



POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Threat-TLS: An Intrusion Detection and Monitoring Tool for Mitigating TLS Attacks

Relatore

Dr. Ing. Diana Berbecaru

Kevin GJEKA

ANNO ACCADEMICO 2023-2024

Summary

This thesis explores the Transport Layer Security (TLS) protocol, a widely-used cryptographic protocol designed for secure communication between two parties by exchanging parameters such as cipher suites, keys, and certificates. Originally introduced as SSL 2.0 in 1995, the protocol has evolved to address emerging vulnerabilities, culminating in the current version, TLS 1.3, released in 2018.

Despite these advancements, TLS remains susceptible to vulnerabilities due to specific implementations or misconfigurations. Existing tools, such as Qualys' SSL Server Test, TLSAssistant, and TLSAttacker, can assess a host's TLS security at a specific point in time, but lack the capability for continuous monitoring to detect changes that may expose systems to attacks. Factors like software updates, configuration changes, and internal threats can make systems vulnerable at any time. Studies have shown that many online recommendations for securing popular web servers such as Apache and Nginx are often inadequate, with high percentages of configurations relying on deprecated or insecure TLS versions and ciphers.

To address this gap, this thesis proposes a tool called Threat-TLS, an evolution of a past tool, designed to use Intrusion Detection Systems (IDS) such as Suricata and Zeek to monitor network traffic for TLS-related vulnerabilities in real time. Threat-TLS continuously analyzes TLS packet exchanges, inspecting parameters such as cipher suites, extensions, and certificates, to identify markers indicative of known vulnerabilities. The tool also verifies the validity of server certificates through mechanisms like Certificate Revocation Lists (CRL), Online Certificate Status Protocol (OCSP), and Certificate Transparency (CT). By integrating offensive tools such as Metasploit, Nmap, and TLS-Attacker, Threat-TLS can further validate potential vulnerabilities and raise alarms if real threats are detected. This work enhances the original tool's performance, accuracy, and scope. Specifically, the architecture has been optimized to improve performance, enabling the tool to handle multiple servers concurrently. The ruleset has been refined to enhance detection accuracy, and additional TLS attack vectors have been integrated to increase its monitoring capabilities. Extensive testing has been conducted to verify the improved performance and robustness of the tool, making it a more effective solution for detecting TLS vulnerabilities in real-time environments.

Contents

1	Introduction	8
1.1	Overview of TLS	8
1.1.1	Sessions and Connections in TLS	9
1.2	History of TLS: From SSL to TLS	10
1.3	TLS Protocol Structure	10
1.3.1	TLS Handshake Protocol	11
1.3.2	TLS Alert Protocol	11
1.3.3	TLS Record Protocol	12
1.4	TLS 1.3	13
1.5	Public Key Certificate (PKC)	15
1.5.1	Public Key Infrastructure (PKI)	15
1.6	X.509 Certificates	16
1.6.1	Certificate Revocation Mechanisms	16
1.6.2	Certificate Transparency (CT)	17
2	Motivation	18
2.1	TLS Scanners	18
2.1.1	SSL Labs	18
2.1.2	Downsides and challenges in detecting TLS vulnerabilities	19
2.2	Monitor for TLS Attacks	20
2.2.1	Architecture	20
2.2.2	Workflow	20
2.2.3	Criticalities and Limitations	21
2.2.4	Goal of this thesis	22
3	Background and Related Work	23
3.1	Attacks on the Handshake Protocol	23
3.1.1	Bleichenbacher Attack	23
3.1.2	Early ChangeCipherSpec Attack	25
3.2	Attacks on the Record Protocol	26
3.2.1	Padding Oracle Attack	26

3.2.2	POODLE Attack	27
3.2.3	Lucky13 Attack	28
3.2.4	Heartbleed Attack	29
3.2.5	BEAST Attack	29
3.3	Compression-Based Attacks	30
3.3.1	CRIME and BREACH Attacks	30
4	Tools and Technologies Used	32
4.1	Packet Sniffers	32
4.1.1	Wireshark	32
4.2	Intrusion Detection Systems (IDS)	33
4.2.1	Suricata	33
4.2.2	Zeek	34
4.3	Security Auditing tools	35
4.3.1	TLS-Attacker	35
4.3.2	O-Saft	36
4.3.3	Metasploit	37
4.3.4	Nmap	38
4.3.5	Testssl.sh	39
4.4	Vulnerability Scanners	39
4.4.1	OpenVAS	39
4.4.2	NIST NVD Database and API	40
5	Threat TLS	41
5.1	Architecture of Threat-TLS	41
5.1.1	Design Choices	42
5.1.2	Description of the Components	42
5.2	Threat-TLS high-level workflow	43
5.2.1	First Phase: Initialization	43
5.2.2	Second Phase: Monitoring and Task Creation	44
5.2.3	Third Phase: Task Execution	44
5.2.4	Fourth Phase: Visualization	45
5.3	Understanding how to write Rules/Scripts	47
5.3.1	Inspecting with Wireshark	48
5.3.2	Writing Suricata Rules	49
5.3.3	Writing Zeek Scripts	50
5.4	Implementation of the Rules/Scripts	51
5.4.1	Heartbleed	51
5.4.2	CRIME	53
5.4.3	BEAST	53

