt is received without a password prompt appearing.

15.1.5 TC5 - SSH Public Key Authentication (Incorrect Key)

Objective: Verify that SSH rejects a public key that has not been authorised on the DUT.

Note: This testcase is currently defined but disabled in the Clause 1.1.1 execution configuration pending DUT environment setup.

15.1.6 TC6 - HTTPS Valid Login

Objective: Verify that the HTTPS web management interface authenticates successfully with correct credentials.

The browser manager navigates to the DUT web interface URL. The auto-login step fills the username and password fields and submits the login form. The testcase verifies that the post-login dashboard is displayed, and a browser screenshot is captured as evidence.

15.1.7 TC7 - HTTPS Invalid Login

Objective: Verify that the HTTPS web management interface rejects incorrect credentials.

The same browser interaction is repeated using a wrong password. The testcase verifies that an authentication failure indicator (error message or login page reload) is displayed rather than the dashboard, and a browser screenshot is captured.

15.1.8 TC8 - gRPC/gNMI Mutual Authentication

Objective: Verify that the gRPC/gNMI management interface requires mutual TLS certificate authentication.

Note: This testcase is currently defined but disabled in the Clause 1.1.1 execution configuration pending PKI certificate provisioning for the test environment.

15.2 Clause 1.6.1 - Cryptographic Based Secure Communication

Clause 1.6.1 verifies that all management protocols use strong cryptographic algorithms and that weak ciphers and legacy protocol versions are rejected. Unlike Clause 1.1.1, this clause uses a **scanner-based execution model** rather than step-based testcases. Each scanner function runs a targeted tool against the DUT and returns structured results.

Protocol applicability is determined automatically using an Nmap service discovery scan at the start of the clause. Tests for a protocol are skipped if the scan does not detect that protocol as active on the DUT.

15.2.1 SSH Tests

- **SSH Cipher Detection** - enumerates all SSH encryption algorithms, MAC algorithms, key exchange methods, and host key types supported by the DUT.
- **SSH Secure Communication Verification** - verifies that the SSH session is encrypted using strong algorithms meeting ITSAR requirements.
- **SSH Weak Cipher Negotiation** - attempts to establish an SSH connection forcing weak cipher suites and verifies the DUT rejects the negotiation.
- **SSH None-Cipher Rejection** - attempts an SSH connection with `Ciphers=none` and verifies the DUT rejects the connection.

15.2.2 HTTPS / TLS Tests

- **HTTPS TLS Cipher Detection** - enumerates all TLS cipher suites supported on the DUT HTTPS service.
- **HTTPS TLS Secure Communication** - verifies that the TLS session uses a strong TLS version (1.2 or 1.3) and acceptable cipher suites.
- **HTTPS Weak Cipher Negotiation** - attempts a TLS connection forcing weak cipher suites and verifies the DUT rejects the negotiation.
- **HTTPS NULL Cipher Rejection** - attempts a TLS connection using NULL (no-encryption) cipher suites and verifies rejection.

15.2.3 SNMP Tests

- **SNMP Version Check** - verifies that SNMPv1 and SNMPv2c are disabled and only SNMPv3 is active.
- **SNMP Secure Communication Verification** - verifies that SNMPv3 is configured with authenticated and privacy-protected (authPriv) security level.

16 Automation Step Library

ICAF uses a modular **automation step library** to construct testcases. Each step represents a single, reusable automation operation. Testcases compose multiple steps using the `StepRunner` utility, which executes each step in sequence and halts on failure.

16.1 CommandStep

Sends a shell command to a specified terminal session.

```
CommandStep("tester", "ssh user@10.10.10.1", settle_time=4)
```

16.2 InputStep

Sends keystrokes to an active terminal prompt (e.g., a password field).

```
InputStep("tester", context.ssh_password)
```

16.3 ExpectOneOfStep

Waits for any one of a list of output patterns to appear in the terminal buffer. The first matched pattern determines the execution branch.

```
ExpectOneOfStep("tester", ["password", "continue connecting"],
   ↪ timeout=10)
```

16.4 VerifyOutputStep

Asserts that a specific string is present in the terminal output. Fails the testcase if the string is absent.

```
VerifyOutputStep("tester", "Permission denied")
```

16.5 CheckOutputStep

Performs a pass/fail check against terminal output using expected or forbidden patterns.

```
CheckOutputStep("tester", expected="sysUpTime", forbidden="Error
   ↪ ")
```

16.6 WaitStep

Pauses execution for a fixed number of seconds.

```
WaitStep(5)
```

16.7 WaitForPatternStep

Blocks until a specific pattern appears in the terminal output.

```
WaitForPatternStep("tester", "password")
```

16.8 ClearTerminalStep

Sends a `clear` command to the terminal to remove previous output.

16.9 SessionResetStep

Interrupts any active process (`Ctrl+C`), clears the terminal, and returns it to a known clean state. Used between testcases.

16.10 EnsureSSHDisconnectedStep

Interrupts any active SSH session and returns the terminal to the local shell prompt.

16.11 OpenURLStep

Instructs the browser manager to navigate to a specified URL.

```
OpenURLStep("https://10.99.214.214")
```

16.12 ClickStep

Simulates a click on a web element identified by a CSS selector or element ID.

```
ClickStep("#login-btn")
```

16.13 FillInputStep

Enters text into a browser form field.

```
FillInputStep("#username", "admin")
```

16.14 AutoLoginStep

Combines username entry, password entry, and login button interaction into a single compound operation.

```
AutoLoginStep(context.web_username, context.web_password)
```

16.15 ScreenshotStep

Captures a screenshot of the specified terminal window. The screenshot is saved to the active testcase's `screenshots/` directory and automatically registered against the testcase via `testcase.add_evidence()`.

```
ScreenshotStep("tester", caption="SSH authentication prompt
    ↪ observed")
```

16.16 BrowserScreenshotStep

Captures a screenshot of the current browser viewport using the Selenium driver. The file is saved to the testcase's `screenshots/` directory and registered as evidence.

```
BrowserScreenshotStep("login_success.png", caption="Dashboard
    ↪ visible after login")
```

16.17 PcapStartStep

Starts a tcpdump process on the configured network interface. The capture file is written to the active testcase's `pcap/` directory.

```
PcapStartStep(interface="eth0", filename="tc3_ssh_positive.
    ↪ pcapng")
```

16.18 PcapStopStep

Stops the active tcpdump process. Sends SIGINT for a graceful flush and waits for the process to exit. Escalates to SIGTERM then SIGKILL if the process does not terminate within the timeout. Confirms that the PCAP file has non-zero bytes before returning.

16.19 AnalyzePcapStep

Runs tshark against the captured PCAP file using a display filter. Extracts per-packet fields including frame number, timestamp, source/destination IP, protocol, TLS version, cipher suite, SSH protocol string, and SNMP version. Stores structured results on `context.pcap_packets` for use by subsequent steps and the report generator.

16.20 WiresharkPacketScreenshotStep

Opens the PCAP file in Wireshark, navigates to the relevant frame identified by `AnalyzePcapStep`, and captures a screenshot. The screenshot is registered as evidence and embedded in the compliance report.

17 Pass/Fail Verification Mechanism

Each testcase contains verification logic that evaluates observed protocol behaviour against ITSAR-defined success and failure conditions. Verification is implemented using the `VerifyOutputStep`, `CheckOutputStep`, and `ExpectOneOfStep` steps at specific points in the execution sequence.

Pass condition example - SSH authentication:

```
Correct credentials produce a shell prompt ($, #, or >)
Incorrect credentials produce "Permission denied"
Authentication prompt appears before any access is granted
```

Fail condition example - SSH authentication:

```
Login succeeds with incorrect credentials
Authentication prompt does not appear
Connection bypasses authentication entirely
```

If any step raises an exception (unexpected output, pattern not found, timeout), the testcase's `fail_test()` method is called and the status is recorded as `FAIL`. If all steps complete without error, `pass_test()` is called and the status is recorded as `PASS`. Testcases that are skipped (e.g., due to a disabled protocol) are recorded as `NOT_RUN`.

18 Troubleshooting

18.1 SSH Connection Failure

Check that:

- `SSH_USER`, `DUT_IP`, and `SSH_PASSWORD` are correctly set in `.env`
- SSH service is enabled on the DUT
- The tester machine has network connectivity to the DUT management IP
- No firewall rule blocks port 22

18.2 HTTPS Access Failure

Verify that:

- `WEB_LOGIN_URL` includes the correct scheme and port
- The web interface is enabled on the DUT
- Firefox and GeckoDriver are installed and compatible versions

18.3 SNMP Test Failures

Check that SNMP utilities (`snmpwalk`, `snmpget`) are installed. Verify that the SNMPv3 credentials in `.env` match the DUT configuration exactly, including authentication and privacy protocols.

18.4 Packet Capture Failures

Ensure `tcpdump` and `tshark` are installed and that the tester user has permission to capture on the interface specified by `INTERFACE` in `.env`. Running with `sudo` or adding the user to the `wireshark` group is typically required.

18.5 Report Generation Failures

Confirm that the Python `python-docx` library is installed via `requirements.txt`. If AI enrichment is enabled, verify that `ANTHROPIC_API_KEY` is valid and the API is reachable.

19 Future Enhancements

Planned improvements for future ICAF releases:

- Full activation of TC5 (SSH incorrect public key) and TC8 (gRPC/gNMI mutual auth)
- Additional clause modules (Clause 1.2, Clause 1.3, etc.)
- Automatic DUT operating system detection integrated into the profile loader
- Extended vendor adapter library (Juniper, Huawei, MikroTik)
- Distributed testing support for multi-DUT environments
- CI/CD pipeline integration
- Automated cryptographic strength scoring

20 Conclusion

The ITSAR Compliance Automation Framework significantly reduces the complexity and effort of telecom security evaluation by automating protocol validation, evidence collection, and compliance report generation. The combination of a step-based execution model for Clause 1.1.1 and a scanner-based model for Clause 1.6.1 provides a flexible foundation that can be extended as additional ITSAR clauses are onboarded. By maintaining a strict separation between evidence collection (on-disk, per-testcase directories) and evidence presentation (embedded in the Word report via registered evidence records), ICAF ensures that every result in a compliance report is directly traceable to a verifiable artifact captured during test execution.

20.1 Internship Outcomes

During this internship, the following objectives were achieved:

- Successfully designed and implemented the ICAF architecture with modular components
- Developed automation support for two critical ITSAR clauses covering SSH, HTTPS, SNMPv3, and TLS
- Created a comprehensive step library with 18 reusable automation steps
- Implemented evidence management with structured directory organization
- Built a report generation engine producing professional compliance documents
- Validated the framework against multiple device types including Linux, OpenWRT, and Cisco devices
- Produced complete operational documentation for security evaluators

20.2 Recommendations

Based on the development and testing experience, the following recommendations are made:

- Extend vendor adapter support to additional telecom equipment manufacturers
- Implement automated DUT fingerprinting to eliminate manual profile selection
- Add support for RESTCONF/NETCONF management protocols
- Integrate with continuous integration pipelines for regression testing
- Develop a web-based dashboard for visualizing compliance trends across DUT versions

References

- ITSAR (Indian Telecom Security Assurance Requirements) Document, Department of Telecommunications, Government of India
- RFC 4251 - The Secure Shell (SSH) Protocol Architecture
- RFC 3411 - Simple Network Management Protocol (SNMPv3)
- RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3
- gNMI Specification - OpenConfig gRPC Network Management Interface
- Selenium WebDriver Documentation - `https://www.selenium.dev/documentation/`
- Wireshark Developer's Guide - `https://www.wireshark.org/docs/`

A Appendix A: Sample .env Configuration

```
# DUT SSH credentials
SSH_USER=admin
DUT_IP=10.99.214.214
SSH_PASSWORD=SecurePass123

# SNMPv3 credentials
SNMP_USER=snmpuser
SNMP_AUTH_PASS=authpass456
SNMP_PRIV_PASS=privpass789
SNMP_COMMUNITY=public

# Web interface credentials
WEB_LOGIN_URL=https://10.99.214.214
WEB_USERNAME=admin
WEB_PASSWORD=SecurePass123

# Network interface used for packet capture
INTERFACE=eth0
```

B Appendix B: Sample DUT Profile (YAML)

```
ssh:
  base: ssh
  target: "{user}@{ip}"
  connect_options: []
  password_prompt:
    - "password"
    - "Password"
  success_prompt:
    - "$"
    - "#"
    - ">"
  failure_prompt:
    - "Permission denied"
    - "Connection refused"

snmp:
  target: "10.99.214.214"
  user: "snmpuser"
  auth_pass: "authpass456"
  priv_pass: "privpass789"
  community: "public"

web:
  login_url: "https://10.99.214.214"
  username_field: "#username"
  password_field: "#password"
  submit_selector: "#login-btn"
```

```
grpc:
  port: 50051
  tls_cert: "certs/client.crt"
  tls_key:  "certs/client.key"
  ca_cert:  "certs/ca.crt"
```

C Appendix C: Glossary

- **DUT** - Device Under Test
- **ICAF** - ITSAR Compliance Automation Framework
- **ITSAR** - Indian Telecom Security Assurance Requirements
- **OAM** - Operations, Administration, and Maintenance
- **PCAP** - Packet Capture (file format)
- **gNMI** - gRPC Network Management Interface
- **TLS** - Transport Layer Security
- **SSH** - Secure Shell
- **SNMP** - Simple Network Management Protocol

ICAF

ITSAR Compliance Automation Framework

Test Case Authoring Manual

A Developer's Guide to Writing Compliance Test Cases

Version 1.0 | For Internal Use

Author: **Abimanyu A**
Under the Guidance of
Sh. Sumit Singh ADG (Security Certification-I)

1. Overview

ICAF, the **ITSAR Compliance Automation Framework** is a Python-based test automation platform purpose-built for verifying that a Device Under Test (DUT) meets the security and protocol requirements defined in the ITSAR (IT Security Assurance Requirements) standard.

Rather than manually executing dozens of protocol checks per device, ICAF orchestrates an end-to-end compliance run: it logs into the DUT over SSH, exercises protocol stacks (SNMP, SSH, TLS/HTTPS, gRPC/gNMI), captures live network traffic via Wireshark/tshark, drives a real browser for web-based checks, and produces a structured evidence report - all automatically.

1.1 What ICAF Does

At a high level, a single ICAF run performs the following sequence:

1. **Loads a DUT profile** - a YAML or XLSX file that describes how to communicate with the target device (SSH options, SNMP credentials, web selectors, platform-specific commands).
2. **Connects to the DUT** - opens a terminal session via SSH and optionally a browser session via Selenium/Firefox.
3. **Runs selected Clauses** - each Clause maps to a section of the ITSAR standard. A Clause contains one or more Test Cases, and each Test Case is a sequence of Steps.
4. **Captures evidence** - screenshots, PCAP files, and terminal output are saved automatically for every significant action.
5. **Generates a report** - a final pass/fail report is produced, referencing all collected evidence.

1.2 Key Concepts Glossary

`Engine`	The top-level orchestrator. Initialises the runtime, loads the profile, and fires the ClauseRunner.
`Clause`	A group of Test Cases corresponding to one ITSAR requirement clause (e.g. 1.1.1, 1.6.1). Registered in clauses/registry.py.
`TestCase`	A single verification scenario (e.g. "SNMPv3 positive auth"). Extends core/testcase.py. Holds a list of Steps and a PASS/FAIL status.
`Step`	The atomic unit of execution (e.g. run a command, take a screenshot, start a PCAP). Extends core/step.py.
`StepRunner`	A helper that executes a list of Steps in sequence against the runtime context.
`RuntimeContext`	A shared object passed to every Step and TestCase, carrying SSH credentials, browser handle, terminal manager, profile, and mutable state.
`Profile`	A YAML (or XLSX) file that describes device-specific parameters - SSH base, connection options, password prompts, SNMP commands, TLS test commands, etc.

Evidence	Files collected during a run: terminal screenshots, PCAP captures, Wireshark screenshots, browser screenshots. Stored under output/runs/<timestamp>-<clause>/.

1.3 Directory Layout

Project Structure

```
icaf/
  cli/
    main.py                # Entry point - CLI commands
  core/
    clause.py              # BaseClause - base class for all clauses
    testcase.py            # TestCase   - base class for all test cases
    step.py                # Step       - abstract base class for all steps
    step_runner.py         # StepRunner - executes a step list
    clause_runner.py       # ClauseRunner - looks up & runs a clause
    engine.py              # Engine - top-level orchestration
  clauses/
    registry.py            # Maps clause IDs to Clause classes
    clause_1_1_1/
      clause.py            # Clause_1_1_1 - instantiates its test cases
      tc1_snmp_v3_positive.py
      tc3_ssh_mutual_auth.py
      ...
    clause_1_6_1/
      clause.py
  profile/
    default.yaml           # Template profile
    alpine.yaml            # Alpine Linux DUT profile
    <your_device>.yaml     # Your custom profile
  steps/                   # All built-in step classes
  evidence/
    manager.py             # Manages run output directories
```

1.4 How to Run ICAF

To launch a compliance check, use the CLI from the project root:

CLI Usage

```
# Run a specific clause
python -m icaf run --clause 1.1.1 --profile alpine

# Run a section of clauses
python -m icaf run --section 1.1 --profile default
```

```
# Run everything
python -m icaf run --profile alpine

# Create a new DUT profile interactively
python -m icaf profile create
```

The CLI prompts you interactively for SSH credentials, SNMP credentials, and web login details before starting the engine.

2. Core Architecture

Understanding the three-layer hierarchy is essential before writing any test case. Every piece of test logic lives inside one of three containers: a Clause, a TestCase, or a Step.

2.1 The Three-Layer Hierarchy

Hierarchy

```
Clause          (e.g. Clause_1_1_1)
  |
  ├─ TestCase   (e.g. TC3_SSH_MUTUAL_AUTH)
  |    |
  |    ├─ Step  CommandStep("tester", "ssh user@host")
  |    ├─ Step  ExpectOneOfStep("tester", ["password:"])
  |    ├─ Step  InputStep("tester", context.ssh password)
  |    ├─ Step  ScreenshotStep("tester")
  |    └─ Step  PcapStopStep()
  |
  └─ TestCase   (e.g. TC1_SNMPV3_POSITIVE)
       └─ ...
```

> **Rule:** Each layer has one responsibility. A Clause groups related tests. A TestCase implements one verification scenario. A Step executes one atomic action. Keep them clean.

2.2 RuntimeContext

The context object is passed to every step and every test case. It carries all runtime state for the current execution. You never instantiate it directly - the Engine creates it.

context.dut_ip	IP address of the DUT (provided by the user at CLI startup)
context.ssh_user	SSH username
context.ssh_password	SSH password
context.snmp_user	SNMPv3 username
context.snmp_auth_pass	SNMPv3 authentication password
context.snmp_priv_pass	SNMPv3 privacy password
context.web_login_url	Full URL for the DUT web interface login page
context.web_username	Web UI username
context.web_password	Web UI password
context.profile	Loaded ProfileLoader instance - use context.profile.get(key) to read profile values

context.terminal_manager	TerminalManager instance - manages 'tester' and 'dut' terminal sessions
context.browser	BrowserManager (Selenium/Firefox) instance
context.dut_name	Hostname of the DUT (auto-collected at startup)
context.dut_version	OS version string of the DUT
context.execution_id	Unique ID for the current run (used in output paths)

2.3 BaseClause - icaf/core/clause.py

Every clause inherits from BaseClause. Its only jobs are to hold a list of TestCases and iterate over them when run() is called.

core/clause.py

```
class BaseClause:
    def __init__(self, context):
        self.context = context
        self.testcases = []

    def add_testcase(self, tc):
        self.testcases.append(tc)

    def run(self):
        results = []
        for tc in self.testcases:
            self.context.current_testcase = tc
            result = tc.run(self.context)
            results.append(result)
            self.context.current_testcase = None
        return results
```

When you create a new Clause, you subclass `BaseClause`, call `super().__init__(context)`, and register your TestCase instances with `self.add_testcase(...)` in the constructor. The `run()` loop is inherited - you almost never override it.

2.4 TestCase - icaf/core/testcase.py

A TestCase represents one verification scenario. It carries a name, a description, an evidence list, and a PASS/FAIL status.

core/testcase.py

```
class TestCase:
    def __init__(self, name, description):
        self.name = name
        self.description = description
```

```
        self.steps = []
        self.evidence = []
        self.status = "NOT_RUN"

    def add_evidence(self, command=None, output=None, screenshot=None):
        self.evidence.append({
            "command": command,
            "output": output,
            "screenshot": screenshot,
        })

    def pass_test(self):  self.status = "PASS"
    def fail_test(self):  self.status = "FAIL"

    def run(self, context):
        context.current_testcase = self
        try:
            for step in self.steps:
                result = step.execute(context)
            self.pass_test()
        except Exception as e:
            self.fail_test()
        return self
```

> ⚠ **Important:** Existing test cases override the run() method directly rather than using self.steps. This is intentional - it gives you full control over branching logic, early exits, and sub-phase helpers. Follow this same pattern in your own test cases.

At the end of your `run()` method, always call either `self.pass_test()` or `self.fail_test()`, then `return self`. The ClauseRunner depends on this to collect results.

2.5 Step - icaf/core/step.py

Steps are the atomic building blocks. Each Step has a name and an execute(context) method. The StepRunner simply iterates a list of steps and calls execute() on each.

core/step.py

```
class Step:
    def __init__(self, name):
        self.name = name

    def execute(self, context):
        raise NotImplementedError   # Each concrete step implements this
```

You do not create new Step subclasses when writing test cases - you compose existing ones. The built-in step library covers all common actions.