5.4.4	Logjam	54
5.4.5	Lucky13 and Padding Oracle	54
5.4.6	POODLE	55
5.4.7	RC4	55
5.4.8	Sweet32	55
5.4.9	TicketBleed	56
5.4.10	FREAK	56
5.4.11	Improvements Made to Zeek Scripts	57
6	Threat-TLS Implementation	58
6.1	Threat-TLS low-level workflow	58
6.1.1	Initialization	58
6.1.2	Monitoring and Task Creation	60
6.1.3	Task Execution	62
6.1.4	Visualization	65
7	Tests and Results	67
7.1	Testbed	67
7.2	Tests	68
7.2.1	Vulnerability Appendix	68
7.2.2	Limitations	69
7.2.3	Applicability Test	69
7.2.4	Performance Test	70
7.2.5	IDS Performance	73
7.2.6	The Importance of Running Multiple Tests	74
8	Conclusions	75
A	User's Manual	76
A.1	Requirements	76
A.2	Installation	76
A.2.1	Suricata Installation	76
A.2.2	Zeek Installation	77
A.2.3	Greenbone Community Edition (GVM) Installation	77
A.2.4	Metasploit Installation	78
A.2.5	Nmap Installation	78
A.2.6	TestSSL Installation	79
A.2.7	Threat-TLS Installation	79
A.3	How to Run the Tool	79
A.3.1	Setup Greenbone	79
A.3.2	Create a NVD API Key	81

A.3.3	Running the IDS	81
A.3.4	Running Threat-TLS	81
A.4	Testbed Installation	82
A.4.1	OpenSSL	82
A.4.2	Apache2	83
A.4.3	Nginx	83
B	Developer's Reference Guide	85
B.0.1	Suricata	85
B.0.2	Zeek	85
B.0.3	Nginx Settings	85
	Bibliography	87

Chapter 1

Introduction

Transport Layer Security (TLS) is crucial for securing internet communications, providing data confidentiality, authenticity, and integrity. TLS enables trusted connections for secure information exchange in applications like e-commerce and messaging. Its robustness is vital in today's digital world, where online services and sophisticated threats are prevalent.

TLS is a cryptographic protocol used for secure communications over TCP/IP networks, such as the internet. Maintaining confidential communication is essential, especially when sensitive information like passwords, banking details, or credit card data is transmitted. Such data, when sent over a network, is vulnerable to interception by malicious actors, who could gain access to confidential information. Therefore, it is necessary to adopt technologies that protect data from being understood by third parties and prevent any alterations during transmission.

Despite improvements like TLS 1.3, vulnerabilities persist due to implementation flaws or misconfigurations. This thesis aims to develop Threat-TLS, a tool that continuously monitors and mitigates TLS-related vulnerabilities. By leveraging network traffic analysis and offensive testing, Threat-TLS enhances system security and provides a comprehensive defense against evolving TLS threats.

1.1 Overview of TLS

TLS functions as an independent security layer above transport protocols like Transmission Control Protocol (TCP) to provide security for application protocols such as Hypertext Transfer Protocol (HTTP) and Internet Message Access Protocol (IMAP). Decoupled from specific applications, TLS ensures confidentiality, integrity, and authentication for data exchanged between two endpoints, typically a client and a server. To achieve this, it leverages cryptographic algorithms grouped into cipher suites, combining symmetric and asymmetric cryptography. The protocol negotiates these algorithms and suitable keys during the handshake phase, which establishes the secure connection. Authentication, particularly for the server side, is often facilitated using X.509 certificates.

Key TLS functions include:

- **Authentication:** All parties verify each other's identities, preventing malicious actors from impersonating legitimate entities and accessing sensitive information.
- **Integrity:** Mechanisms are in place to detect any modification of data during transfer, whether from transmission errors or deliberate tampering.
- **Confidentiality:** Data privacy is maintained so that only authorized participants can access and interpret the information exchanged in the communication.