2.6 StepRunner

`StepRunner` is a thin wrapper that executes a list of steps in order. It logs each step name and calls `execute(context)`. Use it to group logically related steps into a mini-pipeline:

StepRunner usage

```
StepRunner([
    PcapStartStep(interface="eth0", filename="tc1_check.pcapng"),
    CommandStep("tester", "openssl s_client -connect 10.0.0.1:443 -tls1"),
]).run(context)
```

2.7 Profile System

The Profile is a YAML file in icaf/profile/ that controls all device-specific behaviour. Profile values are retrieved in test cases using:

Reading profile values

```
# Single value with fallback
value = context.profile.get("ssh.base", "ssh")

# List value (YAML sequence or comma-separated string)
prompts = context.profile.get_list("ssh.password_prompt")
```

Below is an annotated excerpt from the alpine.yaml profile to illustrate the structure:

icaf/profile/alpine.yaml (excerpt)

```
ssh:
  base: ssh                          # SSH executable name
  target: "{user}@{ip}"               # Target template (interpolated at
runtime)
  connect_options:                     # Extra flags passed to ssh
  password_prompt:
    - password                         # String to wait for after connecting
  success_prompt:
    - $
    - "#"
    - ">"
  failure_prompt:
    - Permission denied
    - connection refused

tls:
```

```
  tls10_test_command: openssl s_client -connect {ip}:443 -tls1
  tls11_test_command: openssl s_client -connect {ip}:443 -tls1_1
  failure_indicators:
    - error
    - protocol version
  success_indicator:
    - Cipher

snmp:
  v1_command: snmpwalk -v1 -c public {ip}
  v3_valid_command: >-
    snmpget -v3 -u {user} -l authPriv -a SHA -A {auth_pass}
    -x AES -X {priv_pass} {ip} 1.3.6.1.2.1.1.3.0
  success_indicators:
    - Timeticks
  failure_indicators:
    - Timeout
    - authenticationFailure
```

Design principle: All device-specific strings (commands, prompts, selectors) belong in the profile. Test case Python code should only contain logic - never hard-coded hostnames, credential strings, or CLI syntax.

3. Built-in Step Reference

The following step classes are available in icaf/steps/. Import and compose them inside your TestCase.run() method.

3.1 Terminal & Command Steps

Step Class	Purpose	Key Parameters
`CommandStep`	Send a shell command to a named terminal	`terminal_name, command, settle_time=0`
`InputStep`	Type raw text into a terminal (no newline logic)	`terminal_name, text`
`ExpectOneOfStep`	Block until one of the given patterns appears	`terminal_name, patterns: list, timeout=10`
`SessionResetStep`	Close/reset a terminal session cleanly	`terminal_name, post_reset_delay=2`
`ClearTerminalStep`	Clear the terminal scrollback buffer	`terminal_name`
`ScreenshotStep`	Capture a screenshot of the named terminal	`terminal_name, caption`

3.2 Network Capture Steps

Step Class	Purpose	Key Parameters
`PcapStartStep`	Start a tshark/tcpdump packet capture	`interface, filename`
`PcapStopStep`	Stop the active packet capture	`(no params)`
`AnalyzePcapStep`	Filter captured PCAP with a Wireshark display filter	`display_filter: str`
`WiresharkPacketScreenshotStep`	Open filtered packets in Wireshark and screenshot	`protocol: str, caption: str`

3.3 Browser Steps

4	Purpose	Key Parameters
`OpenURLStep`	Navigate the browser to a URL	`url: str`
`FillInputStep`	Locate a field by CSS selector and type a value	`selector, value`
`ClickStep`	Click an element by CSS selector	`selector`

`WaitStep`	Pause execution for N seconds	`seconds: int`
`BrowserScreenshotStep`	Save a screenshot of the current browser page	`filename: str, caption: str`
`VerifyOutputStep`	Check browser page source for expected strings	`source, patterns, should_exist=True`

3.4 Typical Step Patterns

The following patterns appear repeatedly across all existing test cases. Memorise them - they are the building blocks of any new test case.

Pattern A - SSH Login

SSH Login

```
StepRunner([CommandStep("tester", ssh_cmd, settle_time=4)]).run(context)
ExpectOneOfStep("tester", ["password:", "Password:"],
timeout=10).execute(context)
StepRunner([InputStep("tester", context.ssh_password)]).run(context)
ExpectOneOfStep("tester", ["#", "$", ">"], timeout=10).execute(context)
```

Pattern B - Command + PCAP capture

Command + PCAP

```
StepRunner([
    PcapStartStep(interface="eth0", filename="tc_check.pcapng"),
    CommandStep("tester", cmd, settle_time=5),
]).run(context)

pattern, _ = ExpectOneOfStep("tester", success_patterns + fail_patterns,
timeout=15).execute(context)

StepRunner([PcapStopStep()]).run(context)
ScreenshotStep("tester").execute(context)
```

Pattern C - Decision branching on matched pattern

Branching on pattern

```
if any(f in pattern for f in fail_patterns):
    logger.error("TC_X: check failed - %s", pattern)
    StepRunner([ClearTerminalStep("tester")]).run(context)
    return False

logger.info("TC_X: check passed")
StepRunner([
    AnalyzePcapStep("snmp"),
    WiresharkPacketScreenshotStep("snmp"),
]).run(context)
return True
```

Pattern D - Browser login

Browser login flow

```
StepRunner([
    PcapStartStep(interface="eth0", filename="tc_https.pcapng"),
    OpenURLStep(login_url),
    WaitStep(2),
]).run(context)

BrowserScreenshotStep("tc_login_page.png").execute(context)

StepRunner([
    FillInputStep(user_selector, username),
    FillInputStep(pass_selector, password),
    ClickStep(submit_selector),
    WaitStep(3),
]).run(context)

BrowserScreenshotStep("tc_after_login.png").execute(context)
StepRunner([PcapStopStep()]).run(context)

ok = VerifyOutputStep("browser", ["Dashboard", "logout"],
should_exist=True).execute(context)
```

4. ScreenshotStep - When and How to Use It

Screenshots are the primary evidence artefact in ICAF. They prove to an auditor that the test actually ran and produced the shown result on the DUT. Using them correctly is therefore critical.

4.1 Terminal Screenshot

`ScreenshotStep(terminal_name)` captures the current visual state of the named terminal window (e.g. `"tester"`). It should be called immediately after a meaningful event so the output is still on screen.

Terminal ScreenshotStep

```
from icaf.steps.screenshot_step import ScreenshotStep

# After running a command and reading the result:
StepRunner([CommandStep("tester", "openssl version",
settle_time=1)]).run(context)
ExpectOneOfStep("tester", ["#", "$"], timeout=8).execute(context)
ScreenshotStep("tester").execute(context)          # Capture while output is
visible
```

> ⚠ **Timing matters:** Always call ScreenshotStep BEFORE calling ClearTerminalStep or SessionResetStep. Once the terminal is cleared or the session is reset, the evidence is gone.

4.2 Browser Screenshot

`BrowserScreenshotStep(filename)` saves a full-page screenshot of the Firefox browser window at the given moment. Use a descriptive filename so it is identifiable in the report.

BrowserScreenshotStep

```
from icaf.steps.browser_screenshot_step import BrowserScreenshotStep

# Capture the login page before submitting credentials:
BrowserScreenshotStep("tc_tls_login_page.png").execute(context)

# Capture the result page after the action:
BrowserScreenshotStep("tc_tls_after_submit.png").execute(context)
```

4.3 Wireshark Packet Screenshot

`WiresharkPacketScreenshotStep(protocol)` is called after `AnalyzePcapStep`. It opens the filtered PCAP in Wireshark, selects the first matching packet, and saves a screenshot of the packet detail pane.

PCAP + Wireshark screenshot

```
from icaf.steps.analyze_pcap_step import AnalyzePcapStep
from icaf.steps.wireshark_packet_screenshot_step import
WiresharkPacketScreenshotStep


StepRunner([
    AnalyzePcapStep("tls"),                      # Filter by protocol/display
filter
    WiresharkPacketScreenshotStep("tls"),         # Screenshot the first TLS
packet
]).run(context)
```

4.4 Screenshot Checklist

Apply this checklist when deciding where to place screenshot steps in a new test case:

- **After every SSH command** that produces a result relevant to the test assertion.
- **After every negative case** - wrong credentials, connection refused, protocol version error.
- **After the browser reaches the login page** - before submitting credentials.
- **After the browser completes the action** - after submitting credentials / navigating.
- **After every PCAP analysis** - use WiresharkPacketScreenshotStep.
- **Before any SessionResetStep or ClearTerminalStep** - once the session ends, evidence is lost.

5. Writing a New Test Case - Step by Step

Follow this process every time you add a new test case. Do not skip steps.

Step 1 - Define the test scenario

Before writing any code, answer these questions in plain English:

1. **What protocol or feature** is being verified?
2. **What is the positive case** - what should happen when everything is correct?
3. **What is the negative case** - what should be rejected or blocked?
4. **What evidence** will prove the result to an auditor (output, PCAP, browser screenshot)?
5. **What profile keys** are needed (commands, prompts, selectors)? Do they exist, or must you add them?

Step 2 - Create the test case file

Create a new file in the appropriate clause directory, following the naming convention `tc<N>_<short_description>.py`. For example: `icaf/clauses/clause_1_1_1/tc9_tls_version_check.py`.

New test case skeleton

```
"""
TC9 - TLS Version Check: verify TLS 1.0 and 1.1 are disabled.
Test Scenario 1.1.1.9
"""

from icaf.core.testcase import TestCase
from icaf.core.step_runner import StepRunner
from icaf.steps.command_step import CommandStep
from icaf.steps.expect_one_of_step import ExpectOneOfStep
from icaf.steps.screenshot_step import ScreenshotStep
from icaf.steps.pcap_start_step import PcapStartStep
from icaf.steps.pcap_stop_step import PcapStopStep
from icaf.steps.analyze_pcap_step import AnalyzePcapStep
from icaf.steps.wireshark_packet_screenshot_step import
WiresharkPacketScreenshotStep
from icaf.steps.session_reset_step import SessionResetStep
from icaf.steps.clear_terminal_step import ClearTerminalStep
from icaf.utils.logger import logger


class TC9TLSVersionCheck(TestCase):

    def __init__(self):
```

```
        super().__init__(
            "TC9_TLS_VERSION_CHECK",
            "Verify that TLS 1.0 and TLS 1.1 are disabled on the DUT",
        )

    def run(self, context):
        # ... implementation ...
        self.pass_test()   # or self.fail_test()
        return self
```

Step 3 - Implement sub-phase helpers

Break the logic into small private methods (prefixed with _). Each method handles one phase of the test (e.g. `_check_tls10_disabled`, `_check_tls11_disabled`). This keeps `run()` readable and makes it easy to skip or reorder phases later.

Step 4 - Register the test case in the Clause

Open the `clause.py` file for your target clause and add an import and a call to `self.add_testcase(...)`:

Registering in clause.py

```
# icaf/clauses/clause_1_1_1/clause.py

from .tc9_tls_version_check import TC9TLSVersionCheck   # <-- add this import

class Clause_1_1_1(BaseClause):
    def __init__(self, context):
        super().__init__(context)
        self.add_testcase(TC1SNMPv3Positive())
        self.add_testcase(TC3SSHMutualAuth())
        self.add_testcase(TC9TLSVersionCheck())          # <-- add this line
```

Step 5 - Add required profile keys

If your test case reads profile values that do not yet exist in the profile, add them to the relevant YAML file under icaf/profile/. Use a logical dotted key path:

Adding profile keys

```
# icaf/profile/alpine.yaml

tls:
  tls10_test_command: openssl s_client -connect {ip}:443 -tls1
  tls11_test_command: openssl s_client -connect {ip}:443 -tls1_1
  failure_indicators:
    - error
```

```
    - protocol version
  success_indicator:
    - Cipher
```

Step 6 - Test and verify

6. Run the clause in isolation against a test DUT.
7. `python -m icaf run --clause 1.1.1 --profile alpine`
8. Check the terminal output for your TC name - it should appear and show PASS or FAIL.
9. Check the evidence directory:
   `output/runs/<timestamp>-1.1.1/<clause>/<testcase>/screenshots/`
10. Confirm screenshots were captured at all the right moments.
11. If FAIL - check the logger.error() messages to diagnose the problem.

6. Example 1 - Verify TLS 1.0 is Disabled

6.1 Scenario Description

This test verifies that the DUT refuses any TLS handshake using the deprecated TLS 1.0 protocol. An attacker who can only negotiate TLS 1.0 should receive a connection error, not a valid cipher suite.

Test Case ID	TC9_TLS10_DISABLED
Objective	Confirm the DUT rejects TLS 1.0 connection attempts
Positive case	openssl s_client -tls1 returns an error / protocol version message - no cipher established
Negative case	(N/A for this check - any successful handshake is the failure)
Evidence	Terminal screenshot showing the openssl error; PCAP proving no full TLS 1.0 handshake completed
Profile keys	tls.tls10_test_command, tls.failure_indicators

6.2 Profile Additions

Add the following to your DUT profile YAML before writing the test case:

Profile - tls section

```yaml
# icaf/profile/alpine.yaml

tls:
  tls10_test_command: openssl s_client -connect {ip}:443 -tls1
  failure_indicators:
    - error
    - no protocols available
    - protocol version
    - Connection refused
  success_indicator:
    - Cipher is
```

6.3 Full Test Case Implementation

tc9_tls10_disabled.py - full implementation

```python
"""
TC9 - TLS 1.0 Disabled: confirm the DUT rejects TLS 1.0 handshakes.
Test Scenario 1.1.1.9
"""

from icaf.core.testcase import TestCase
```

```
from icaf.core.step_runner import StepRunner
from icaf.steps.command_step import CommandStep
from icaf.steps.expect_one_of_step import ExpectOneOfStep
from icaf.steps.screenshot_step import ScreenshotStep
from icaf.steps.pcap_start_step import PcapStartStep
from icaf.steps.pcap_stop_step import PcapStopStep
from icaf.steps.analyze_pcap_step import AnalyzePcapStep
from icaf.steps.wireshark_packet_screenshot_step import
WiresharkPacketScreenshotStep
from icaf.steps.clear_terminal_step import ClearTerminalStep
from icaf.utils.logger import logger


class TC9TLS10Disabled(TestCase):

    def __init__(self):
        super().__init__(
            "TC9_TLS10_DISABLED",
            "Verify that the DUT refuses TLS 1.0 handshakes",
        )

    # ─── Phase 1: attempt TLS 1.0 handshake ──────────────────────────

    def _attempt_tls10(self, context):
        """
        Build the test command from the profile, start a PCAP, run the
        command, and capture a screenshot.  Returns True if the DUT
        correctly rejected the connection (failure indicator seen).
        """
        # Read the test command template from the profile and inject the DUT IP
        cmd_template = context.profile.get(
            "tls.tls10_test_command",
            "openssl s_client -connect {ip}:443 -tls1",
        )
        cmd = cmd_template.format(ip=context.dut_ip)

        fail_indicators = context.profile.get_list("tls.failure_indicators")
        ok_indicator    = context.profile.get("tls.success_indicator", "Cipher
is")

        # Start network capture, then run the openssl test command
        StepRunner([
            PcapStartStep(interface="eth0", filename="tc9_tls10.pcapng"),
            CommandStep("tester", cmd, settle_time=6),
        ]).run(context)
```

```
        # Wait for the terminal to settle - any of these patterns ends the wait
        pattern, _ = ExpectOneOfStep(
            "tester",
            fail_indicators + [ok_indicator],
            timeout=15,
        ).execute(context)

        # Stop the capture and take evidence
        StepRunner([PcapStopStep()]).run(context)
        ScreenshotStep("tester").execute(context)          # Evidence: terminal
output

        # Evaluate the result
        if any(f.lower() in pattern.lower() for f in fail_indicators):
            # Good - the DUT refused the TLS 1.0 handshake
            logger.info("TC9: TLS 1.0 correctly rejected - '%s'", pattern)
            # Capture the PCAP evidence: filter to TLS alerts/handshakes on
port 443
            StepRunner([
                AnalyzePcapStep("tcp.port == 443 && tls"),
                WiresharkPacketScreenshotStep("tls"),
            ]).run(context)
            StepRunner([ClearTerminalStep("tester")]).run(context)
            return True

        # Bad - openssl established a session (TLS 1.0 is NOT disabled)
        logger.error("TC9: TLS 1.0 handshake SUCCEEDED - DUT must be hardened")
        StepRunner([ClearTerminalStep("tester")]).run(context)
        return False

    # ── Entry point ──────────────────────────────────────────────────

    def run(self, context):
        tls10_ok = self._attempt_tls10(context)

        self.pass_test() if tls10_ok else self.fail_test()
        return self
```

6.4 Register in Clause

clause.py - registering TC9

```
# icaf/clauses/clause_1_1_1/clause.py

from .tc9_tls10_disabled import TC9TLS10Disabled

class Clause_1_1_1(BaseClause):
```

```
    def __init__(self, context):
        super().__init__(context)
        self.add_testcase(TC1SNMPv3Positive())
        self.add_testcase(TC3SSHMutualAuth())
        self.add_testcase(TC9TLS10Disabled())   # <-- registered here
```

6.5 Expected Output

When the test runs correctly against a hardened DUT, you will see the following in the ICAF terminal:

Expected log output

```
[INFO]  Running step: PcapStartStep
[INFO]  Running step: CommandStep(tester, openssl s_client -connect
10.0.0.1:443 -tls1)
[INFO]  Running step: ExpectOneOfStep
[INFO]  Running step: PcapStopStep
[INFO]  Running step: ScreenshotStep(tester)
[INFO]  TC9: TLS 1.0 correctly rejected - 'no protocols available'
[INFO]  Running step: AnalyzePcapStep(tcp.port == 443 && tls)
[INFO]  Running step: WiresharkPacketScreenshotStep(tls)
[INFO]  TC9_TLS10_DISABLED → PASS
```

7. Example 2 - Verify SSH Root Login is Disabled

7.1 Scenario Description

This test verifies that the DUT's SSH server refuses direct root login. It covers both the configuration side (reading sshd_config) and the protocol side (actually attempting a root SSH login and confirming it is rejected).

Test Case ID	TC10_SSH_ROOT_LOGIN_DISABLED
Objective	Confirm PermitRootLogin is set to 'no' in sshd_config, and confirm a root login attempt is rejected
Positive case	grep output shows 'PermitRootLogin no' (or 'without-password')
Negative case	An SSH attempt as root with the correct password is rejected
Evidence	Two terminal screenshots: one showing the grep output, one showing the root login rejection; PCAP of the negative SSH attempt
Profile keys	ssh.root_login_check_command, ssh.root_username, ssh.failure_prompt

7.2 Profile Additions

Profile - ssh section additions

```
# icaf/profile/alpine.yaml

ssh:
  # ... existing keys ...

  # New keys for TC10
  root_login_check_command: grep -i PermitRootLogin /etc/ssh/sshd_config
  root_username: root
  root_login_permitted_indicators:
    - PermitRootLogin yes
    - PermitRootLogin Yes
  root_login_denied_indicators:
    - PermitRootLogin no
    - PermitRootLogin prohibit-password
    - PermitRootLogin without-password
```