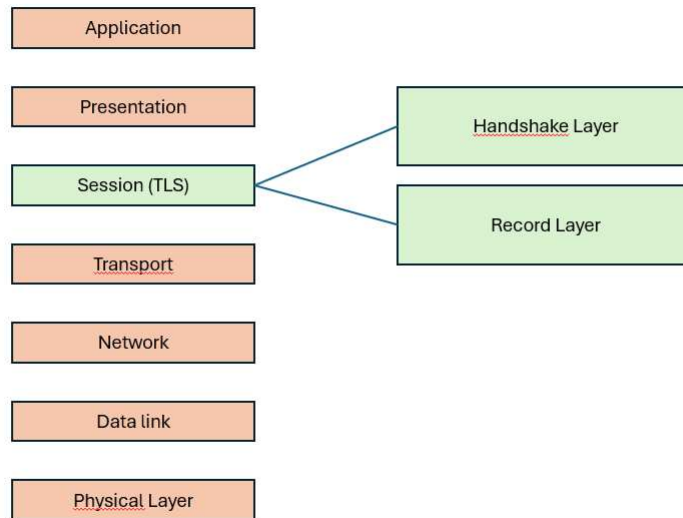


Figure 1.1: TLS in the OSI Model

1.1.1 Sessions and Connections in TLS

TLS distinguishes between two major concepts for secure communications - namely, **sessions** and **connections**. These concepts form the basis of how the TLS protocol maintains or establishes state and secure channels.

- **Sessions:** A *session* in TLS is a stateful object with the added advantage of avoiding the overhead for a full handshake every new connection. A session is established during the TLS handshake and may be reused in subsequent connections in an efficient manner to maintain secure communications between parties. A session is defined by the following parameters:
 - **Session identifier:** An arbitrary byte sequence containing the identifier for an active or resumable session state, chosen by the server.
 - **Peer certificate:** Identifies the node's peer certificate. It can either be null or of type X509.v3.
 - **Cipher spec:** Specifies the encryption and hash algorithm used for MAC calculation.
 - **Master secret:** A 48-byte secret shared between the client and the server.
 - **Is resumable:** A flag used to indicate whether the session can be used to initiate a new connection.
 - **Compression method:** Specifies the algorithm used to compress the data.

- **Connections:** A *connection* is a transport layer endpoint that provides a secure communication channel within a session. Each connection is assigned to a session, and it inherits its security parameters from the session. One session can have multiple parallel-in connections for parallel secure communications. Each connection is defined based on the following parameters:
 - **Server and client random:** Sequences of bytes chosen for each connection by the server and the client.
 - **Server and client MAC secret:** The keys used in MAC operations on data sent by the server and the client. The server uses a secret key, while the client uses a symmetric key.
 - **Server and client key:** The symmetric encryption keys used. The server key is used to encrypt data sent by the server and decrypt data by the client, and vice versa for the client key.
 - **Initialization Vector (IV):** When the CBC mode is used, an IV is maintained for each key.
 - **Sequence number:** A separate sequence number maintained by each party for transmitted and received messages for each connection. This helps prevent replay attacks.

Connections are established through the TLS handshake process, which negotiates the security parameters to be used for that particular connection. Once established, connections provide secure, reliable data transfer, with encryption and integrity checks applied to each message exchanged between the communicating parties.

1.2 History of TLS: From SSL to TLS

The Transport Layer Security (TLS) protocol evolved from the Secure Sockets Layer (SSL) protocol to address security vulnerabilities and improve compatibility. The Internet Engineering Task Force (IETF) introduced **TLS 1.0** in 1999 as RFC 2246[3], building on SSL 3.0 with security enhancements and structural changes that prevented interoperability with SSL. TLS 1.0 improved cryptographic security, extensibility, and efficiency by offering a framework for future cryptographic methods and an optional session caching mechanism.

TLS 1.1, released in 2006 as RFC 4346[4], addressed specific security issues in TLS 1.0, notably those related to the Cipher Block Chaining (CBC) mode. The addition of explicit initialization vectors (IVs) helped protect against attacks such as BEAST (Browser Exploit Against SSL/TLS), enhancing the protocol’s overall security.

TLS 1.2, introduced in 2008 with RFC 5246[5], brought substantial changes to improve flexibility and resilience. Key updates included replacing the MD5/SHA-1 pseudorandom function (PRF) with customizable PRFs, mandating the "TLS_RSA_WITH_AES_128_CBC_SHA" cipher suite, and adding HMAC-SHA256 for stronger message integrity. TLS 1.2 also deprecated weaker cipher suites and supported authenticated encryption with additional data (AEAD) modes to strengthen cryptographic robustness against emerging threats.

TLS 1.3, the latest version, was introduced in 2018 as RFC 8446[6] with a focus on simplifying the handshake process, improving security and performance. Key changes included the removal of outdated algorithms, the introduction of forward secrecy by default, protection against version downgrade attacks, and the adoption of probabilistic signature schemes (e.g., RSASSA-PSS) to improve security. Additionally, TLS 1.3 improved version negotiation through the *supported versions* extension in ClientHello and allowed clients to specify acceptable signature algorithms for X.509 certificates.

1.3 TLS Protocol Structure

The TLS protocol structure is composed of several sub-protocols, each serving a distinct role in establishing and maintaining secure communications

1.3.1 TLS Handshake Protocol

The TLS handshake protocol allows the client and server to establish a secure communication channel by negotiating cryptographic parameters, authenticating each other, and generating shared keys. This process determines the TLS version, encryption algorithms, and other settings before application data is exchanged.