7.3 Full Test Case Implementation

tc10_ssh_root_login_disabled.py - full implementation

```
"""
TC10 - SSH Root Login Disabled: verify the DUT refuses direct root SSH access.
Test Scenario 1.1.1.10
```

```
"""

from icaf.core.testcase import TestCase
from icaf.core.step_runner import StepRunner
from icaf.steps.command_step import CommandStep
from icaf.steps.expect_one_of_step import ExpectOneOfStep
from icaf.steps.input_step import InputStep
from icaf.steps.screenshot_step import ScreenshotStep
from icaf.steps.pcap_start_step import PcapStartStep
from icaf.steps.pcap_stop_step import PcapStopStep
from icaf.steps.analyze_pcap_step import AnalyzePcapStep
from icaf.steps.wireshark_packet_screenshot_step import
WiresharkPacketScreenshotStep
from icaf.steps.session_reset_step import SessionResetStep
from icaf.steps.clear_terminal_step import ClearTerminalStep
from icaf.utils.logger import logger


class TC10SSHRootLoginDisabled(TestCase):

    def __init__(self):
        super().__init__(
            "TC10_SSH_ROOT_LOGIN_DISABLED",
            "Verify that direct root SSH login is disabled on the DUT",
        )

    # ─── Phase 1: check configuration file ──────────────────────────────

    def _check_sshd_config(self, context):
        """
        SSH into the DUT as the normal admin user and grep sshd_config
        to confirm PermitRootLogin is set to a denied value.
        """
        base  = context.profile.get("ssh.base", "ssh")
        options = context.profile.get_list("ssh.connect_options")
        target  = context.profile.get("ssh.target", "{user}@{ip}").format(
            user=context.ssh_user, ip=context.dut_ip,
        )
        ssh_cmd = " ".join([base] + options + [target])

        check_cmd    = context.profile.get(
            "ssh.root_login_check_command",
            "grep -i PermitRootLogin /etc/ssh/sshd_config",
        )
        denied_list  =
context.profile.get_list("ssh.root_login_denied_indicators")
```

```
        allowed_list =
context.profile.get_list("ssh.root_login_permitted_indicators")

        # Log in as the normal admin user
        StepRunner([CommandStep("tester", ssh_cmd,
settle_time=4)]).run(context)
        ExpectOneOfStep(
            "tester", context.profile.get_list("ssh.password_prompt"),
timeout=10
        ).execute(context)
        StepRunner([InputStep("tester", context.ssh_password)]).run(context)
        ExpectOneOfStep("tester", ["#", "$", ">"], timeout=10).execute(context)

        # Run the grep command on sshd_config
        StepRunner([CommandStep("tester", check_cmd,
settle_time=2)]).run(context)
        pattern,  _ = ExpectOneOfStep(
            "tester",
            denied_list + allowed_list + ["#", "$"],
            timeout=10,
        ).execute(context)

        # Capture the terminal - shows the grep output
        ScreenshotStep("tester").execute(context)

        # Close the admin session before the next phase
        StepRunner([SessionResetStep("tester",
post_reset_delay=2)]).run(context)

        # Evaluate
        config_ok = any(d.lower() in pattern.lower() for d in denied_list)
        if config_ok:
            logger.info("TC10 Config: PermitRootLogin correctly restricted -
'%s'", pattern)
        else:
            logger.error("TC10 Config: PermitRootLogin is set to '%s' - FAIL",
pattern)

        return config_ok

    # ── Phase 2: attempt root login (negative test) ──────────────────

    def _attempt_root_login(self, context):
        """
        Attempt to SSH as root using the known admin password.
        Expect rejection.  The session is captured in a PCAP.
        """
```

```
        base       = context.profile.get("ssh.base", "ssh")
        options      = context.profile.get_list("ssh.connect_options")
        root_user    = context.profile.get("ssh.root_username", "root")
        root_target  = f"{root_user}@{context.dut_ip}"
        root_cmd     = " ".join([base] + options + [root_target])

        pass_prompts = context.profile.get_list("ssh.password_prompt")
        fail_prompts = context.profile.get_list("ssh.failure_prompt") + [
            "Permission denied",
            "not allowed",
            "Connection closed",
            "Disconnected",
        ]
        success_prompts = ["#", "$", ">"]

        # Start PCAP to capture the SSH exchange
        StepRunner([
            PcapStartStep(interface="eth0",
filename="tc10_root_login_attempt.pcapng"),
            CommandStep("tester", root_cmd, settle_time=4),
        ]).run(context)

        pattern, _ = ExpectOneOfStep(
            "tester",
            pass_prompts + fail_prompts + success_prompts,
            timeout=12,
        ).execute(context)

        # If a password prompt appeared, send the credential and wait again
        if pattern in pass_prompts:
            StepRunner([InputStep("tester",
context.ssh_password)]).run(context)
            pattern, _ = ExpectOneOfStep(
                "tester",
                fail_prompts + success_prompts,
                timeout=12,
            ).execute(context)

        StepRunner([PcapStopStep()]).run(context)
        ScreenshotStep("tester").execute(context)       # Evidence: rejection
message

        rejected = any(f.lower() in pattern.lower() for f in fail_prompts)

        if rejected:
```

```
            logger.info("TC10 Negative: root login correctly rejected - '%s'",
pattern)
            StepRunner([
                AnalyzePcapStep("ssh"),
                WiresharkPacketScreenshotStep("ssh"),
            ]).run(context)
            StepRunner([ClearTerminalStep("tester")]).run(context)
            return True

        logger.error("TC10 Negative: root login SUCCEEDED - DUT is
misconfigured")
        StepRunner([SessionResetStep("tester",
post_reset_delay=2)]).run(context)
        return False

    # ── Entry point ──────────────────────────────────────────────

    def run(self, context):
        config_ok = self._check_sshd_config(context)
        login_ok  = self._attempt_root_login(context)

        # Both phases must pass for the test to pass
        self.pass_test() if (config_ok and login_ok) else self.fail_test()
        return self
```

7.4 Register in Clause

clause.py - registering TC10

```
# icaf/clauses/clause_1_1_1/clause.py

from .tc10_ssh_root_login_disabled import TC10SSHRootLoginDisabled

class Clause_1_1_1(BaseClause):
    def __init__(self, context):
        super().__init__(context)
        self.add_testcase(TC1SNMPv3Positive())
        self.add_testcase(TC3SSHMutualAuth())
        self.add_testcase(TC9TLS10Disabled())
        self.add_testcase(TC10SSHRootLoginDisabled())    # <-- registered here
```

7.5 Expected Output

A passing run against a correctly hardened DUT:

Expected log output

```
[INFO]  Running step: CommandStep(tester, ssh admin@10.0.0.1)
```

```
[INFO]  Running step: ExpectOneOfStep
[INFO]  Running step: InputStep(tester, ***)
[INFO]  Running step: CommandStep(tester, grep -i PermitRootLogin
/etc/ssh/sshd_config)
[INFO]  Running step: ScreenshotStep(tester)
[INFO]  TC10 Config: PermitRootLogin correctly restricted - 'PermitRootLogin
no'
[INFO]  Running step: SessionResetStep(tester)
[INFO]  Running step: PcapStartStep
[INFO]  Running step: CommandStep(tester, ssh root@10.0.0.1)
[INFO]  Running step: InputStep(tester, ***)
[INFO]  Running step: PcapStopStep
[INFO]  Running step: ScreenshotStep(tester)
[INFO]  TC10 Negative: root login correctly rejected - 'Permission denied'
[INFO]  Running step: AnalyzePcapStep(ssh)
[INFO]  Running step: WiresharkPacketScreenshotStep(ssh)
[INFO]  TC10_SSH_ROOT_LOGIN_DISABLED → PASS
```

8. Quick Reference

8.1 New Test Case Checklist

1. **File created** in the correct clause directory with naming convention tc<N>_<description>.py
2. **Class header** - docstring, class name, super().__init__(name, description)
3. **Sub-phase helpers** - private methods prefixed with _; each handles one logical phase
4. **Profile keys** - all device-specific strings read from context.profile.get() / get_list()
5. **PCAP start/stop** - PcapStartStep before and PcapStopStep after every protocol exchange
6. **Screenshots** - ScreenshotStep taken before any ClearTerminalStep or SessionResetStep
7. **Wireshark evidence** - AnalyzePcapStep + WiresharkPacketScreenshotStep on success
8. **Logging** - logger.info() on PASS, logger.error() on FAIL; include the matched pattern in the message
9. **run() returns self** - always ends with self.pass_test() or self.fail_test() then return self
10. **Registered in clause.py** - import and self.add_testcase(...) added
11. **Profile updated** - new keys added to the relevant .yaml file

8.2 Common Mistakes

✕ **Hard-coding credentials or IPs:** Never write context.ssh_password == "admin123" or ip = "192.168.1.1" inside a test case. Always use context.dut_ip, context.ssh_password, and context.profile.get(...).

✕ **Missing return self:** If run() does not return self, the ClauseRunner cannot collect the result and the test appears to never finish.

✕ **Screenshot after session reset:** Calling SessionResetStep before ScreenshotStep means the terminal is blank. Always screenshot first.

✕ **Missing PcapStopStep:** If PcapStopStep is not called, the PCAP file stays open and subsequent captures may fail or the file will be corrupt.

✕ **Not registering in clause.py:** A test case that is not added via self.add_testcase() will never run, regardless of how correct the implementation is.

8.3 SSHMixin - Reuse for SSH Operations

If your test case performs multiple SSH operations, inherit from both `TestCase` and `SSHMixin` (from `icaf/clauses/clause_1_1_1/ssh_mixin.py`) to avoid duplicating SSH boilerplate:

Using SSHMixin

```
from icaf.core.testcase import TestCase
```

```
from icaf.clauses.clause_1_1_1.ssh_mixin import SSHMixin

class TC10SSHRootLoginDisabled(TestCase, SSHMixin):

    def _check_sshd_config(self, context):
        self.ssh_open_session(context)                          # login
        self.ssh_run_commands(context, [check_cmd])             # run command
        ScreenshotStep("tester").execute(context)
        self.ssh_close_session(context)                         # logout
```

For further questions or contributions, refer to the inline docstrings in each file and the existing test case implementations under icaf/clauses/. The existing test cases - particularly tc1_snmp_v3_positive.py and tc3_ssh_mutual_auth.py - are the canonical reference for complex, multi-phase test structure.

TSTL Evaluation Test Report

for

amd64_linux26

Prepared For: National Centre of Communications Security, Bengaluru, Department of Telecom, Ministry of Communications, Government of India

Document No.	Created By	Reviewed By	Approved By
/IP ROUTER/1.1	Test Engineer	Senior Security Reviewer	Approval Authority

Test conducted by	Test conducted on	Test reviewed by	Test reviewed on
Tester	2026-04-09 06:37:52.065200	Reviewer	2026-04-09 06:37:52.065200

Revision History

Version	Date	Changes
V.1.0	Initial Release	NCCS Approved Test Plan with initial Test Cases.
V.1.1	2026-04-09 06:37:52.065200	First Release of Test Report — automated evidence collected.

TSTR for Evaluation of 1.1 Management Protocols Entity Mutual Authentication (1.1.1 of CSR)

Preface

DUT Details: amd64_linux26

DUT IP Address: 10.80.127.211

DUT Software Version:

Software Name	Software Version
Device Firmware / OS	Ubuntu 24.04.3 LTS

Digest Hash of OS:

Software Version	Hash Integrity Value
Ubuntu 24.04.3 LTS	3e5851448bae5b36f351becde037a8b13b77307279f484eda808f8177d9a4293
sshd_config	64325541513d33ea1d2ccd19c77750d458e67e7967fd2e7ef81d92f0aa2ffe21

Applicable ITSAR:

- IP Router-ITSAR201012401 (ITSAR201012401)
- TSDSI STD T1.3GPP 33.117-14.2.0 V.1.0.0 section 4.2.3.4.4.1 (TSDSI-STD-T1.3GPP-33.117)

ITSAR Version No.:

- v.1.0.1 (Date of Release: 03rd January 2024)
- v.1.0.0 (Date of Release: N/A)

1. ITSAR Section No. & Name: 1.1 Access and Authorization

2. Security Requirement No. & Name: 1.1.1 Management Protocols and Mutual Authentication

3. Requirement Description:

The protocols used for the Network Product's management shall support mutual authentication mechanisms. There is mutual authentication of entities for management interfaces on the network product. HTTPS with TLS 1.2, SNMP V3 Protocols are allowed. [Reference: TSDSI STD T1.3GPP 33.117-14.2.0 V.1.0.0. section 4.2.3.4.4.1]

4. DUT Configuration:

1) OAM Access supported by DUT:

Protocol	Supported
HTTP	Yes (Verified)
HTTPS	Yes (Verified)
SNMP	Yes (Verified)
SSH	Yes (Verified)
TELNET	Yes (Not Detected)

Legend:

- Yes (Verified): Protocol configured and detected
- Yes (Not Detected): Configured but not reachable
- No (Unexpected): Running but not documented

NOTE:

5. Pre-conditions:

- Review OEM Documentation that lists all the management protocols and describes the authentication mechanism used for each one. List is to be furnished with the test report.
- Documentation that lists each of the management protocols and describes the mutual authentication mechanism (password, certificate, key pairs/PSK, user-based security models e.g. SNMPv3) used for each one on each port.
- Documentation including the details of the mechanism used for mutual authentication and cipher suites enabled on DUT.
- List of all the management protocols supported by DUT with the details.

6. Test Objective:

- Based on the OEM Documentation regarding DUT Supported Management Protocols, checking of mutual authentication for all such protocols is required. To verify that DUT permits access through supported management protocols only after successful mutual authentication.

7. Test Plan:

- The DUT supports SNMPv3, SSH (password and public key), HTTPS, and gRPC/gNMI for management. All applicable protocols are tested for mutual authentication. Console access is out of scope for automated testing.

a. Number of Test Scenarios:

- Test case 1: Test Scenario 1.1.1 - Positive: Configure and verify SNMPv3 protocol mutual authentication.
- Test case 2: Test Scenario 1.1.1.2 - Negative: Verify SNMPv3 login with incorrect credentials and verify response by DUT.
- Test case 3: Test Scenario 1.1.1.3 - Configure and verify successful and unsuccessful mutual authentication over SSH.
- Test case 4: Test Scenario 1.1.1.4 - Configure and verify SSH login using the correct public key.
- Test case 5: Test Scenario 1.1.1.5 - Configure and verify SSH login using the incorrect public key.
- Test case 6: Test Scenario 1.1.1.6 - Configure and verify successful mutual authentication over HTTPS.
- Test case 7: Test Scenario 1.1.1.7 - Configure and verify unsuccessful mutual authentication over HTTPS.
- Test case 8: Test Scenario 1.1.1.8 - Configure and verify mutual authentication over gRPC/gNMI.

b. Tools Required:

- Kali GNU/Linux Rolling 2025.1
- Wireshark 4.6.0
- snmpget
- Selenium
- tshark
- Wireshark
- Nmap 7.95

8. Expected Results for Pass:

- Test case 1: DUT shall accept SNMP requests only over SNMPv3 when correct username, authentication protocol, and privacy credentials are provided. Weaker algorithms must be rejected.
- Test case 2: The DUT shall reject all SNMPv3 requests made with incorrect authentication or privacy credentials.
- Test case 3: When correct SSH credentials are used, DUT shall establish SSHv2 session with encrypted communication. When incorrect credentials are used, DUT shall deny access without revealing sensitive information.
- Test case 4: The DUT shall allow SSH access only when a correctly configured public key is presented by the client.
- Test case 5: The DUT shall reject SSH login attempts made with an incorrect or untrusted public key.
- Test case 6: The DUT shall successfully establish a secure HTTPS session only when valid credentials are provided.
- Test case 7: The DUT shall deny HTTPS access when incorrect credentials are entered.
- Test case 8: The DUT shall establish a TLS-protected gRPC/gNMI session only when a trusted CA certificate, valid client certificate, and correct user credentials are provided.

9. Expected Format of Evidence:

Screenshots of commands performed on Kali Linux, web browser terminal, and packets captured in Wireshark are to be provided. Automated evidence includes command output and system responses captured directly.

10. Testbed diagram:

10. Test Execution

NOTE: This report covers 8 of 8 defined test cases. 1 case(s) were not applicable (required protocol not present on DUT).

1. Test Case Name: TC1_SNMPV3_POSITIVE

a. Test Case Description:

This test case verifies that SNMPv3 is correctly configured on the DUT and that mutual authentication between the SNMP manager and SNMP agent works as expected. The DUT accepts SNMPv3 communication only when correct authentication credentials (username, SHA auth protocol, AES-128 privacy) are provided. SNMPv1 and SNMPv2c must be disabled by default. Attempts to connect with weaker algorithms (MD5/3DES) must be rejected.

b. Execution Steps:

- Verify SNMPv1 is disabled (expect timeout): snmpget -v1 -c public 10.80.127.211 1.3.6.1.2.1.1.3.0
- Capture SNMPv1 packets in Wireshark; confirm DUT sends no response.
- Verify SNMPv2c is disabled (expect timeout): snmpget -v2c -c public 10.80.127.211 1.3.6.1.2.1.1.3.0
- Capture SNMPv2c packets in Wireshark; confirm DUT sends no response.
- Log in to DUT via SSH: ssh dut@10.80.127.211
- Authenticate: username=dut password=[REDACTED]
- Configure SNMPv3 group: snmp-agent group v3 SNMP_Group privacy
- Configure SNMPv3 user (User1): snmp-agent usm-user v3 User1 SNMP_GROUP simple authentication-mode sha [REDACTED] privacy-mode aes128 [REDACTED]
- Run SNMPv3 GET with valid credentials (expect sysUpTime response): snmpget -v3 -u User1 -l authPriv -a SHA -A [REDACTED] -x AES -X [REDACTED] 10.80.127.211 1.3.6.1.2.1.1.3.0
- Capture SNMPv3 traffic in Wireshark; confirm traffic is encrypted.
- Attempt SNMPv3 GET with weak algorithms MD5/DES (expect rejection): snmpget -v3 -u user2 -l authPriv -a MD5 -A [REDACTED] -x DES -X [REDACTED] 10.80.127.211 1.3.6.1.2.1.1.3.0

c. Evidence Captured:

Command Executed: snmpget -v1 -c public 10.80.127.211 1.3.6.1.2.1.1.3.0

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ snmpget -v1 -c public 10.80.127.211 1.3.6.1.2.1.1.3.0
iso.3.6.1.2.1.1.3.0 = Timeticks: (16439) 0:02:44.39


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$
```

Evidence Screenshot - TC1_SNMPV3_POSITIVE_tester_023806_783701.png

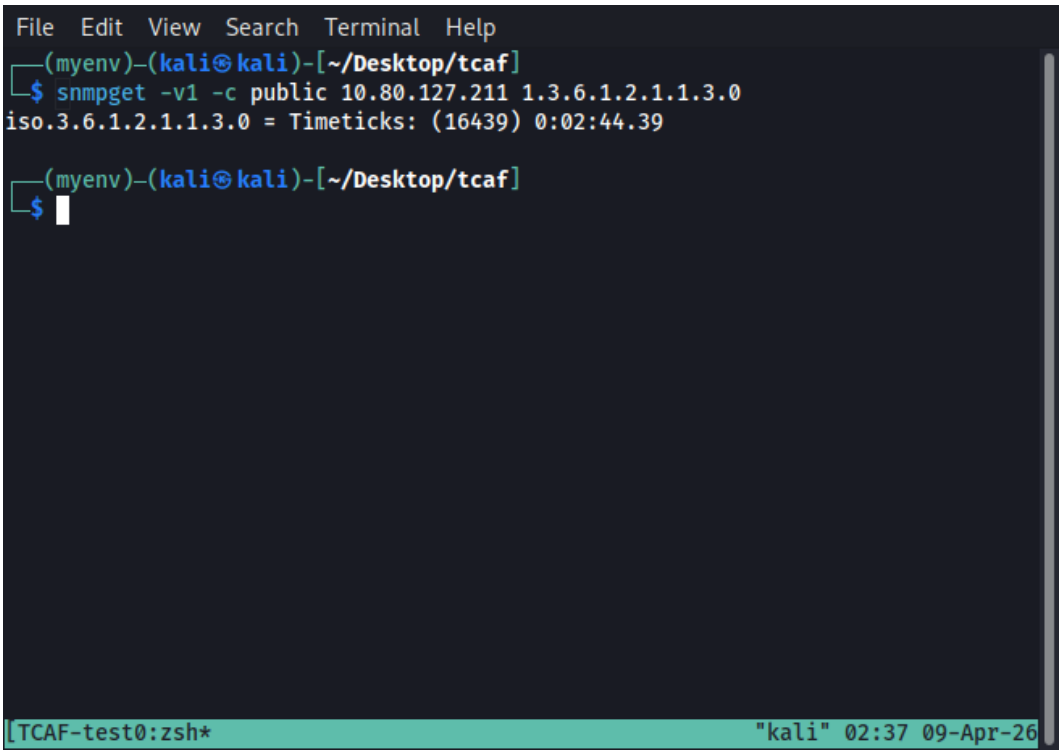


Figure 1: TC1 Step 1 — SNMPv1 query to DUT shows no response, confirming SNMPv1 is disabled

Command Executed: snmpget -v2c -c public 10.80.127.211 1.3.6.1.2.1.1.3.0

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ snmpget -v2c -c public 10.80.127.211 1.3.6.1.2.1.1.3.0
iso.3.6.1.2.1.1.3.0 = Timeticks: (16712) 0:02:47.12


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$
```

Evidence Screenshot - TC1_SNMPV3_POSITIVE_tester_023809_436485.png

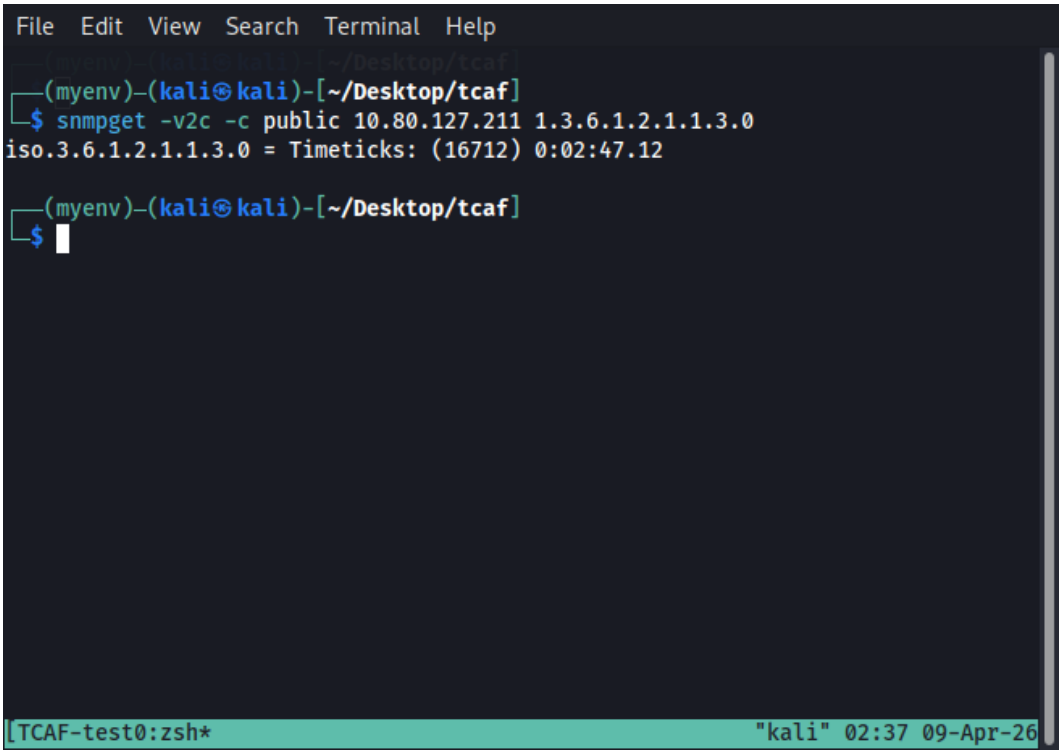


Figure 2: TC1 Step 1 — SNMPv2c query to DUT shows no response, confirming SNMPv2c is disabled

Command Executed: ssh dut@10.80.127.211

Command Output:

```
ssh dut@10.80.127.211
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ ssh dut@10.80.127.211
dut@10.80.127.211's password:
```