The handshake messages have a common structure, including three fields:

1. **Type (1 byte):** Specifies the message type, like **ClientHello** or **ServerHello**.
2. **Length (3 bytes):** Indicates the message length.
3. **Content (≥ 0 bytes):** Contains parameters for the message.

Up to TLS 1.2, the handshake involves these main phases:

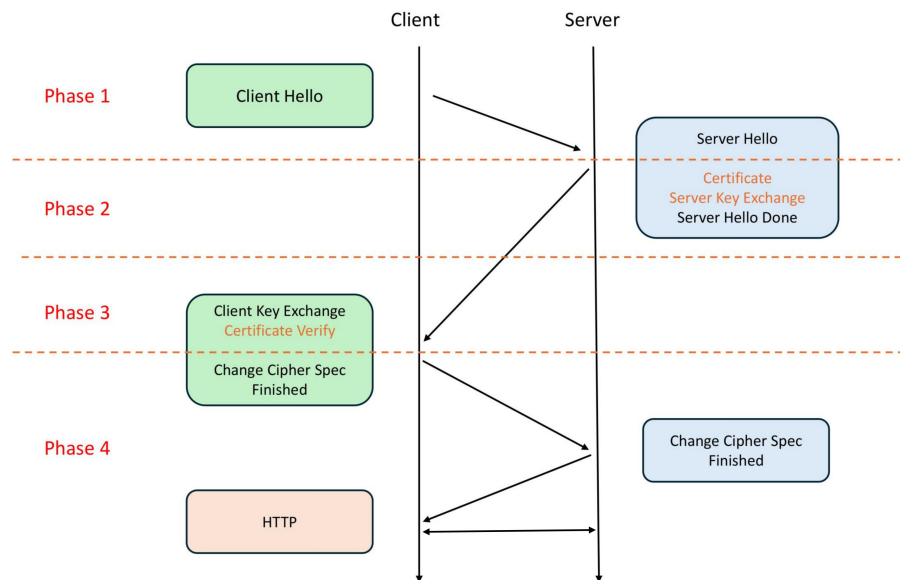


Figure 1.2: TLS Hanshake Structure

1. The client sends a **ClientHello** with its supported parameters. The server responds with a **ServerHello** message indicating chosen settings.
2. The server **optionally** sends a **Certificate** and a **ServerKeyExchange**, and a **ServerHelloDone**.
3. The client **optionally** verifies the server's certificate, sends **ClientKeyExchange** and, if needed, a **CertificateVerify**.
4. Both parties exchange **ChangeCipherSpec** and **Finished** messages to start secure data transmission.

1.3.2 TLS Alert Protocol

The TLS Alert Protocol communicates critical information about connection status, including warnings, errors, and termination signals, helping maintain a secure and synchronized session state between client and server.

Alerts in TLS are divided into closure alerts and error alerts. These messages are encrypted and compressed with session parameters to maintain confidentiality. In TLS 1.3, alert severity is inferred based on type, whereas previous versions used an explicit severity field.

Alert Types and Structure

Each alert message contains:

- **Content-Type:** Identifies the message as an alert.
- **Alert Level:** Specifies if the alert is a warning or fatal; fatal alerts terminate the connection.
- **Alert Description:** A code detailing the nature of the alert, like **unexpected_message** or **handshake_failure**.

Closure Alerts

Closure alerts signal the end of a secure connection. The **close_notify** alert informs the recipient that no further data will be sent, ensuring both parties are aware of session termination and reducing risks such as truncation attacks. In TLS 1.3, only one party is required to send **close_notify**. The **user_canceled** alert can also terminate a connection for non-protocol-related reasons, though it should be followed by **close_notify**.

Error Alerts

Error alerts indicate critical issues affecting the connection's security and are often fatal, requiring immediate termination. Key error alerts include:

- **unexpected_message:** An inappropriate message was received.
- **bad_record_mac:** A message failed the MAC check, suggesting tampering.
- **decryption_failed:** A decryption operation failed, possibly due to padding or key issues.
- **handshake_failure:** The handshake failed due to incompatible security parameters.
- **certificate-related alerts:** Issues like **bad_certificate** or **certificate_expired** indicate certificate problems.
- **unknown_ca:** The certificate chain's CA is untrusted.
- **protocol_version:** Protocol version mismatch.
- **internal_error:** An internal issue unrelated to protocol operations.

Fatal alerts require immediate termination, with all session keys and secrets discarded to prevent any reuse of potentially compromised data.

1.3.3 TLS Record Protocol

The TLS Record Protocol provides protection for application data using keys created during the handshake phase. It provides the protection, integrity, and origin verification of these records, managing the following functions:

- **Fragmentation and Reassembly:** Splits data into manageable blocks (up to bytes) and reassembles them upon receipt.
- **Compression and Decompression:** Messages can be optionally compressed; default in TLS 1.2 is no compression.
- **Encryption and Decryption:** Uses session keys for confidentiality.
- **Message Authentication Code (MAC):** Verifies integrity and authenticity of messages.