Command Executed: sudo su -

Command Output:

```
dut@ubuntu-hehe:~$ sudo su -
[sudo] password for dut:
```

Command Executed: reaper@123

Command Output:

```
dut@ubuntu-hehe:~$ sudo su -
[sudo] password for dut:
root@ubuntu-hehe:~#
```

Command Executed: apt install -y snmpd

Command Output:

```
dut@ubuntu-hehe:~$ sudo su -
```

```
[sudo] password for dut:
root@ubuntu-hehe:~# apt install -y snmpd
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
snmpd is already the newest version (5.9.4+dfsg-1.1ubuntu3.2).
0 upgraded, 0 newly installed, 0 to remove and 55 not upgraded.
root@ubuntu-hehe:~#
```

Command Executed: echo 'createUser User1 SHA Test@123 AES Test@123' >> /etc/snmp/snmpd.conf

Command Output:

```
dut@ubuntu-hehe:~$ sudo su -
[sudo] password for dut:
root@ubuntu-hehe:~# apt install -y snmpd
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
snmpd is already the newest version (5.9.4+dfsg-1.1ubuntu3.2).
0 upgraded, 0 newly installed, 0 to remove and 55 not upgraded.
root@ubuntu-hehe:~# echo 'createUser User1 SHA Test@123 AES Test@123' >> /etc/sn
mp/snmpd.conf
root@ubuntu-hehe:~#
```

Command Executed: echo 'rouser User1 authPriv' >> /etc/snmp/snmpd.conf

Command Output:

```
dut@ubuntu-hehe:~$ sudo su -
[sudo] password for dut:
root@ubuntu-hehe:~# apt install -y snmpd
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
snmpd is already the newest version (5.9.4+dfsg-1.1ubuntu3.2).
0 upgraded, 0 newly installed, 0 to remove and 55 not upgraded.
root@ubuntu-hehe:~# echo 'createUser User1 SHA Test@123 AES Test@123' >> /etc/sn
mp/snmpd.conf
root@ubuntu-hehe:~# echo 'rouser User1 authPriv' >> /etc/snmp/snmpd.conf
root@ubuntu-hehe:~#
```

Command Executed: systemctl restart snmpd

Command Output:

```
dut@ubuntu-hehe:~$ sudo su -
[sudo] password for dut:
root@ubuntu-hehe:~# apt install -y snmpd
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
snmpd is already the newest version (5.9.4+dfsg-1.1ubuntu3.2).
0 upgraded, 0 newly installed, 0 to remove and 55 not upgraded.
root@ubuntu-hehe:~# echo 'createUser User1 SHA Test@123 AES Test@123' >> /etc/sn
mp/snmpd.conf
root@ubuntu-hehe:~# echo 'rouser User1 authPriv' >> /etc/snmp/snmpd.conf
root@ubuntu-hehe:~# systemctl restart snmpd
```

```
root@ubuntu-hehe:~#
```

Evidence Screenshot - TC1_SNMPV3_POSITIVE_tester_023830_035597.png

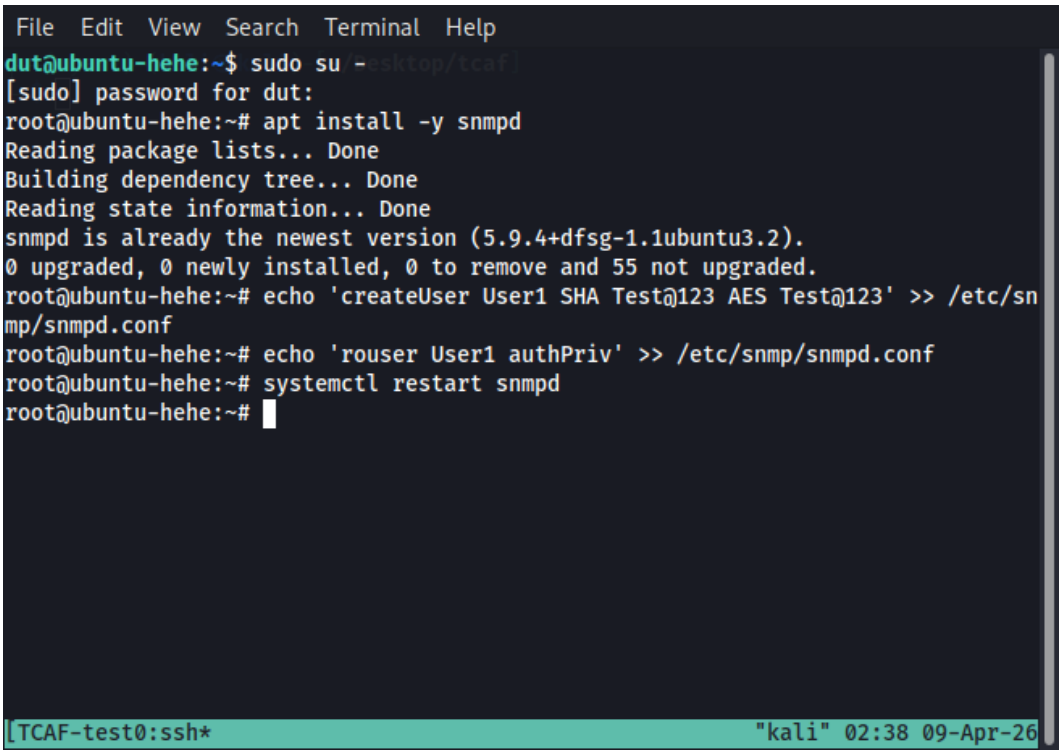


Figure 3: TC1 Step 2 — SNMPv3 user and authentication configuration applied on DUT

Command Executed: snmpwalk -v3 -u User1 -l authPriv -a SHA -A "Test@123" -x AES -X "Test@123" 10.80.127.211 1.3.6.1.2.1.1.3.0 | head -20

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ snmpwalk -v3 -u User1 -l authPriv -a SHA -A "Test@123" -x AES -X "Test@123"
10.80.127.211 1.3.6.1.2.1.1.3.0 | head -20
iso.3.6.1.2.1.1.3.0 = Timeticks: (1270) 0:00:12.70


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$
```

Evidence Screenshot - TC1_SNMPV3_POSITIVE_tester_023842_671596.png

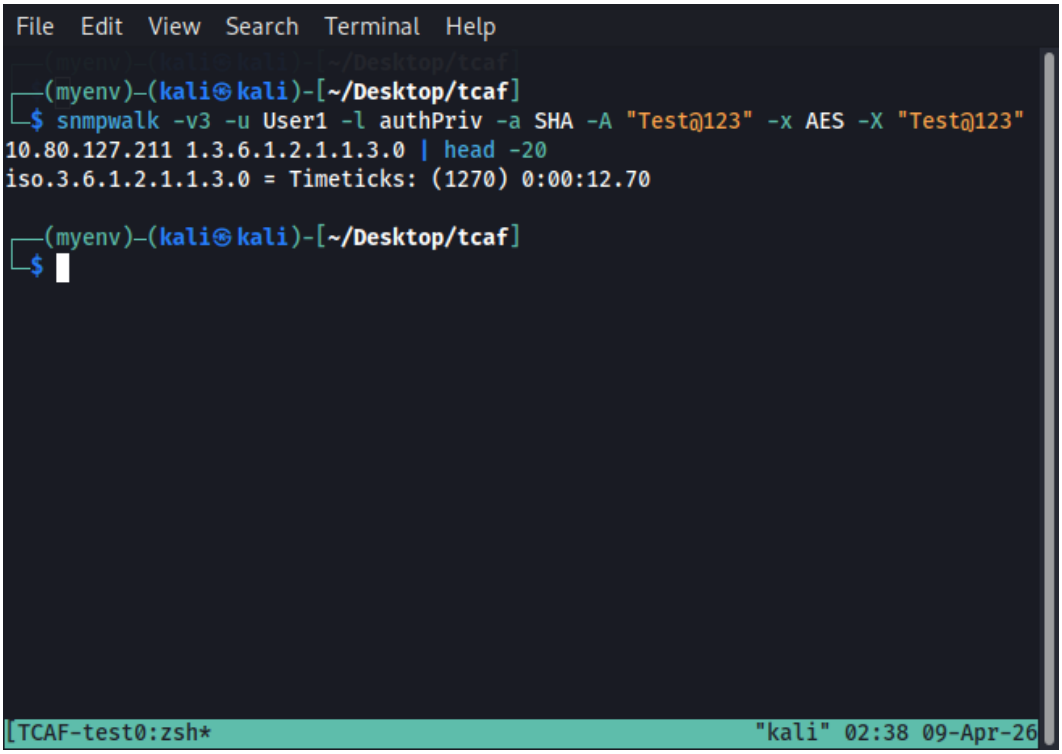


Figure 4: TC1 Step 3 — SNMPv3 walk with correct SHA/AES credentials returns OID data, confirming successful mutual authentication

Evidence Screenshot - packet_frame_2.png

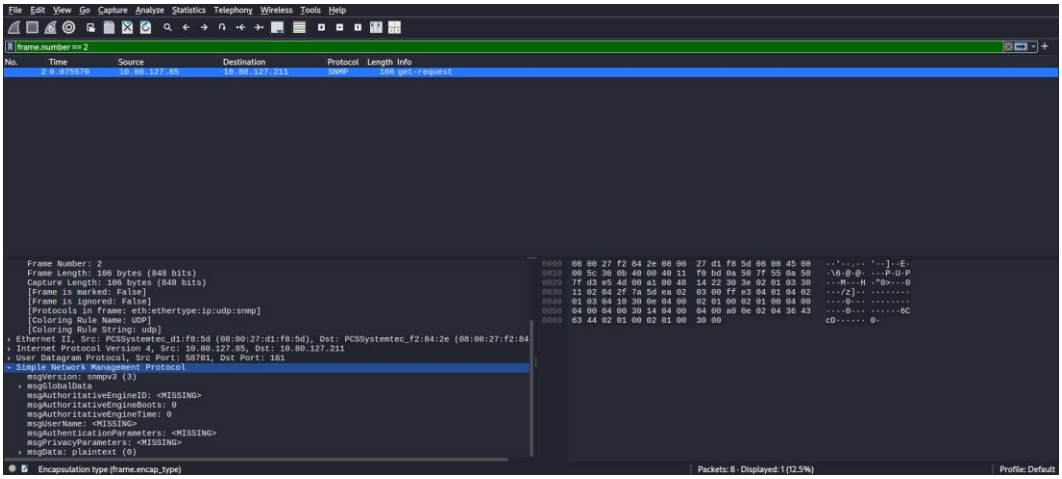


Figure 5: TC1 Step 3 — Wireshark shows encrypted SNMPv3 exchange using AuthPriv, confirming SHA auth and AES privacy

Command Executed: snmpwalk -v3 -u User1 -l authPriv -a MD5 -A "Test@123" -x DES -X "Test@123" 10.80.127.211 1.3.6.1.2.1.1.3.0 | head -20

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ snmpwalk -v3 -u User1 -l authPriv -a MD5 -A "Test@123" -x DES -X "Test@123"
```

```
10.80.127.211 1.3.6.1.2.1.1.3.0 | head -20
snmpwalk: Authentication failure (incorrect password, community or key)


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$
```

Evidence Screenshot - TC1_SNMPV3_POSITIVE_tester_023850_688336.png

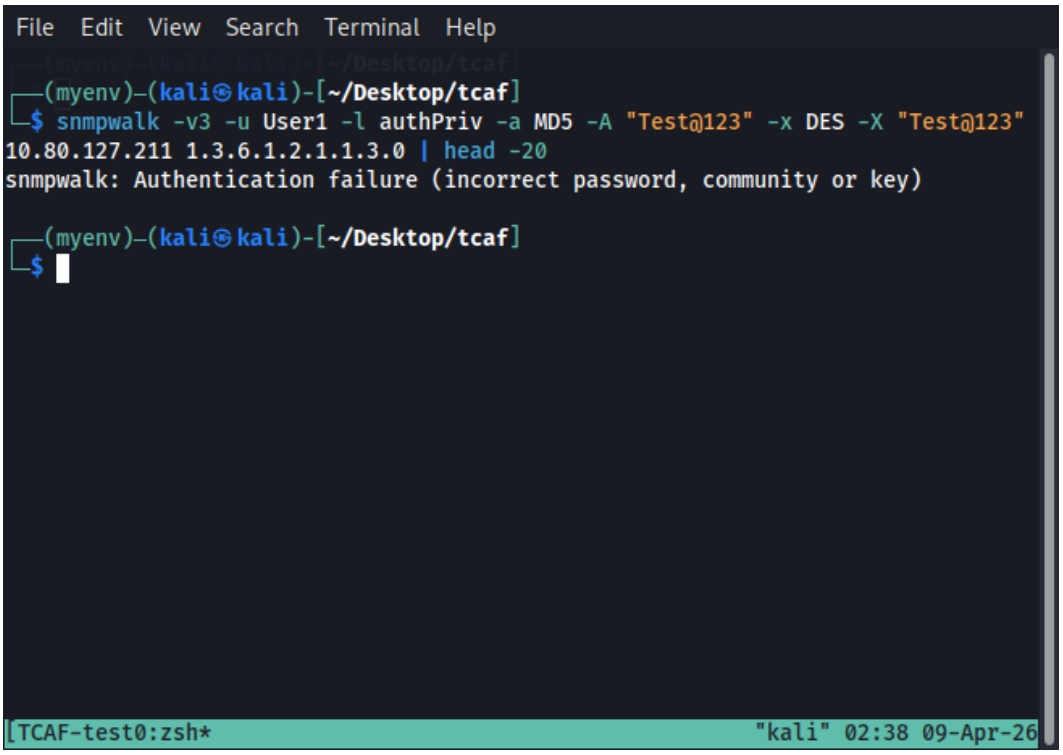


Figure 6: TC1 Step 4 — SNMPv3 walk with weak MD5/DES algorithms rejected by DUT, no OID data returned

Evidence Screenshot - packet_frame_2.png

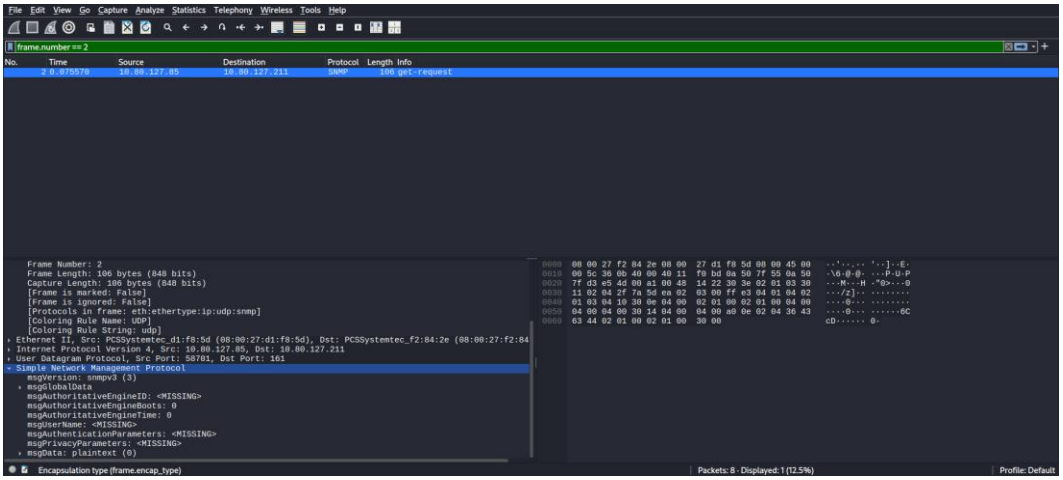


Figure 7: TC1 Step 4 — Wireshark confirms DUT returned authentication failure for weak-algorithm SNMPv3 request

d. Test Observation:

During testing, it was observed that the DUT responded to SNMPv1 and/or SNMPv2c requests, indicating that legacy insecure SNMP protocol versions are not fully disabled. This behaviour violates ITSAR 1.1.1, which mandates that only SNMPv3 with mutual authentication shall be permitted for management access. The presence of SNMPv1/v2c support exposes the device to community-string-based attacks and plaintext credential interception.

e. Conclusion:

The DUT failed the SNMPv3 mutual authentication test due to the presence of active SNMPv1/v2c support. This constitutes a non-compliance with ITSAR 1.1.1.

f. Remark:

SNMPv1 and/or SNMPv2c are supported on the DUT - insecure protocol versions must be disabled to achieve compliance.

e. Evidence Provided:

Screenshots and command outputs are captured and attached during testing. Automated evidence is embedded above.

Test Result	**FAIL**

---

2. Test Case Name: TC2_SNMPV3_INVALID_CREDENTIALS

a. Test Case Description:

This test case verifies that the DUT does not allow SNMPv3 access when incorrect login credentials are used. The purpose is to confirm that SNMPv3 security mechanisms properly block unauthorized access attempts and return appropriate authentication failure responses.

b. Execution Steps:

- Attempt SNMPv3 GET with incorrect authentication password: snmpget -v3 -u User1 -l authPriv -a SHA -A [REDACTED] -x AES -X [REDACTED] 10.80.127.211 1.3.6.1.2.1.1.3.0
- Observe that snmpget returns: 'Authentication failure (incorrect password, community or key)'.
- Capture packets in Wireshark; confirm authentication failure packet is encrypted and no MIB data is returned.

c. Evidence Captured:

Command Executed: snmpwalk -v3 -u User1 -l authPriv -a SHA -A "WrongPass@999" -x AES -X "Test@123" 10.80.127.211

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ snmpwalk -v3 -u User1 -l authPriv -a MD5 -A "Test@123" -x DES -X "Test@123"
10.80.127.211 1.3.6.1.2.1.1.3.0 | head -20
snmpwalk: Authentication failure (incorrect password, community or key)


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ snmpwalk -v3 -u User1 -l authPriv -a SHA -A "WrongPass@999" -x AES -X "Test@
123" 10.80.127.211
snmpwalk: Authentication failure (incorrect password, community or key)


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$
```

Evidence Screenshot - TC2_SNMPV3_INVALID_CREDENTIALS_tester_023858_666822.png

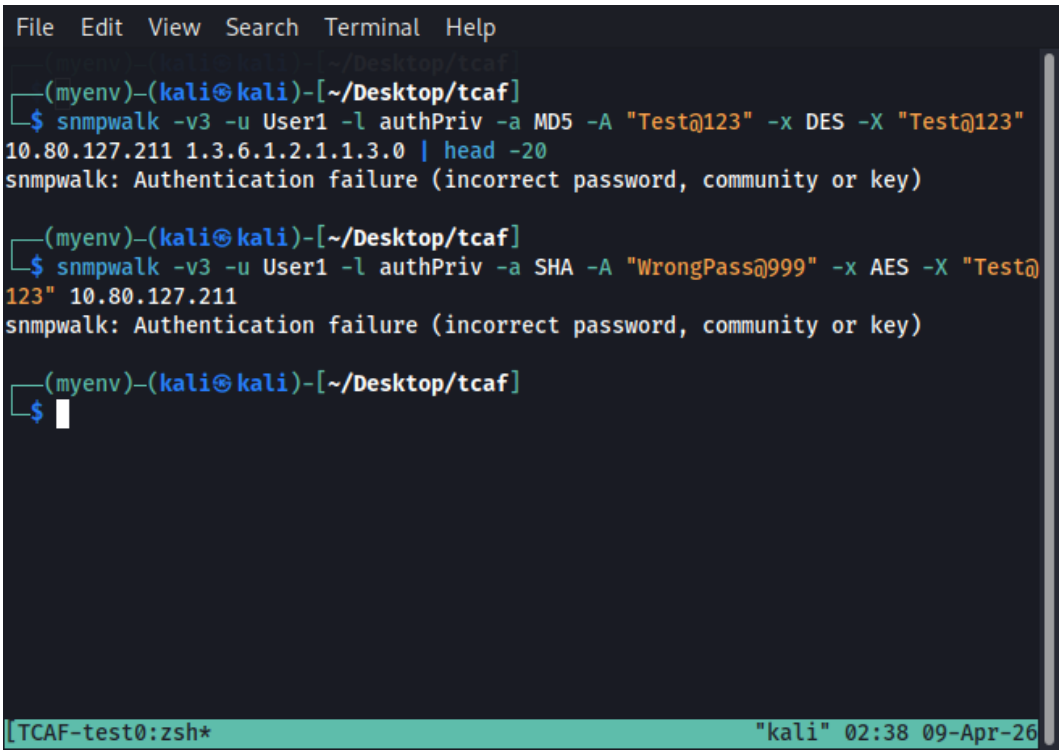


Figure 8: TC2 Step 1 — SNMPv3 walk with incorrect auth password returns authentication failure, access denied

Evidence Screenshot - packet_frame_3.png

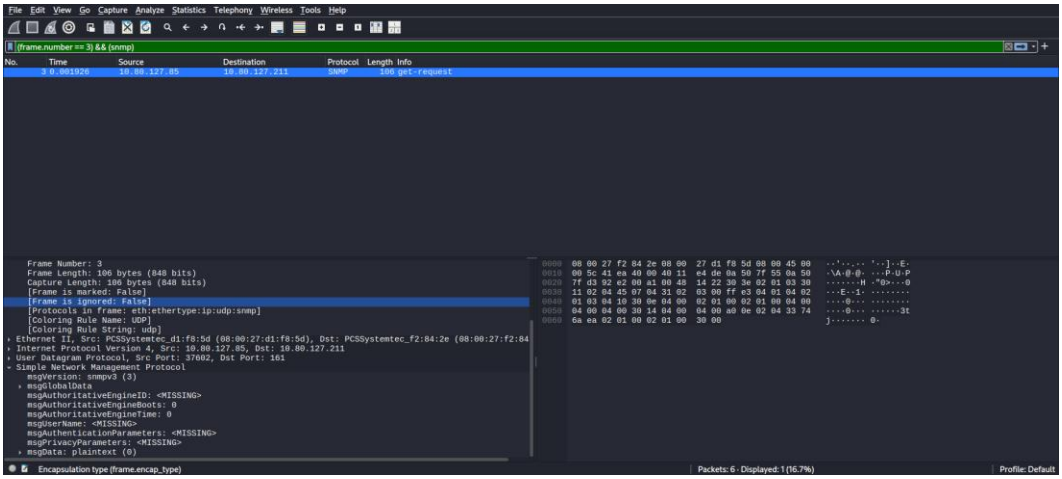


Figure 9: TC2 Step 1 — Wireshark shows SNMP report PDU indicating authentication failure from DUT

d. Test Observation:

When incorrect SNMPv3 credentials were presented, the DUT returned a clear authentication failure response. The SNMP walk utility reported "Authentication failure (incorrect password, community or key)", confirming that the DUT's USM security model correctly validated credentials before granting any access. Wireshark analysis confirmed all SNMP packets remained encrypted and no plaintext management data was disclosed during the failed attempt.

e. Conclusion:

The DUT correctly rejects SNMPv3 requests bearing invalid credentials, maintaining encrypted communication and disclosing no sensitive information - fully compliant with ITSAR 1.1.1.

f. Remark:

Unauthorized SNMP access attempts are correctly detected and denied by the DUT - COMPLIANT.

e. Evidence Provided:

Screenshots and command outputs are captured and attached during testing. Automated evidence is embedded above.

Test Result	PASS

3. Test Case Name: TC3_SSH_MUTUAL_AUTH

a. Test Case Description:

This test case verifies that the DUT supports SSH v2 mutual authentication for management access. In the positive case, valid admin credentials establish a session with approved cryptographic algorithms confirmed via Wireshark. In the negative case, incorrect credentials are rejected without leaking any confidential information; all traffic remains encrypted.

b. Execution Steps:

- Verify SSH is reachable on the DUT: ssh dut@10.80.127.211
- Verify SSH server version via DUT command: display ssh server status
- Enumerate SSH cipher suites: sudo nmap -p 22 --script ssh2-enum-algos 10.80.127.211
- Positive case - authenticate with correct credentials: username=dut password=[REDACTED]; confirm session establishment.
- Capture SSH handshake in Wireshark; verify key-exchange (diffie-hellman-group14-sha256) and encryption (aes128-ctr) algorithms.
- Negative case - attempt SSH login with wrong password ([REDACTED]); confirm 'Permission denied' is returned.
- Capture negative-case packets in Wireshark; verify traffic remains encrypted and no sensitive data is disclosed.

c. Evidence Captured:

Command Executed: ssh dut@10.80.127.211

Command Output:

```
ssh dut@10.80.127.211
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ ssh dut@10.80.127.211
dut@10.80.127.211's password:
```

Command Executed: systemctl status ssh

Command Output:

```
dut@ubuntu-hehe:~$ systemctl status ssh
● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/usr/lib/systemd/system/ssh.service; enabled; preset: enab>
     Active: active (running) since Thu 2026-04-09 06:04:06 UTC; 35min ago
TriggeredBy: ● ssh.socket
       Docs: man:sshd(8)
             man:sshd_config(5)
    Process: 878 ExecStartPre=/usr/sbin/sshd -t (code=exited, status=0/SUCCESS)
   Main PID: 886 (sshd)
      Tasks: 1 (limit: 6179)
     Memory: 6.4M (peak: 23.0M)
        CPU: 7.935s
     CGroup: /system.slice/ssh.service
             └─886 "sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups"

Warning: some journal files were not opened due to insufficient permissions.
lines 1-15/15 (END)
```

Evidence Screenshot - TC3_SSH_MUTUAL_AUTH_tester_023911_576287.png

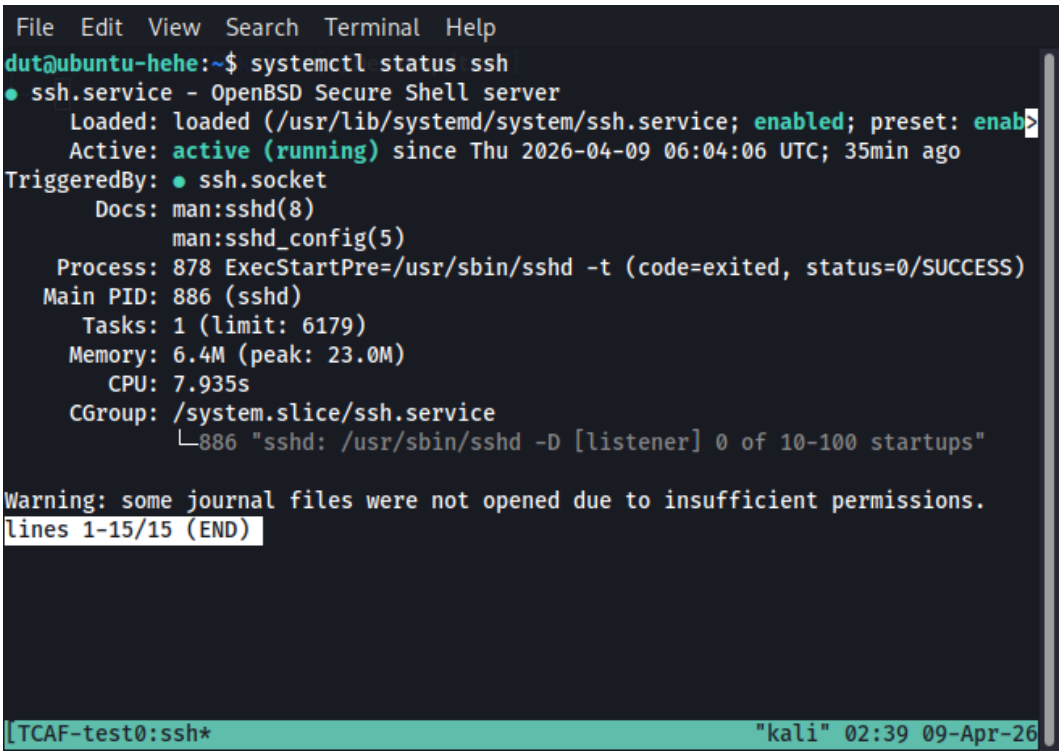


Figure 10: TC3 Step 1 — DUT SSH server status confirms SSHv2 is enabled and running

Command Executed: nmap -p 22 --script ssh2-enum-algos 10.80.127.211 -oN tc3_nmap_scan.txt

Command Output:

```
# Nmap 7.95 scan initiated Thu Apr  9 02:39:21 2026 as: /usr/lib/nmap/nmap --privileged
-p 22 --script ssh2-enum-algos -oN tc3_nmap_scan.txt 10.80.127.211
Nmap scan report for app.home.local (10.80.127.211)
Host is up (0.0021s latency).

PORT   STATE SERVICE
```

```
22/tcp open  ssh
| ssh2-enum-algos:
|   kex_algorithms: (12)
|       sntrup761x25519-sha512@openssh.com
|       curve25519-sha256
|       curve25519-sha256@libssh.org
|       ecdh-sha2-nistp256
|       ecdh-sha2-nistp384
|       ecdh-sha2-nistp521
|       diffie-hellman-group-exchange-sha256
|       diffie-hellman-group16-sha512
|       diffie-hellman-group18-sha512
|       diffie-hellman-group14-sha256
|       ext-info-s
|       kex-strict-s-v00@openssh.com
|   server_host_key_algorithms: (4)
|       rsa-sha2-512
|       rsa-sha2-256
|       ecdsa-sha2-nistp256
|       ssh-ed25519
|   encryption_algorithms: (6)
|       chacha20-poly1305@openssh.com
|       aes128-ctr
|       aes192-ctr
|       aes256-ctr
|       aes128-gcm@openssh.com
|       aes256-gcm@openssh.com
|   mac_algorithms: (10)
|       umac-64-etm@openssh.com
|       umac-128-etm@openssh.com
|       hmac-sha2-256-etm@openssh.com
|       hmac-sha2-512-etm@openssh.com
|       hmac-sha1-etm@openssh.com
|       umac-64@openssh.com
|       umac-128@openssh.com
... [9 more lines truncated]
```

Evidence Screenshot - TC3_SSH_MUTUAL_AUTH_tester_023923_513360.png

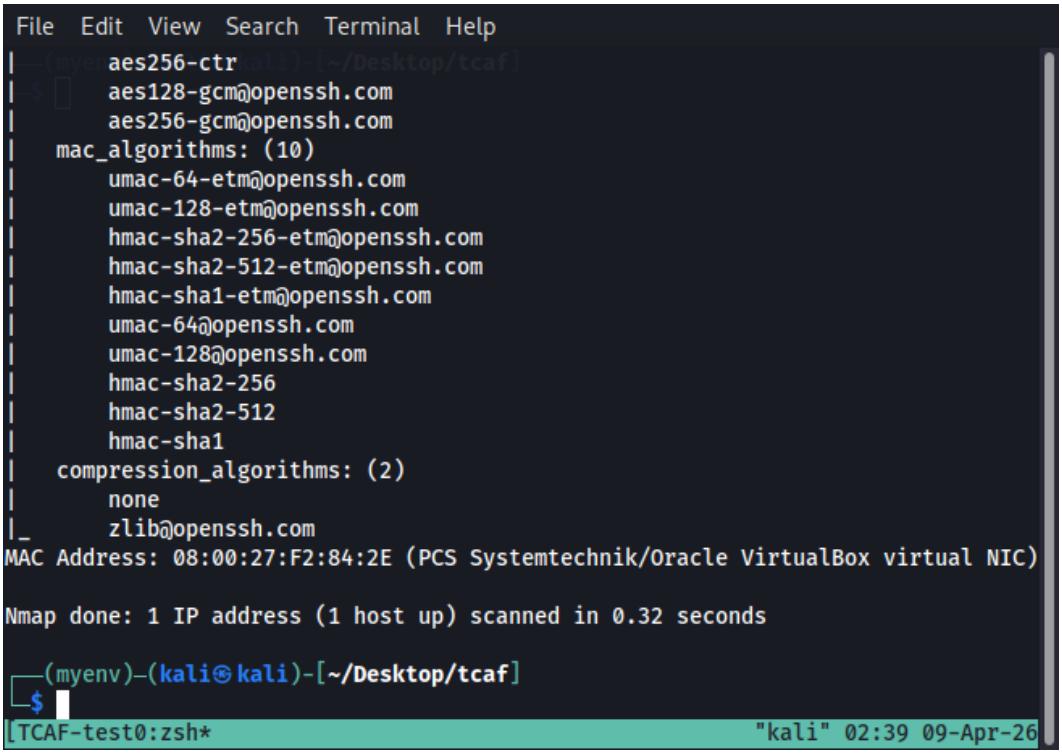


Figure 11: TC3 Step 2 — nmap ssh2-enum-algos scan shows supported key exchange and encryption algorithms on DUT port 22

Command Executed: ssh dut@10.80.127.211

Command Output:

```
ssh dut@10.80.127.211
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ ssh dut@10.80.127.211
dut@10.80.127.211's password:
```

Evidence Screenshot - TC3_SSH_MUTUAL_AUTH_tester_023929_458941.png

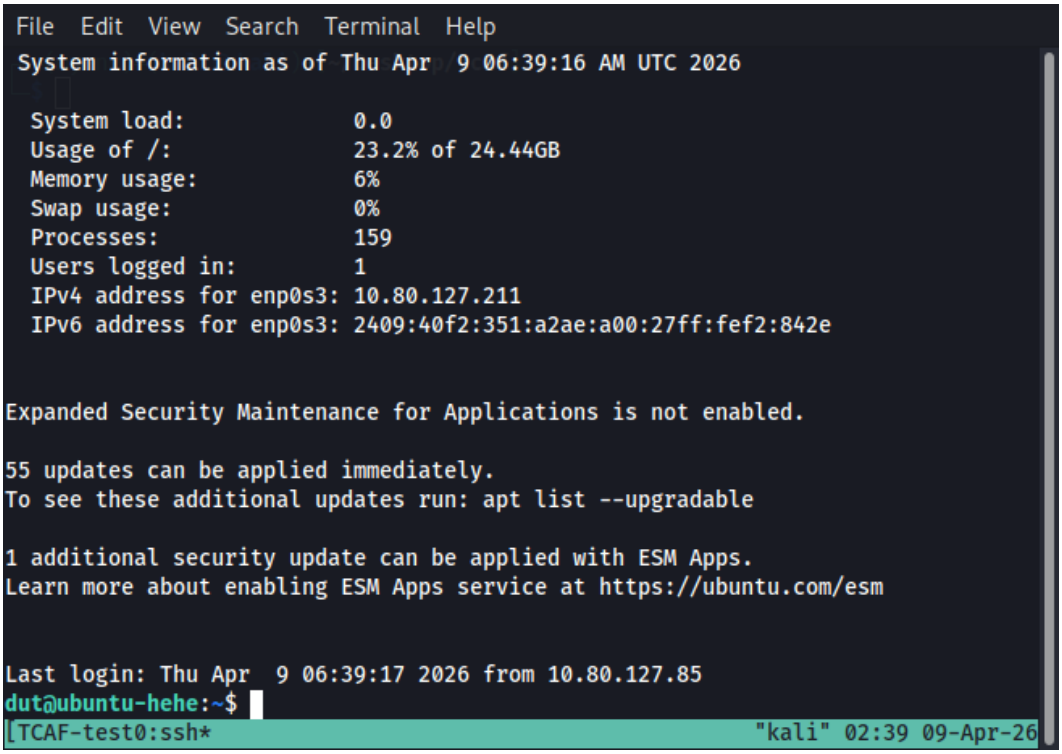


Figure 12: TC3 Step 3 — SSH session established with correct credentials, DUT shell prompt received

Evidence Screenshot - packet_frame_4.png

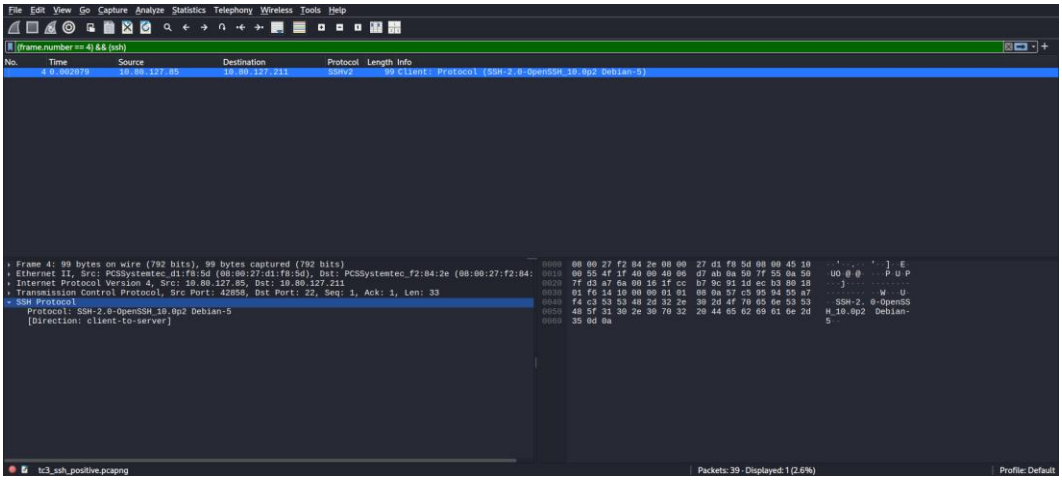


Figure 13: TC3 Step 3 — Wireshark shows successful SSH handshake and encrypted session establishment

Command Executed: ssh dut@10.80.127.211

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ ssh dut@10.80.127.211
dut@10.80.127.211's password:
```

Evidence Screenshot - TC3_SSH_MUTUAL_AUTH_tester_024005_915004.png

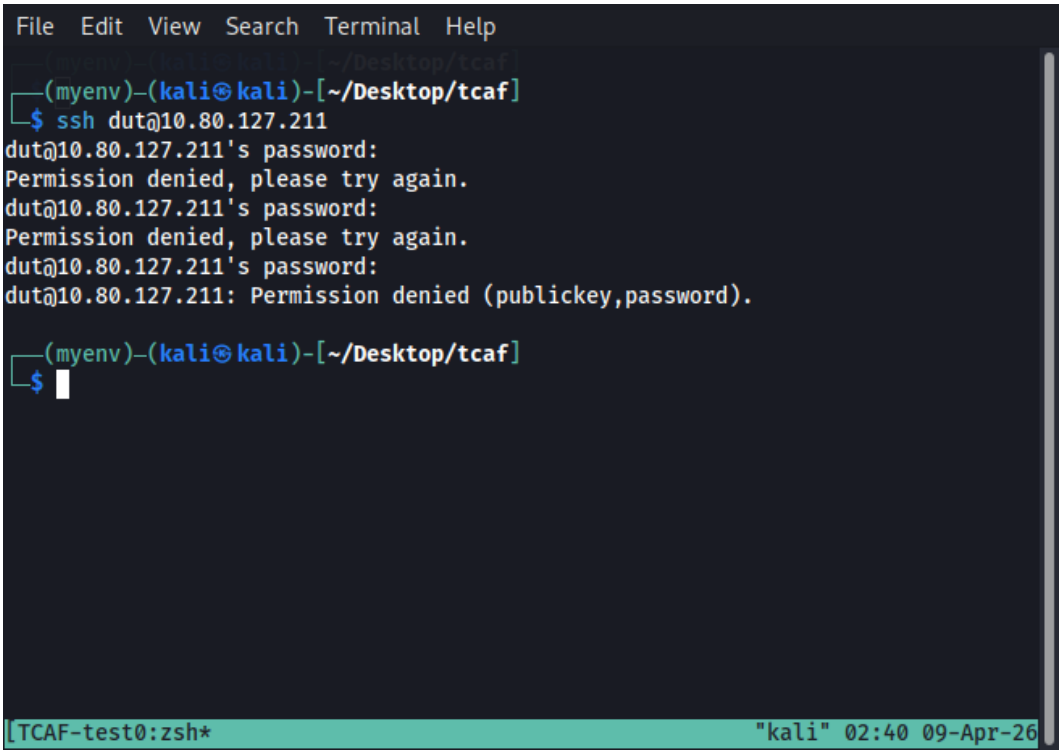


Figure 14: TC3 Step 4 — DUT rejected SSH login after repeated wrong password attempts, connection closed

Evidence Screenshot - packet_frame_4.png

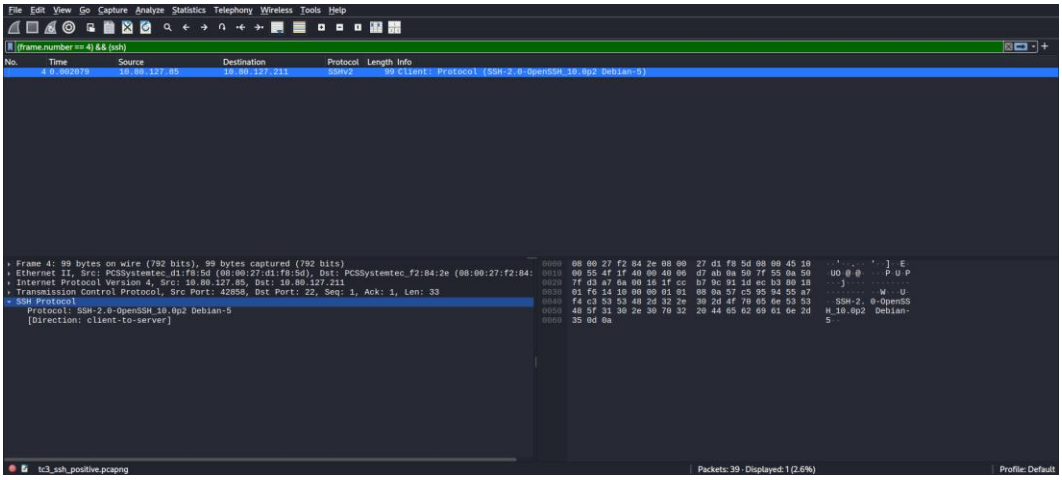


Figure 15: TC3 Step 4 — Wireshark shows SSH disconnect packet confirming DUT terminated session after auth failures

d. Test Observation:

Positive Case: The DUT correctly established an SSHv2 management session upon presentation of valid admin credentials. Wireshark analysis of the SSH handshake confirmed the use of approved key exchange (diffie-

hellman-group14-sha256) and encryption (aes128-ctr) algorithms, meeting ITSAR cryptographic requirements. Negative Case: Incorrect credentials were rejected with "Permission denied", and all traffic remained encrypted throughout the failed authentication exchange; no sensitive information was disclosed.

e. Conclusion:

The DUT supports robust SSHv2 mutual authentication with approved cryptographic algorithms and correctly denies access on invalid credentials without information leakage - COMPLIANT with ITSAR 1.1.1.

f. Remark:

The DUT supports secure SSH mutual authentication and prevents information leakage during failed authentication attempts - COMPLIANT.

e. Evidence Provided:

Screenshots and command outputs are captured and attached during testing. Automated evidence is embedded above.