Record Types and Structure

The Record Protocol handles four content types:

- **Handshake:** Establishes security parameters.
- **Application Data:** Actual data exchanged after the handshake.
- **Alert:** Signals warnings or errors.
- **Change Cipher Spec:** Indicates changes in cryptographic parameters.

Records have a standardized format: Content-Type (1 byte), Version (2 bytes), Length (2 bytes), and Fragment (variable length).

Record Protection Mechanisms

The Record Protocol uses several protection mechanisms:

1. **Encryption:** Uses symmetric encryption (e.g., AES) for confidentiality.
2. **MAC:** Ensures integrity using a hash function and shared secret keys.
3. **AEAD:** In TLS 1.3, AEAD algorithms (e.g., AES-GCM) provide combined encryption and integrity protection.
4. **Padding:** Applies padding to obscure data length.

TLS 1.3 improves security by using AEAD, which combines encryption and integrity protection in one step.

1.4 TLS 1.3

TLS 1.3^[6], introduced by the IETF in 2018, aims to improve security, performance, and reduce handshake latency. It builds on TLS 1.2, addressing vulnerabilities from outdated algorithms and protocol features.

TLS 1.3 enhances security and efficiency through:

- **Reduced Handshake Latency:** TLS 1.3 uses a 1-RTT handshake, with support for 0-RTT for repeated connections, reducing latency.
- **Modern Algorithms Only:** Outdated cryptographic algorithms like static RSA are removed. Only secure key exchanges (e.g., ECDHE) and AEAD cipher suites (e.g., AES-GCM, ChaCha20-Poly1305) are supported.
- **Encrypted Handshake Messages:** Handshake messages are encrypted from the **Server-Hello** onwards, improving privacy.
- **Simplified Cipher Suites:** Uses AEAD exclusively, simplifying negotiation and enhancing security.
- **Improved Key Derivation:** Uses HKDF for better key separation and security.
- **Forward Secrecy by Default:** All key exchanges provide forward secrecy, protecting past session data.

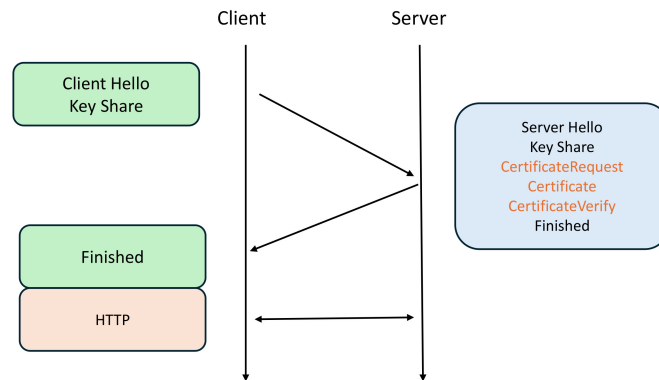


Figure 1.3: TLS 1.3 Handshake Structure

TLS 1.3 Handshake Protocol

The TLS 1.3 handshake is redesigned for lower latency and improved security:

1. The client sends a **ClientHello** with the supported ciphersuites and, compared to older versions, a **Key Share** message; the server responds with a **ServerHello** and its **Key Share** message. This allows to initiate the key exchange immediately thus resulting in 1-RTT.
2. The server sends **EncryptedExtensions** with additional data, and may send **CertificateRequest**, **Certificate** and **CertificateVerify** for authentication.
3. Both parties verify credentials and exchange **Finished** messages, after which secure data transmission can begin.

TLS 1.3 supports optional 0-RTT data for faster reconnections, though it requires careful implementation to prevent replay attacks.

Removed Features in TLS 1.3

TLS 1.3 has removed several insecure features:

- **Compression:** Removed due to vulnerabilities like CRIME.
- **Insecure Hash Functions:** MD5 and SHA-224 are no longer used.
- **Renegotiation:** Removed to prevent man-in-the-middle attacks.
- **Static RSA and DH Key Exchange:** Removed in favor of ephemeral key exchanges for forward secrecy.

1.5 Public Key Certificate (PKC)

A Public Key Certificate (PKC) is a digital document that binds a public key to its owner, signed by a trusted Certificate Authority (CA). PKCs enable secure communication without prior key exchange, as any party can retrieve the public key from a trusted certificate repository. The CA's signature verifies the authenticity of both the public key and the owner.

PKCs should be:

- Readable by anyone, including the owner's public key and identity.
- Created, updated, or revoked only by the CA.
- Verifiable by any party using the CA's digital signature.
- Time-bound, including validity start and expiration dates.

1.5.1 Public Key Infrastructure (PKI)

Public Key Infrastructure (PKI) manages the lifecycle of digital certificates, involving entities like End Entities, Certificate Authorities (CA), Registration Authorities (RA), and Repositories. The CA issues and revokes certificates, the RA verifies identity, and the repository stores certificates for retrieval.

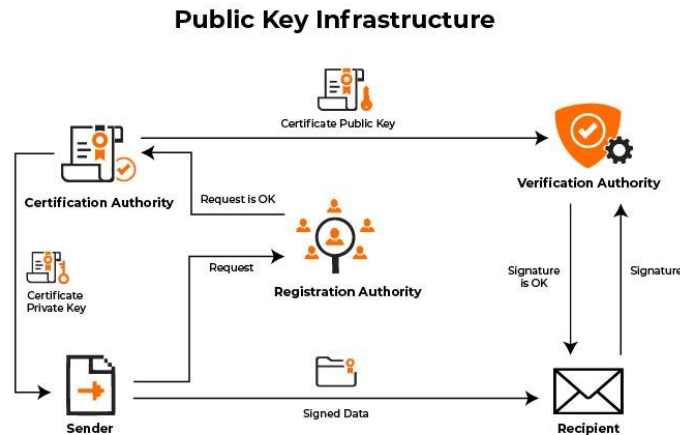


Figure 1.4: Public Key Infrastructure [10]

Certificate Generation Process

The process involves:

1. The end entity generates a public-private key pair and submits the public key with identification.
2. The RA verifies identity, then informs the CA.
3. The CA creates, signs, and issues the certificate, publishing it for others to access.

PKI Structures

Common PKI structures:

- **Hierarchical:** A Root CA issues certificates to subordinate CAs and end entities.
- **Mesh:** Multiple Root CAs cross-certify each other, creating flexible but complex trust relationships.
- **Bridge:** A Bridge CA connects multiple PKIs, reducing cross-certification complexity.

1.6 X.509 Certificates

X.509 is a standard for digital certificates used in protocols like SSL/TLS. It binds an entity's identity to a public key, verified by a CA's signature.

Structure of an X.509 Certificate

Key fields include:

- **Version:** Specifies the X.509 version.
- **Serial Number:** Unique identifier for the certificate.
- **Issuer/Subject Name:** Identifies the CA and certificate owner.
- **Validity Period:** Certificate's start and expiration dates.
- **Public Key Information:** Contains the public key and algorithm identifier.
- **Extensions:** Provide additional functionality, such as Key Usage or Subject Alternative Name (SAN).
- **Signature:** Digital signature of the CA.

Certificate Validation Process

Validation involves:

1. Verifying the CA's signature on the certificate.
2. Checking certificate fields and extensions.
3. Confirming revocation status via CRL or OCSP.
4. Validating the certificate chain to the Root CA.

1.6.1 Certificate Revocation Mechanisms

Certificates can be revoked before their expiration if compromised. Two mechanisms exist for revocation:

- **Certificate Revocation List (CRL):** A CRL is a list of revoked certificates signed by the CA. It contains revoked certificates' serial numbers, revocation dates, and metadata. CRLs can grow large, leading to inefficiencies.
- **Online Certificate Status Protocol (OCSP):** OCSP allows real-time certificate status checking, where a client queries an OCSP responder about a certificate's status, receiving a response indicating if it is **Good**, **Revoked**, or **Unknown**.

1.6.2 Certificate Transparency (CT)

Certificate Transparency (CT) is a public logging system for certificates to detect unauthorized issuance. It consists of public logs, monitors that watch for anomalies, and auditors that verify certificate presence in the logs.

Chapter 2

Motivation

Not all connections made using TLS are secure; many can be classified as weak. Such connections may be exploited by attackers to distribute malicious content, such as malware. Weak connections can also indicate compromised servers relying on outdated TLS versions or weak cryptographic algorithms. This chapter examines the limitations of traditional TLS scanners, the challenges in detecting TLS vulnerabilities, and the critical need for continuous, proactive monitoring to ensure system security.

2.1 TLS Scanners

TLS scanners are tools that assess the security configuration of SSL/TLS implementations by taking a snapshot of a server's current state. They help identify vulnerabilities, misconfigurations, and weaknesses in server SSL/TLS setups. Popular examples of TLS scanners are the SSL Server Test by Qualys SSL Labs and TLSAssistant.

2.1.1 SSL Labs

SSL Labs^[42] is a research project by Qualys that focuses on understanding SSL/TLS and the Public Key Infrastructure (PKI). One of its main tools is an online service that rates how well SSL servers are configured. The assessment looks at things like certificate validity, protocol support, key exchange mechanisms, and cipher strength. Results are given as both a numerical score and a letter grade, ranging from A+ (best) to F (worst).

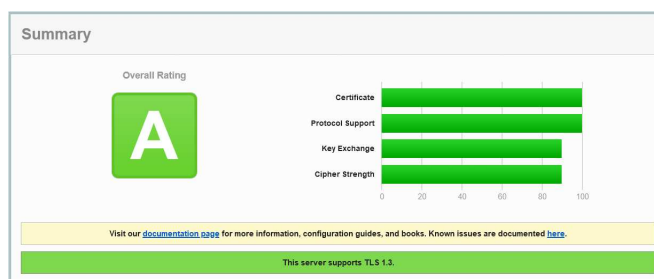


Figure 2.1: SSL Labs scan score for didattica.polito.it

The SSL Labs scoring is based on:

- **Certificate Inspection:** Evaluates the server's certificate for validity and trust. Issues like a domain name mismatch, expired certificates, or self-signed certificates result in a failing grade.