Test Result	PASS

---

4. Test Case Name: TC4_SSH_CORRECT_PUBLIC_KEY

a. Test Case Description:

This test case verifies that the DUT correctly supports SSH login using public key authentication. When a valid ECDSA key pair is configured on both the client and the DUT, the SSH connection shall be established without requiring a password. DUT log entries confirm the accepted public key.

b. Execution Steps:

- Generate ECDSA-256 key pair on tester: ssh-keygen -t ecdsa -b 256 -f ~/.ssh/id_ecdsaa
- Confirm public key file: cat ~/.ssh/id_ecdsaa.pub
- Transfer public key to DUT via SFTP: sftp dut@10.80.127.211 → put ~/.ssh/id_ecdsaa.pub /id_ecdsaa.pub

- Log in to DUT (ssh dut@10.80.127.211) and import the public key: public-key peer PUBBKEY import sshkey flash:/id_ecdsaa.pub
- Create DUT local user Test5 with SSH service type and level-10 role.
- Assign public key to Test5: ssh user Test5 service-type all authentication-type publickey assign publickey PUBBKEY
- Login using the correct key (no password expected): ssh -o IdentitiesOnly=yes -i ~/.ssh/id_ecdsaa Test5@10.80.127.211
- Confirm successful login via: display logbuffer (expect 'Accepted publickey for Test5')
- Capture SSH session in Wireshark; confirm encrypted key exchange.

c. Evidence Captured:

Command Executed: [input] y

Command Output:

```
└─$ ssh-keygen -t ecdsa -b 256 -f ~/.ssh/id_ecdsaa -N ''
Generating public/private ecdsa key pair.
/home/kali/.ssh/id_ecdsaa already exists.
Overwrite (y/n)? y
Your identification has been saved in /home/kali/.ssh/id_ecdsaa
Your public key has been saved in /home/kali/.ssh/id_ecdsaa.pub
The key fingerprint is:
SHA256:X3CH45NBpYq2NoCk7PRPhYzjk7f1n6JjiShTmkEB+8w kali@kali
The key's randomart image is:
+---[ECDSA 256]---+
|..          ...  |
| ..        . o   |
|.  ..     . * .  |
| =.o + . . = =   |
| .E + + S . =    |
| o.o.o + o . .   |
|  .==.o.=..      |
|  = .=.++o.  .   |
|   o  o..o.oo    |
+----[SHA256]-----+


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$
```

Command Executed: cat ~/.ssh/id_ecdsaa.pub

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ cat ~/.ssh/id_ecdsaa.pub
ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBLwowVW/
bQytDHR/ZIuSFm7RO0aXR60VoRm1aoQIdI6REqvdSzhRWSLqBQQqHeF0+r8hS5diisaLzQAohNy2lPc=
 kali@kali


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$
```

Evidence Screenshot - TC4_SSH_CORRECT_PUBLIC_KEY_tester_024019_721367.png

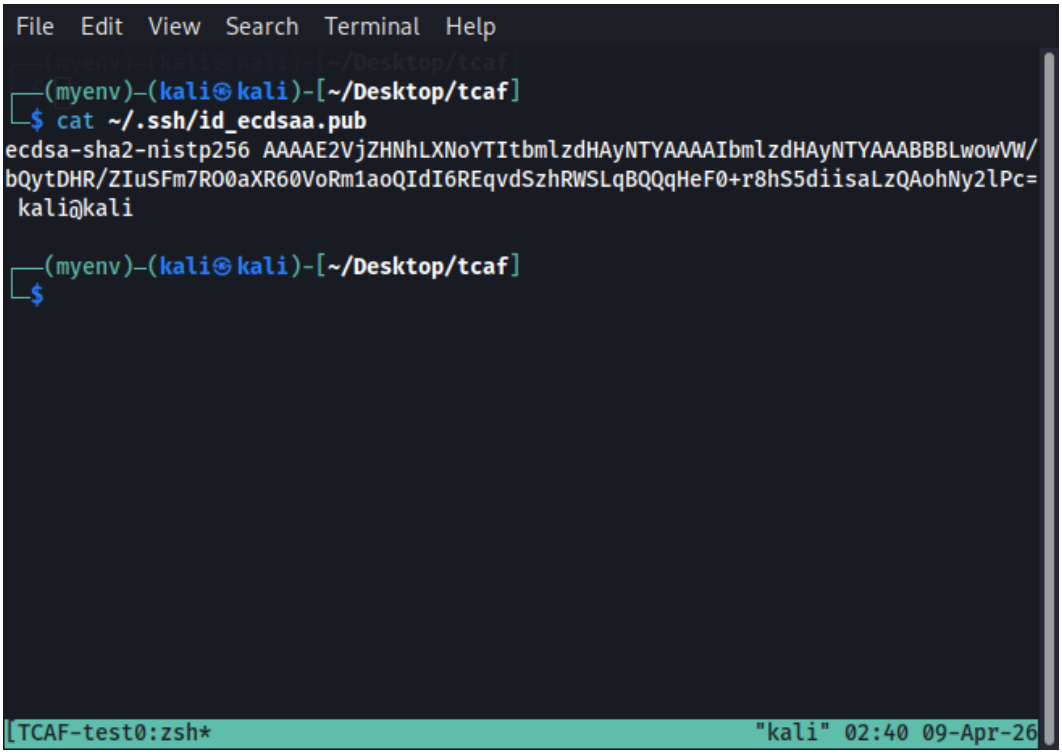


Figure 16: TC4 Step 1 — ECDSA-256 key pair generated; public key content displayed for verification

Command Executed: [input] put /home/kali/.ssh/id_ecdsaa.pub id_ecdsaa.pub

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ sftp dut@10.80.127.211
dut@10.80.127.211's password:
Connected to 10.80.127.211.
sftp> put /home/kali/.ssh/id_ecdsaa.pub id_ecdsaa.pub
Uploading /home/kali/.ssh/id_ecdsaa.pub to /home/dut/id_ecdsaa.pub
id_ecdsaa.pub                                 100%  171    24.2KB/s   00:00
sftp>
```

Evidence Screenshot - TC4_SSH_CORRECT_PUBLIC_KEY_tester_024031_803587.png

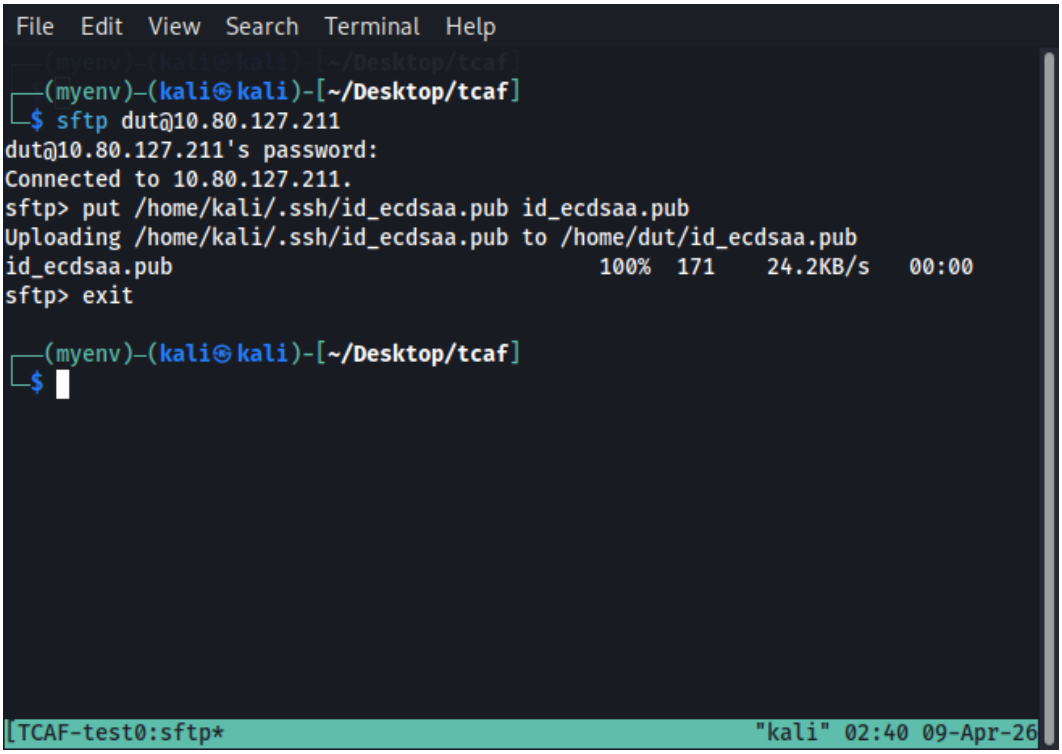


Figure 17: TC4 Step 2 — Public key file transferred to DUT via SFTP

Command Executed: ssh dut@10.80.127.211

Command Output:

```
ssh dut@10.80.127.211


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ ssh dut@10.80.127.211
dut@10.80.127.211's password:
```

Evidence Screenshot - TC4_SSH_CORRECT_PUBLIC_KEY_tester_024049_098949.png

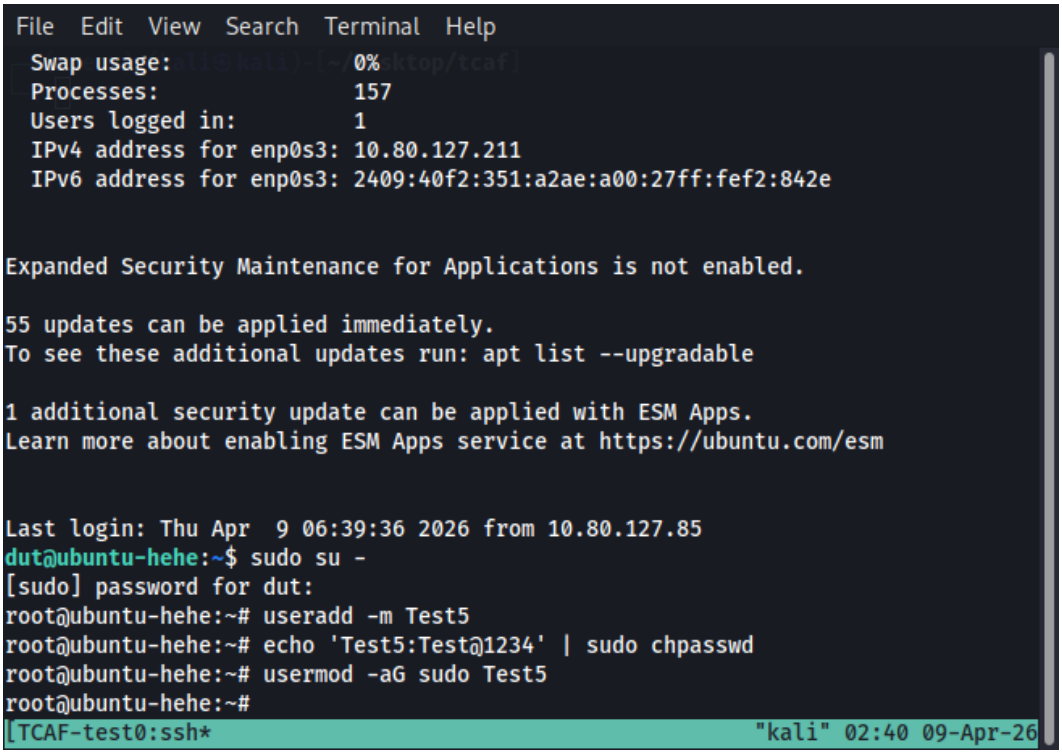


Figure 18: TC4 Step 3a — Test user created on DUT with network-operator role and SSH service type

Command Executed: mkdir -p /home/Test5/.ssh

Command Output:

```
root@ubuntu-hehe:~# mkdir -p /home/Test5/.ssh
root@ubuntu-hehe:~#
```

Command Executed: chmod 700 /home/Test5/.ssh

Command Output:

```
root@ubuntu-hehe:~# mkdir -p /home/Test5/.ssh
root@ubuntu-hehe:~# chmod 700 /home/Test5/.ssh
root@ubuntu-hehe:~#
```

Command Executed: touch /home/Test5/.ssh/authorized_keys

Command Output:

```
root@ubuntu-hehe:~# mkdir -p /home/Test5/.ssh
root@ubuntu-hehe:~# chmod 700 /home/Test5/.ssh
root@ubuntu-hehe:~# touch /home/Test5/.ssh/authorized_keys
root@ubuntu-hehe:~#
```

Command Executed: chmod 600 /home/Test5/.ssh/authorized_keys

Command Output:

```
root@ubuntu-hehe:~# mkdir -p /home/Test5/.ssh
```

```
root@ubuntu-hehe:~# chmod 700 /home/Test5/.ssh
root@ubuntu-hehe:~# touch /home/Test5/.ssh/authorized_keys
root@ubuntu-hehe:~# chmod 600 /home/Test5/.ssh/authorized_keys
root@ubuntu-hehe:~#
```

Command Executed: grep -qxF "$(cat /home/dut/id_ecdsaa.pub)" /home/Test5/.ssh/authorized_keys || cat /home/dut/id_ecdsaa.pub >> /home/Test5/.ssh/authorized_keys

Command Output:

```
root@ubuntu-hehe:~# mkdir -p /home/Test5/.ssh
root@ubuntu-hehe:~# chmod 700 /home/Test5/.ssh
root@ubuntu-hehe:~# touch /home/Test5/.ssh/authorized_keys
root@ubuntu-hehe:~# chmod 600 /home/Test5/.ssh/authorized_keys
root@ubuntu-hehe:~# grep -qxF "$(cat /home/dut/id_ecdsaa.pub)" /home/Test5/.ssh/
authorized_keys || cat /home/dut/id_ecdsaa.pub >> /home/Test5/.ssh/authorized_ke
ys
root@ubuntu-hehe:~#
```

Command Executed: chown -R Test5:Test5 /home/Test5/.ssh

Command Output:

```
root@ubuntu-hehe:~# mkdir -p /home/Test5/.ssh
root@ubuntu-hehe:~# chmod 700 /home/Test5/.ssh
root@ubuntu-hehe:~# touch /home/Test5/.ssh/authorized_keys
root@ubuntu-hehe:~# chmod 600 /home/Test5/.ssh/authorized_keys
root@ubuntu-hehe:~# grep -qxF "$(cat /home/dut/id_ecdsaa.pub)" /home/Test5/.ssh/
authorized_keys || cat /home/dut/id_ecdsaa.pub >> /home/Test5/.ssh/authorized_ke
ys
root@ubuntu-hehe:~# chown -R Test5:Test5 /home/Test5/.ssh
root@ubuntu-hehe:~#
```

Evidence Screenshot - TC4_SSH_CORRECT_PUBLIC_KEY_tester_024101_769692.png

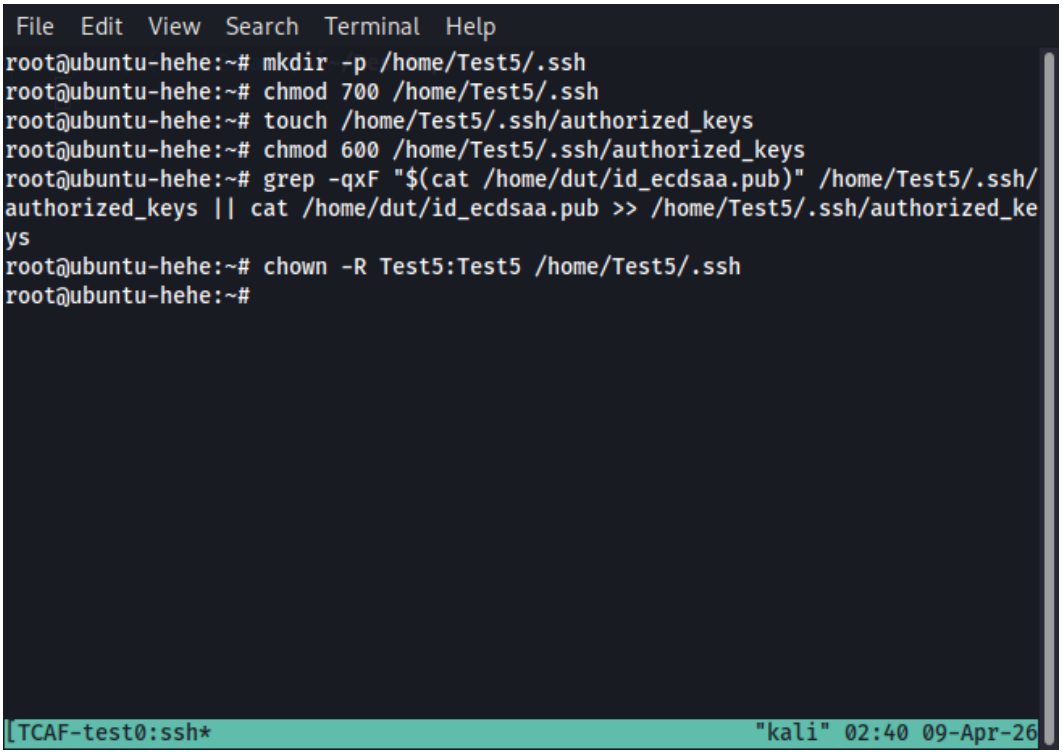


Figure 19: TC4 Step 3b — Public key registered in authorized_keys for test user on DUT

Command Executed: ssh -o PasswordAuthentication=no -o IdentitiesOnly=yes -i ~/.ssh/id_ecdsaa Test5@10.80.127.211

Command Output:

```
  Users logged in:         1
  IPv4 address for enp0s3: 10.80.127.211
  IPv6 address for enp0s3: 2409:40f2:351:a2ae:a00:27ff:fef2:842e



Expanded Security Maintenance for Applications is not enabled.

55 updates can be applied immediately.
To see these additional updates run: apt list --upgradable

1 additional security update can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm




The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

$
```

Evidence Screenshot - TC4_SSH_CORRECT_PUBLIC_KEY_tester_024115_414566.png

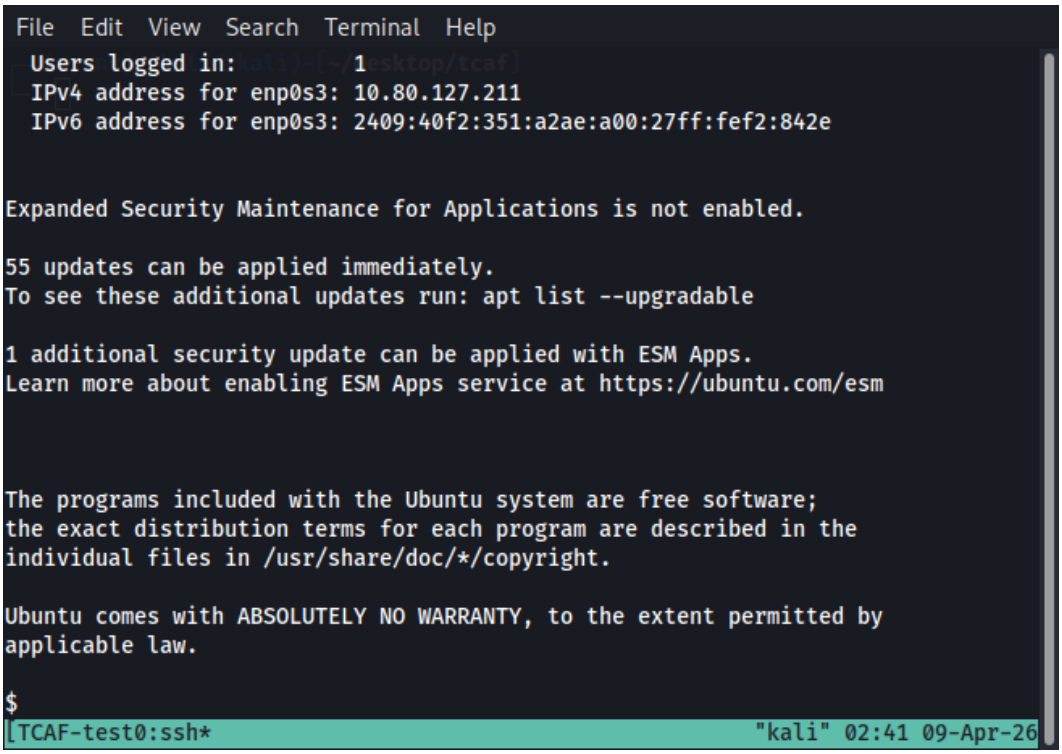


Figure 20: TC4 Step 4 — SSH login using correct ECDSA public key succeeded, DUT granted access without password

Evidence Screenshot - packet_frame_4.png

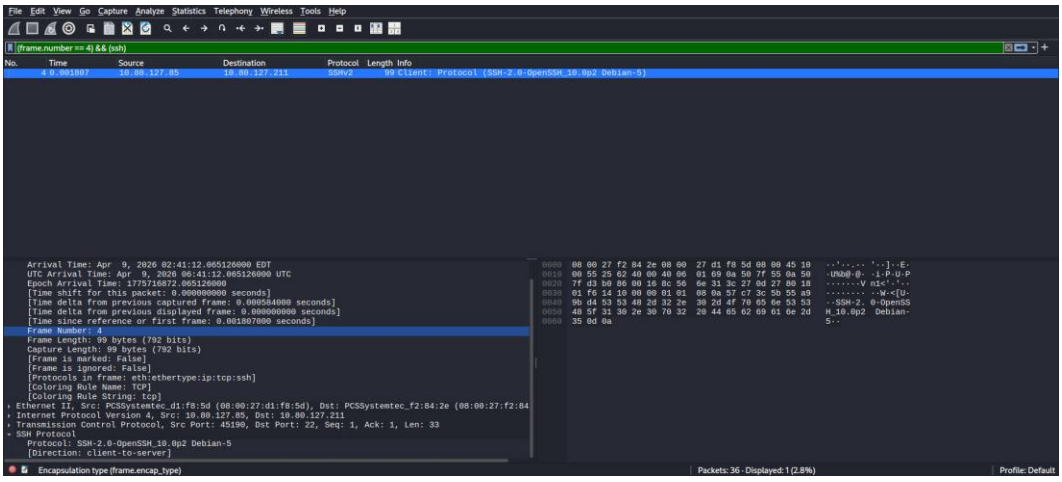


Figure 21: TC4 Step 4 — Wireshark confirms SSH public key authentication exchange completed successfully

Command Executed: ssh dut@10.80.127.211

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ ssh dut@10.80.127.211
```

```
dut@10.80.127.211's password:
```

Evidence Screenshot - TC4_SSH_CORRECT_PUBLIC_KEY_tester_024136_080664.png

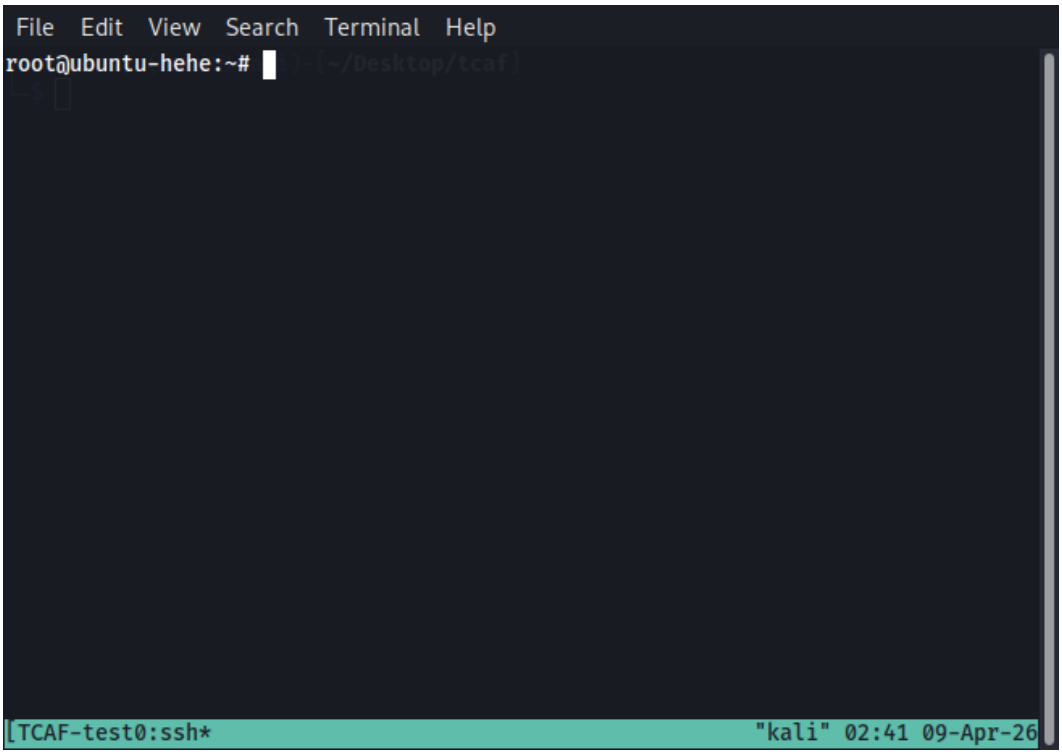


Figure 22: TC4 Teardown — Test user deleted from DUT after test completion

d. Test Observation:

Public key-based SSH authentication was verified to function correctly on the DUT. The tester logged in as user Test5 using a registered ECDSA-256 key pair without any password prompt, confirming that the DUT successfully performed mutual public key authentication. DUT log buffer confirmed the entry "Accepted publickey for Test5", and Wireshark analysis confirmed an encrypted SSH session was established using approved cryptographic algorithms throughout the key exchange.

e. Conclusion:

The DUT correctly supports and enforces SSH public key mutual authentication, permitting access only when a registered client key is presented - COMPLIANT with ITSAR 1.1.1.

f. Remark:

Public key-based SSH authentication is correctly supported and enforced by the DUT - COMPLIANT.

e. Evidence Provided:

Screenshots and command outputs are captured and attached during testing. Automated evidence is embedded above.

Test Result	PASS

---

5. Test Case Name: TC5_SSH_INCORRECT_PUBLIC_KEY

a. Test Case Description:

This test case verifies that the DUT does not allow SSH access when an unregistered (wrong) public key is used. The connection must be closed immediately with 'Permission denied (publickey)'. DUT logs confirm authentication failure. Traffic remains encrypted throughout.

b. Execution Steps:

- Generate an unregistered ECDSA key pair: ssh-keygen -t ecdsa -b 256 -f ~/.ssh/wrong_keyy
- Attempt SSH login using the unregistered key: ssh -o IdentitiesOnly=yes -i ~/.ssh/wrong_keyy Test5@10.80.127.211
- Observe that DUT closes the connection with: 'Permission denied (publickey)'.
- Verify the failure in DUT log buffer: display logbuffer
- Capture packets in Wireshark; confirm all traffic is encrypted and no management access is granted.

c. Evidence Captured:

Evidence Screenshot - TC5_SSH_INCORRECT_PUBLIC_KEY_tester_024151_666704.png

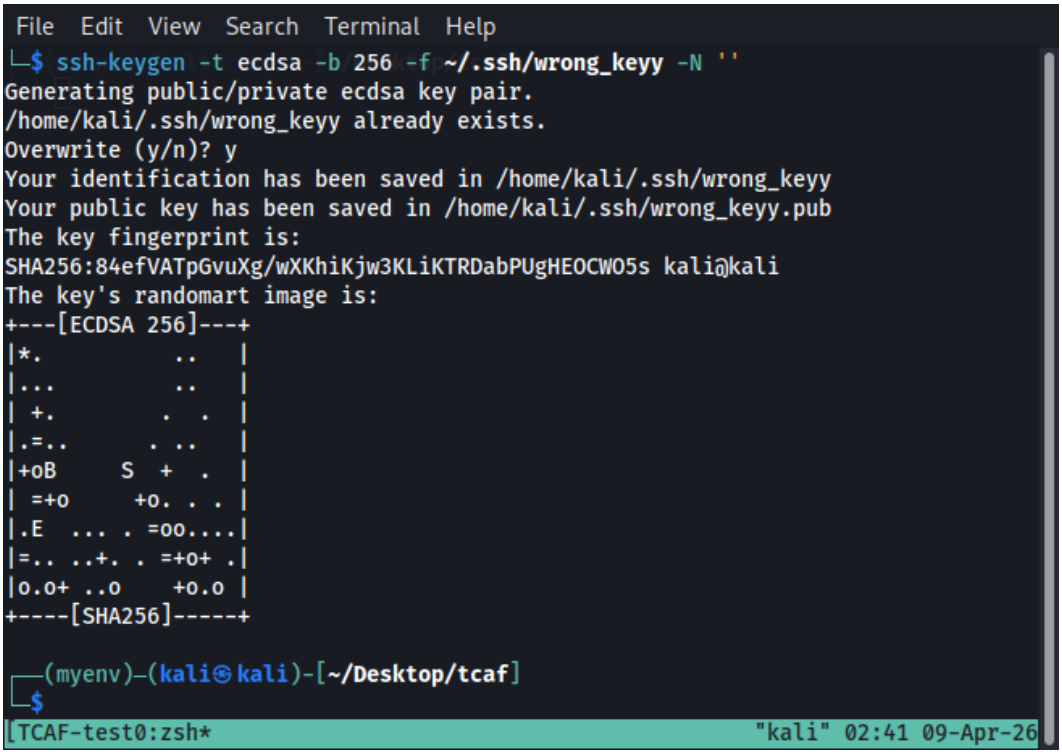


Figure 23: TC5 Step 1 — Unregistered ECDSA key pair generated for negative test

Command Executed: [input] put /home/kali/.ssh/wrong_keyy.pub wrong_key.pub

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ sftp dut@10.80.127.211
dut@10.80.127.211's password:
Connected to 10.80.127.211.
sftp> put /home/kali/.ssh/wrong_keyy.pub wrong_key.pub
Uploading /home/kali/.ssh/wrong_keyy.pub to /home/dut/wrong_key.pub
wrong_keyy.pub                                100%  171     3.0KB/s   00:00
sftp>
```

Evidence Screenshot - TC5_SSH_INCORRECT_PUBLIC_KEY_tester_024203_709591.png

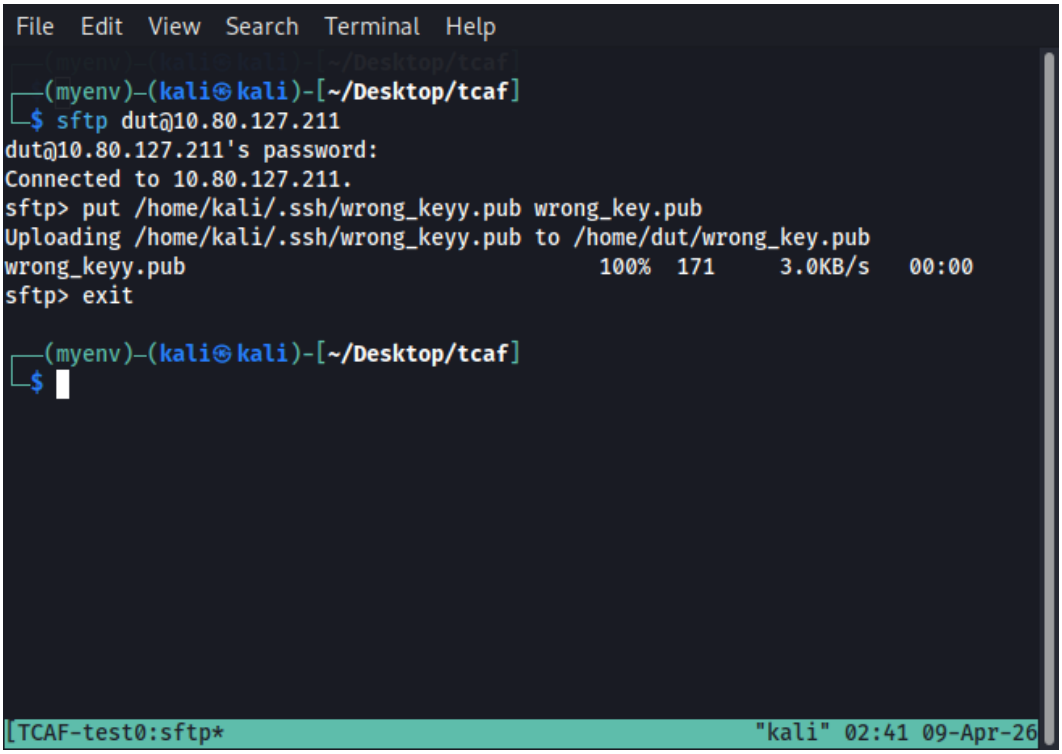


Figure 24: TC5 Step 2 — Unregistered public key uploaded to DUT filesystem (not added to authorized_keys)

Command Executed: ssh dut@10.80.127.211

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ ssh dut@10.80.127.211
dut@10.80.127.211's password:
```

Evidence Screenshot - TC5_SSH_INCORRECT_PUBLIC_KEY_tester_024231_146996.png

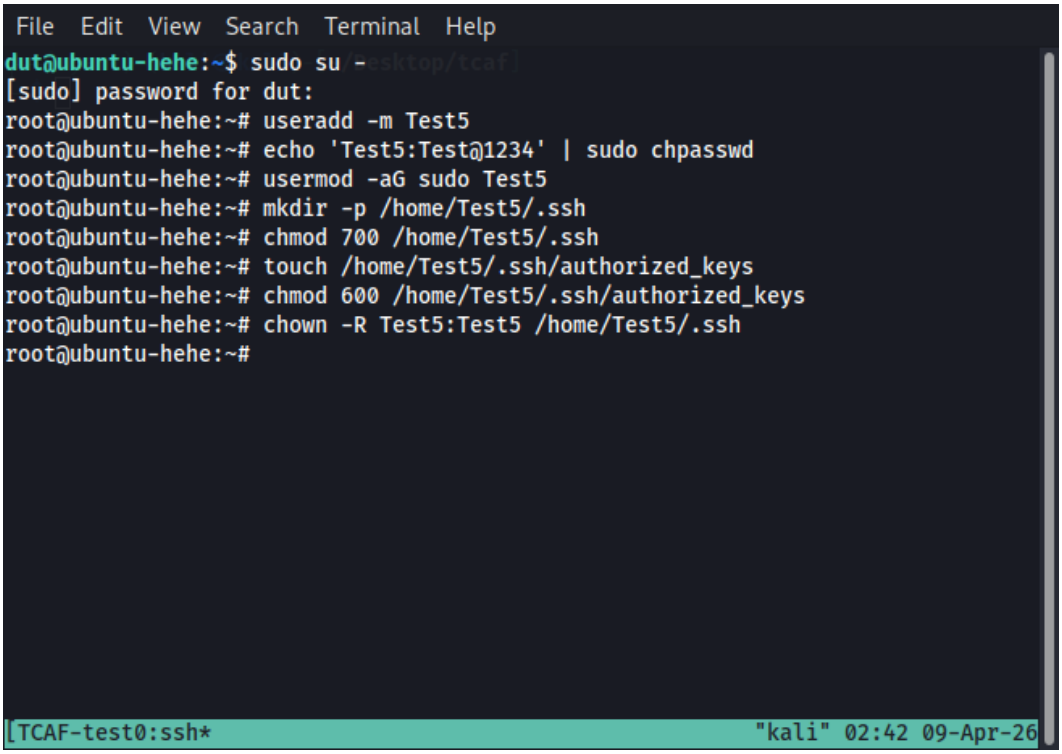


Figure 25: TC5 Step 3 — Test user created on DUT with empty authorized_keys; unregistered key will be rejected

Command Executed: ssh -o PasswordAuthentication=no -o IdentitiesOnly=yes -i ~/.ssh/wrong_keyy Test5@10.80.127.211

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ ssh -o PasswordAuthentication=no -o IdentitiesOnly=yes -i ~/.ssh/wrong_keyy
Test5@10.80.127.211
Test5@10.80.127.211: Permission denied (publickey,password).


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$
```

Evidence Screenshot - TC5_SSH_INCORRECT_PUBLIC_KEY_tester_024247_258730.png

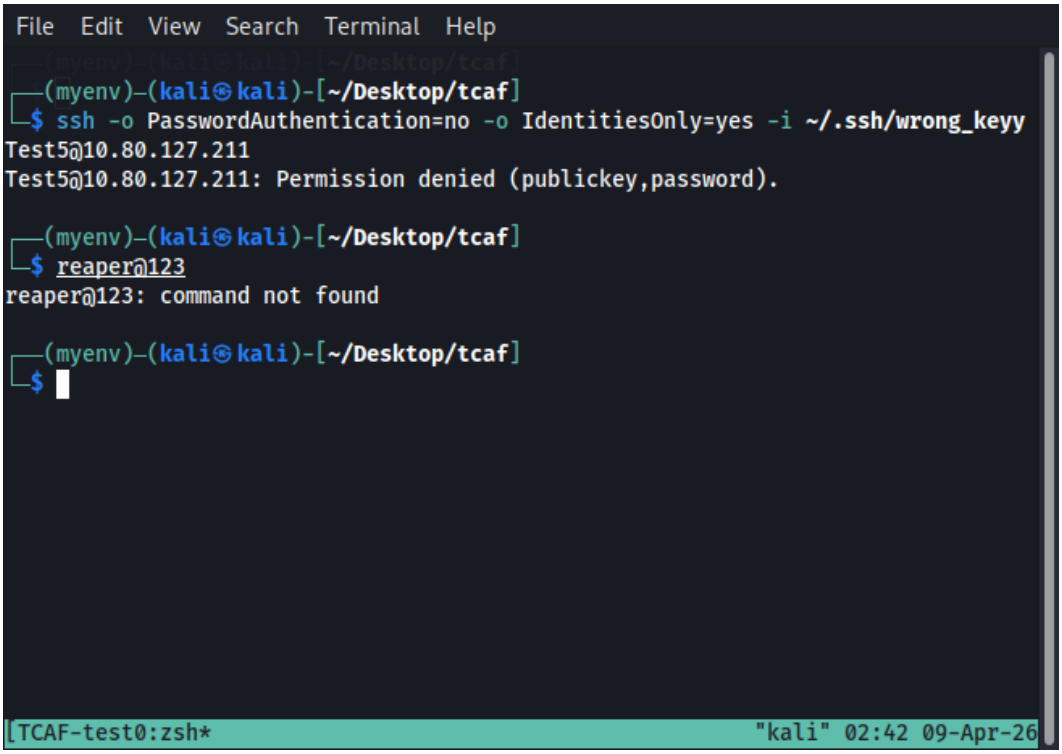


Figure 26: TC5 Step 4 — SSH login with unregistered public key rejected by DUT, permission denied

Command Executed: ssh dut@10.80.127.211

Command Output:

```
┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ ssh -o PasswordAuthentication=no -o IdentitiesOnly=yes -i ~/.ssh/wrong_keyy
Test5@10.80.127.211
Test5@10.80.127.211: Permission denied (publickey,password).


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ reaper@123
reaper@123: command not found


┌──(myenv)─(kali㉿kali)-[~/Desktop/tcaf]
└─$ ssh dut@10.80.127.211
dut@10.80.127.211's password:
```

Evidence Screenshot - TC5_SSH_INCORRECT_PUBLIC_KEY_tester_024300_416194.png

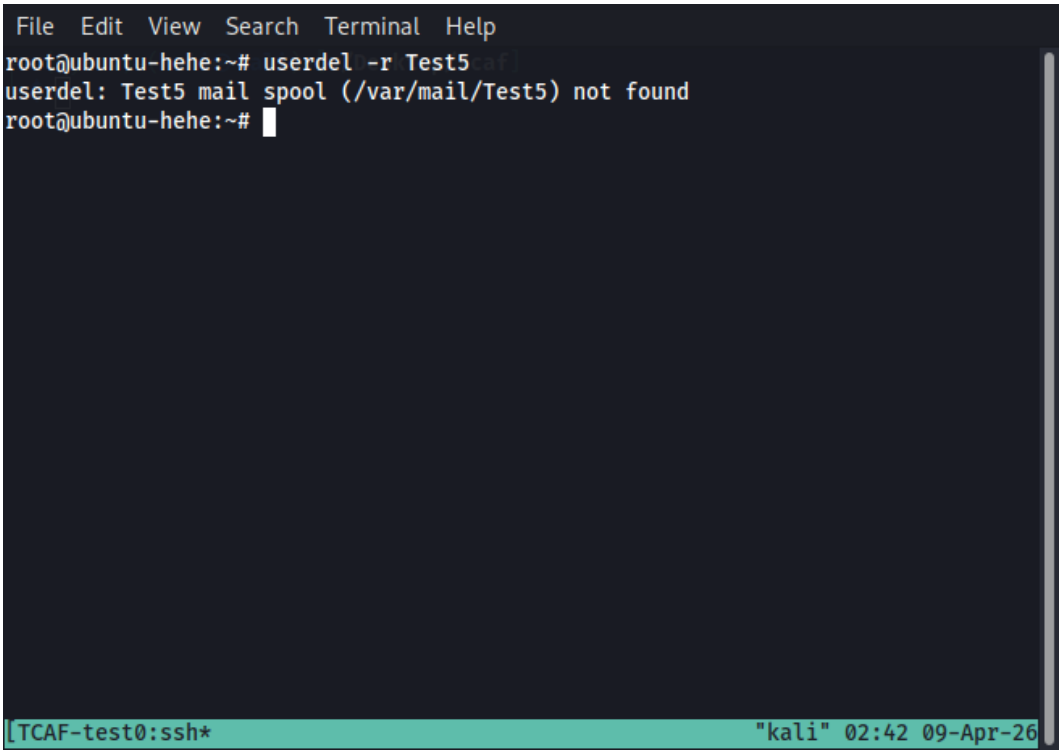


Figure 27: TC5 Teardown — Test user deleted from DUT after test completion

d. Test Observation:

The DUT correctly rejected the SSH login attempt made with an unregistered ECDSA key pair. The connection was terminated immediately with the message "Permission denied (publickey)", and no management session was granted. The DUT log buffer contained the entry "SSH user Test5 failed to pass public key authentication for wrong public key", confirming proper enforcement. Wireshark analysis verified that all traffic during the failed attempt remained fully encrypted.

e. Conclusion:

The DUT effectively blocks SSH access attempts using untrusted or unregistered public keys, demonstrating correct enforcement of public key mutual authentication - COMPLIANT with ITSAR 1.1.1.

f. Remark:

The DUT effectively blocks unauthorized SSH access attempts using invalid keys - COMPLIANT.

e. Evidence Provided:

Screenshots and command outputs are captured and attached during testing. Automated evidence is embedded above.