- **Protocol Support:** Checks which versions are supported, ranging from insecure SSL 2.0 to the most secure TLS 1.3.
- **Key Exchange and Cipher Strength:** Looks at the strength of the key exchange mechanism and symmetric ciphers used. Weak keys or ciphers lower the score.

The final grade considers scores from these categories, with penalties for known issues like the BEAST, CRIME, and POODLE attacks. Extra features like HTTP Strict Transport Security (HSTS) can boost the score to an A+.

Limitations

SSL Labs can't be run locally and it would be hard to integrate in a automation pipeline

2.1.2 Downsides and challenges in detecting TLS vulnerabilities

Traditional TLS scanners do a good job of showing a snapshot of a server's security, but they have limitations. They only assess security on demand, so they can't detect changes or new vulnerabilities in real time, especially in the case that the configuration of a server is changed. In this way, problems may go unnoticed between scans, leaving servers exposed to new vulnerabilities.

Some IDPS, like Suricata and Zeek, can do continuous monitoring and analyze network traffic by looking at certificates, TLS fingerprints, or handshake messages [32, 33], however, they may not detect more complex TLS threats related to the detailed mechanics of TLS attacks.

Many frameworks also require cryptographic security without offering ways to verify compliance. For example, The eIDAS Network, which facilitates cross-border electronic identification within the EU, mandates the use of TLS 1.3 for secure communications between nodes. This requirement is outlined in the eIDAS Cryptographic Requirements document, version 1.4.1, endorsed by the Cooperation Network on September 9, 2024 [11].

However, the eIDAS framework does not specify a centralized mechanism for monitoring compliance with this TLS 1.3 requirement. Instead, it is the responsibility of individual Member States and their respective node operators to adhere to these security protocols. This decentralized approach means that each operator must implement their own monitoring and compliance verification processes to uphold the required security standards.

How could we improve this?

To address these challenges, a real-time monitoring tool for TLS connections could be implemented. This monitoring tool should:

- **Detect Vulnerable TLS Connections:** Look at data exchanged during the TLS handshake, including protocol versions, cipher suites, and certificates, to find weak or suspicious connections.
- **Identify Attack Patterns:** Detect more sophisticated threats which traditional Intrusion Detection and Prevention Systems (IDPS) might miss.
- **Integrate Platform Information:** Combine TLS alerts with platform-specific information to detect vulnerabilities linked to particular software versions or implementations.

2.2 Monitor for TLS Attacks

The tool "Monitor for TLS Attacks", developed by Giuseppe Petraglia and presented at the IEEE 20th Consumer Communications & Networking Conference (CCNC) in Las Vegas [12], aimed to detect anomalies in a server's TLS connections and assess whether the server is vulnerable to related attacks. This tool was developed with the goal of solving the problems we found when analyzing current TLS scanning solutions.

2.2.1 Architecture

The tool consists of three main components:

1. **IDS:** Zeek and Suricata are the IDS in charge of detecting the anomalies.
2. **Core:** This is the central unit of the tool in charge of processing the vulnerability alerts and starting the attack tools
3. **Attack Module:** This module executes TLS attack tools against the target server to verify vulnerabilities. Results are logged in a dedicated directory named after the tested IP address.