Test Result	PASS

6. Test Case Name: TC6_HTTPS_VALID_LOGIN

a. Test Case Description:

This test case verifies that the DUT correctly supports and enforces successful mutual authentication (client certificate authentication) during an HTTPS connection. Valid admin credentials shall grant access to the web management dashboard over TLS 1.2/1.3.

b. Execution Steps:

- Open web browser and navigate to the DUT login page: https://10.80.127.211/
- Enter valid admin credentials - username: admin password: [REDACTED]
- Verify the management dashboard is displayed (System Information page confirming authenticated access).
- Capture HTTPS traffic in Wireshark; verify TLS Client Hello and Server Hello exchange.
- Confirm TLS version (1.2 or 1.3) and approved cipher suite from Wireshark capture.
- Verify that TLS 1.0 is rejected by the DUT: openssl s_client -connect 10.80.127.211:443 -tls1
- Verify that TLS 1.1 is rejected by the DUT: openssl s_client -connect 10.80.127.211:443 -tls1_1

c. Evidence Captured:

Evidence Screenshot - tc6_login_page.png

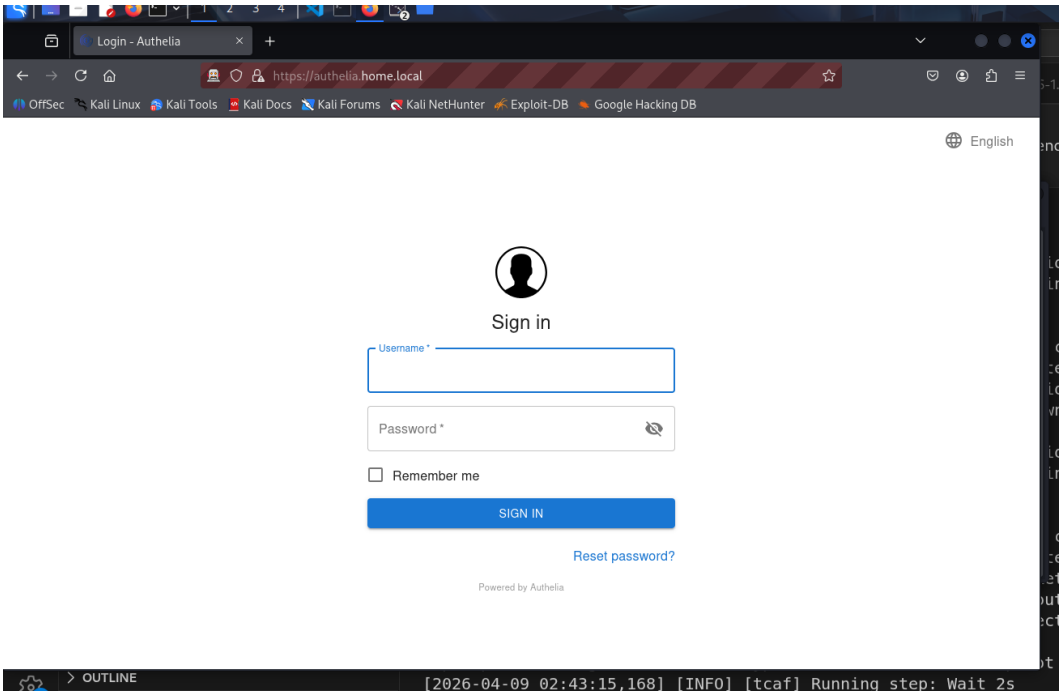


Figure 28: TC6 Step 1 — DUT HTTPS management login page loaded, TLS connection established

Evidence Screenshot - tc6_after_login.png

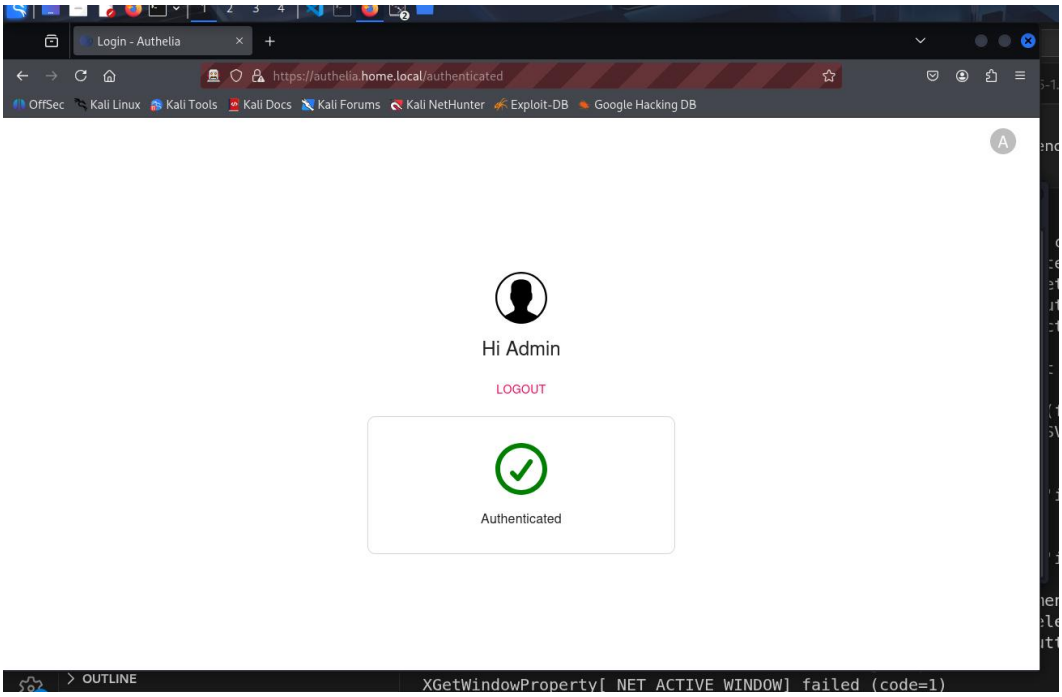


Figure 29: TC6 Step 2 — Dashboard accessible after valid credentials submitted, confirming successful HTTPS mutual authentication

Evidence Screenshot - packet_frame_4.png

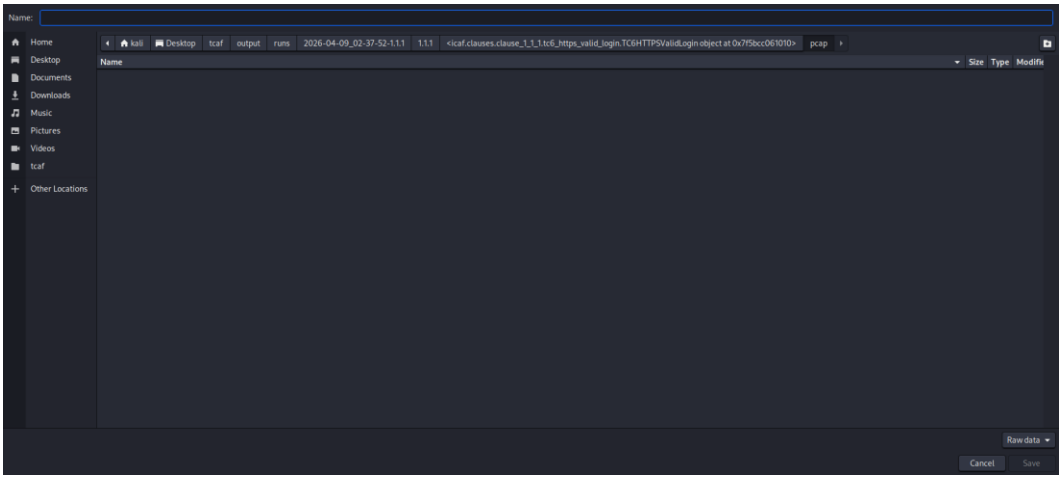


Figure 30: TC6 Step 2 — Wireshark shows TLS handshake completed with valid certificate exchange for HTTPS session

d. Test Observation:

The DUT successfully established a secure HTTPS management session upon submission of valid admin credentials. The web management dashboard (System Information page) was displayed, confirming authenticated access. Wireshark capture of the TLS handshake confirmed that both client and server traffic were encrypted, and that the TLS session was negotiated using TLS 1.2 or 1.3 with an approved cipher suite, meeting ITSAR transport security requirements.

e. Conclusion:

HTTPS mutual authentication over TLS is correctly implemented on the DUT, granting management access only upon valid credential submission - COMPLIANT with ITSAR 1.1.1.

f. Remark:

The DUT provides secure HTTPS access with encrypted transport and proper authentication enforcement - COMPLIANT.

e. Evidence Provided:

Screenshots and command outputs are captured and attached during testing. Automated evidence is embedded above.

Test Result	**PASS**

7. Test Case Name: TC7_HTTPS_INVALID_LOGIN

a. Test Case Description:

This test case verifies that the DUT does not allow HTTPS management access when incorrect admin credentials are entered. The browser must remain on the login page with an error message. No management data shall be disclosed. TLS encryption shall be maintained.

b. Execution Steps:

- Open web browser and navigate to: https://10.80.127.211/
- Enter incorrect credentials - username: admin password: [REDACTED]
- Observe DUT displays 'Failed to log in' error and no management access is granted.
- Capture HTTPS traffic in Wireshark; confirm TLS encryption is maintained throughout the failed authentication exchange.

c. Evidence Captured:

Evidence Screenshot - tc7_login_page.png

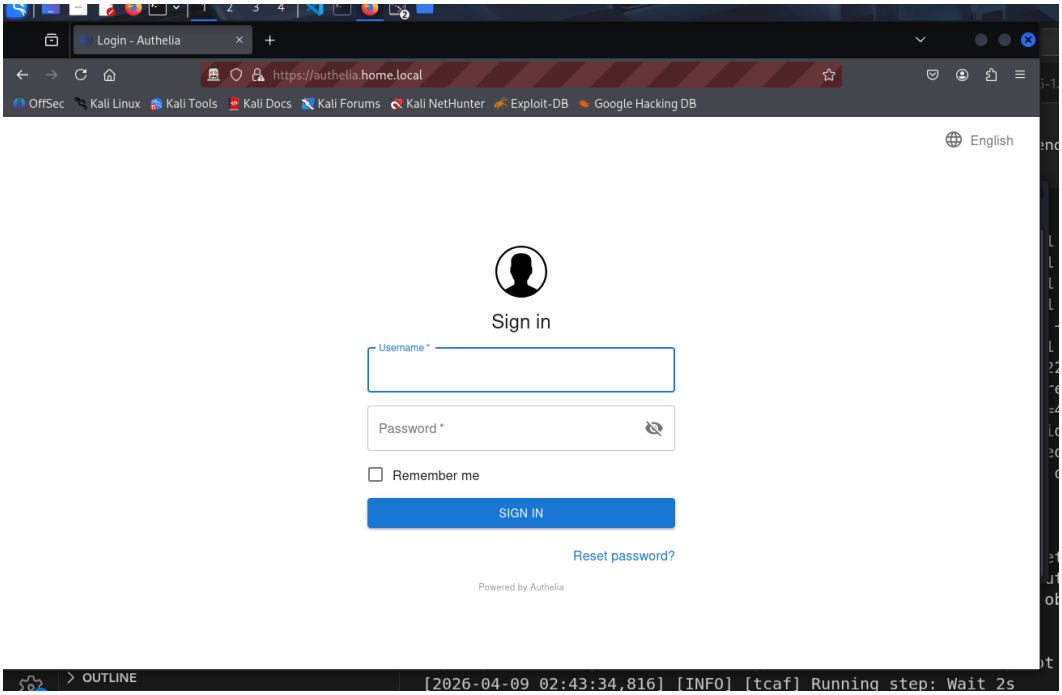


Figure 31: TC7 Step 1 — DUT HTTPS management login page loaded prior to invalid credential attempt

Evidence Screenshot - tc7_after_bad_login.png

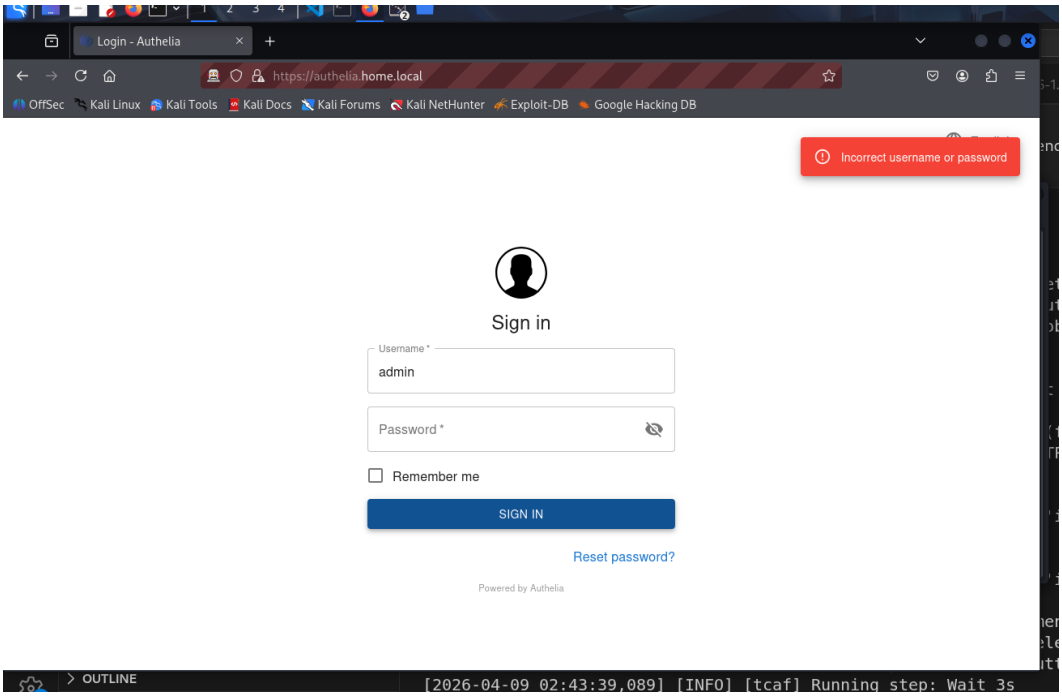


Figure 32: TC7 Step 2 — Login error displayed after invalid credentials submitted, dashboard not accessible

Evidence Screenshot - packet_frame_1.png

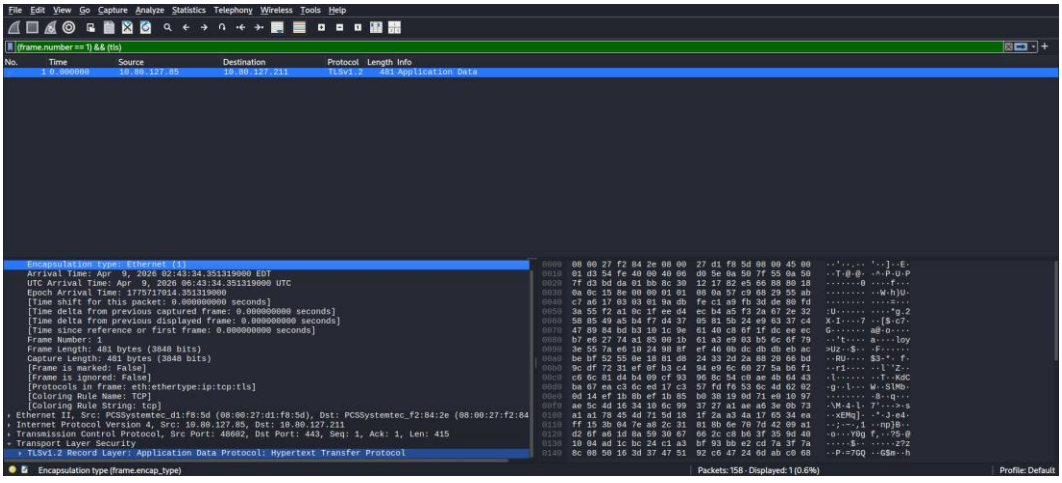


Figure 33: TC7 Step 2 — Wireshark shows TLS session terminated after HTTP 401/403 response from DUT

d. Test Observation:

HTTPS access was correctly denied when incorrect credentials were submitted to the DUT management web interface. The browser remained on the login page and displayed the "Failed to log in" error message, confirming

that no management access was granted. Wireshark packet capture verified that all traffic between client and server was TLS-encrypted throughout the failed authentication attempt, and no sensitive device data was disclosed.

e. Conclusion:

The DUT correctly rejects HTTPS management access on invalid credentials, maintaining TLS encryption and not exposing sensitive information - COMPLIANT with ITSAR 1.1.1.

f. Remark:

Unauthorized HTTPS access attempts are properly blocked without compromising data confidentiality - COMPLIANT.

e. Evidence Provided:

Screenshots and command outputs are captured and attached during testing. Automated evidence is embedded above.

Test Result	PASS

8. Test Case Name: TC8_GRPC_GNMI_MUTUAL_AUTH

a. Test Case Description:

This test case verifies that the DUT supports and enforces mutual authentication over gRPC/gNMI communication. The DUT shall establish a TLS-protected session only when a trusted CA certificate, valid client certificate, and correct user credentials are provided. A Token-ID shall be returned on successful authentication. Invalid credentials must be rejected without issuing a token.

b. Execution Steps:

- Generate CA key/cert, gRPC client key/cert, and PKCS#12 bundle on tester (CN=10.80.127.211) using OpenSSL.
- Transfer grpc.p12 and ca.pem to DUT: sftp dut@10.80.127.211

- Import CA certificate into DUT PKI domain: pki import domain grpc_pki pem ca filename ca.pem
- Import client PKCS#12 into DUT PKI domain: pki import domain grpc_pki p12 local filename grpc.p12
- Verify certificate installation: display pki certificate domain grpc_pki local
- Enable gRPC on DUT: grpc pki domain grpc_pki → grpc enable → grpc port 50051
- Create DUT user Test1 with password [REDACTED] and network-admin role.
- Positive - run grpcurl with valid credentials; verify token_id is returned: grpcurl -insecure -cert grpc.crt -key grpc.key -cacert ca.pem -d '{"username":"Test1","password":"[REDACTED]"}' 10.80.127.211:50051 gnmi.gNMI/Login
- Capture gRPC traffic in Wireshark; verify TLS handshake and encrypted application data.
- Negative - run grpcurl with incorrect password; verify no token_id is returned: grpcurl -insecure -cert grpc.crt -key grpc.key -cacert ca.pem -d '{"username":"Test1","password":"[REDACTED]"}' 10.80.127.211:50051 gnmi.gNMI/Login
- Capture negative-case traffic in Wireshark; confirm TLS encryption is maintained throughout.

d. Test Observation:

This test case was not executed because the required protocol is not present on the DUT. Protocol 'grpc' was not detected on the DUT during OAM verification — test case does not apply to the evaluated device.

e. Conclusion:

Not applicable — the protocol required for TC8_GRPC_GNMI_MUTUAL_AUTH is not supported by the evaluated device. No compliance determination is made for this test case.

f. Remark:

Protocol not present on DUT. Protocol 'grpc' was not detected on the DUT during OAM verification — test case does not apply to the evaluated device. This test case is excluded from the pass/fail count and does not affect the overall evaluation result.

e. Evidence Provided:

Screenshots and command outputs are captured and attached during testing. Automated evidence is embedded above.

Test Result	NOT_APPLICABLE

11. Test Case Result Summary

SL No.	Test Case Name	Result	Remarks
1	TC1_SNMPV3_POSITIVE	FAIL	SNMPv1 and/or SNMPv2c are supported on the DUT - insecure protocol versions must be disabled to achieve compliance.
2	TC2_SNMPV3_INVALID_CREDENTIALS	PASS	Unauthorized SNMP access attempts are correctly detected and denied by the DUT - COMPLIANT.
3	TC3_SSH_MUTUAL_AUTH	PASS	The DUT supports secure SSH mutual authentication and prevents information leakage during failed authentication attempts - COMPLIANT.
4	TC4_SSH_CORRECT_PUBLIC_KEY	PASS	Public key-based SSH authentication is correctly supported and enforced by the DUT - COMPLIANT.
5	TC5_SSH_INCORRECT_PUBLIC_KEY	PASS	The DUT effectively blocks unauthorized SSH access attempts using invalid keys - COMPLIANT.
6	TC6_HTTPS_VALID_LOGIN	PASS	The DUT provides secure HTTPS access with encrypted transport and proper authentication enforcement - COMPLIANT.
7	TC7_HTTPS_INVALID_LOGIN	PASS	Unauthorized HTTPS access attempts are properly blocked without compromising data confidentiality - COMPLIANT.
8	TC8_GRPC_GNMI_MUTUAL_AUTH	NOT_APPLICABLE	Protocol not present on DUT. Protocol 'grpc' was not detected on the DUT during OAM verification — test case does not apply to the evaluated device. This test case is excluded from the pass/fail count and does not affect the overall evaluation result.

	Total: 8 defined | 8 ran	**6P / 1F / 1NA**	**1 case(s) failed, 1 not applicable (protocol absent on DUT).**

12. Test Conclusion

- The DUT was successfully configured for SNMPv3 with authentication and privacy enabled. SNMPv1 and SNMPv2c were found to be disabled by default, which aligns with ITSAR security requirements. SNMPv3 communication was established only when correct credentials and approved cryptographic algorithms were used. Packet analysis confirmed SNMP traffic was encrypted and mutual authentication was enforced.
- The DUT correctly rejected SNMPv3 access attempts made with invalid credentials. SNMP walk operations returned authentication failure errors, and Wireshark captures confirmed traffic remained encrypted. No management access was granted and no sensitive information was disclosed.
- The DUT successfully supported SSH v2 mutual authentication when valid credentials were used. Secure key exchange and encrypted communication were verified through Wireshark, confirming the use of approved cryptographic algorithms. When incorrect credentials were provided, the DUT denied access and maintained encrypted communication without exposing any sensitive information.
- The DUT successfully authenticated the SSH client using public key-based authentication. The user was able to log in without a password only when the correct public key was configured. Logs and packet captures confirmed successful authentication and encrypted communication.
- The DUT correctly rejected SSH login attempts made with an incorrect public key. The connection was closed without granting access, and logs confirmed the authentication failure. Wireshark analysis showed that traffic remained encrypted throughout the attempt.
- The DUT successfully established secure HTTPS sessions using valid credentials. Wireshark captures confirmed TLS was used and all communication was encrypted. The DUT granted access only after proper authentication, demonstrating compliant HTTPS management access.
- The DUT rejected HTTPS login attempts made with incorrect credentials and returned appropriate error messages. Packet captures confirmed TLS encryption was maintained and no sensitive information was leaked.
- The DUT complies with the security requirement for mutual authentication over gRPC/gNMI. Successful authentication was achieved only with valid certificates and credentials, while invalid attempts were effectively blocked.
- The DUT successfully rejects all connection attempts made using insecure protocols or invalid credentials, demonstrating compliance with ITSAR Section 1.1.1.
- NOTE: 1 test case(s) were not applicable because the required protocol was not detected on the DUT during OAM verification. These cases are excluded from the pass/fail determination and do not affect the overall evaluation result.

Overall Evaluation Result: FAIL	6 passed, 1 failed, 1 not applicable (protocol absent on DUT).