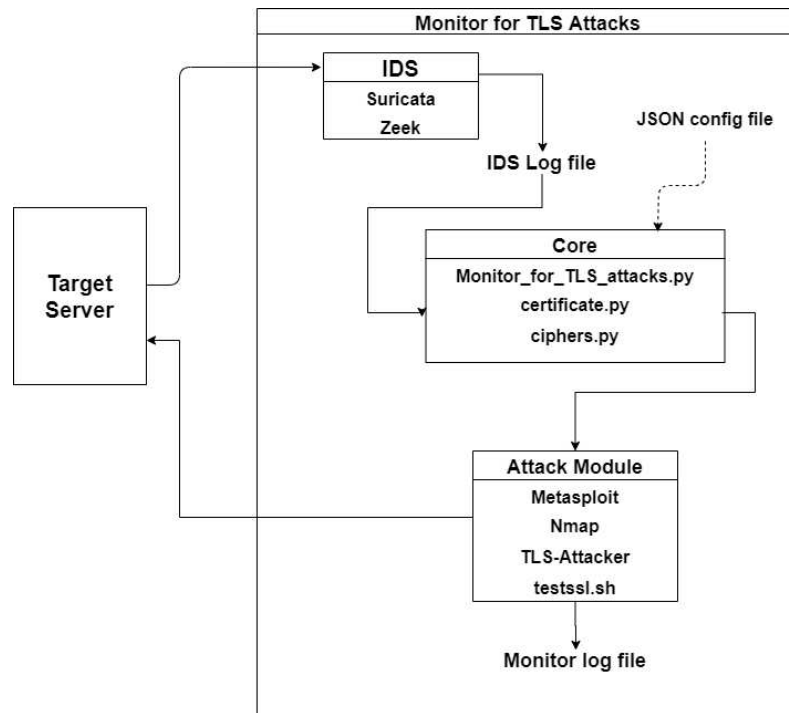


Figure 2.2: Architecture of the Monitor for TLS Attacks Tool

2.2.2 Workflow

At a high level, the tool operates on a Producer-Consumer multi-threaded pattern, leveraging a shared dictionary called `vuln_conn` used by the two threads.

The producer thread is represented by a Watchdog object, which monitors an Handler object. Upon initialization, the Handler object sets its logfile variable to the path of the corresponding Intrusion Detection System (IDS) log file. The IDS, either Zeek or Suricata, intercepts network traffic and writes results to the log file. When changes to the log file are detected, the Watchdog invokes a `file_reader` function that is in charge of extracting data from the alert and populating with key-value pairs, where:

- **Key:** The sniffed IP address.
- **Value:** An array containing vulnerabilities associated with that connection.

The consumer thread monitors `vuln_conn` for new items. When a new entry is detected, the consumer wakes up, removes the item from `vuln_conn`, and initiates the corresponding attack tool (e.g., nmap) against the IP address represented by the key. The results of these targeted tests are then logged in server-specific folders for further analysis.

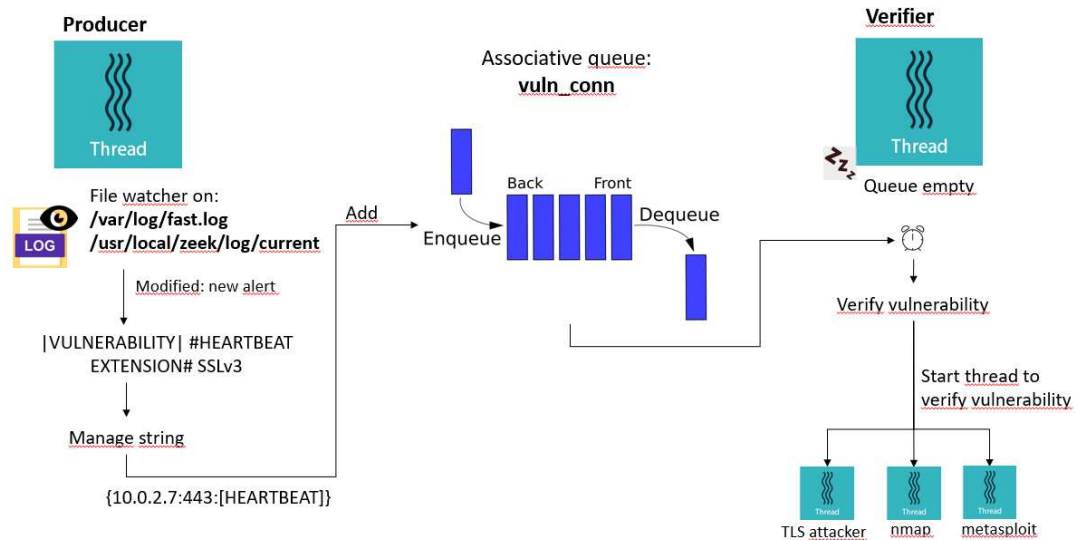


Figure 2.3: Monitor for TLS Attacks Workflow

2.2.3 Criticalities and Limitations

Monitor for TLS Attacks presents various advantages when compared to traditional scanning tools, it allows for a continuous monitoring of the network reducing the window of vulnerability of the servers and aiding the work of system administrators. Despite this, the original work presents some limitations:

Performance Limitations

The original implementation has some performance overhead when monitoring numerous TLS connections. This is partly due to the computational demand of launching a new thread for every attack but also stems from the unoptimized threading architecture.

Additionally, the tool does not verify whether a potentially vulnerable connection, as reported by the IDS, is relevant to the specific server being monitored before initiating a full test. Consequently, many tests are launched unnecessarily, returning negative results and wasting valuable time and resources.

Finally, the absence of caching exacerbates performance issues. Tests are repeatedly executed on the same target for the same vulnerability, leading to redundant thread spawning and resource wastage.

Rules/Scripts limitations

While most Suricata rules effectively check for correct patterns, some are incomplete, missing essential ciphers, while others either incorrectly flag certain ciphers or are overly generic. Similarly,

Zeek scripts tend to be excessively broad, generating alerts for every marker related to ciphers. This approach relies on the tool to handle cipher-specific checks, which leads to performance bottlenecks.

Additionally, some attacks that could be executed using available tools are missing from the ruleset/scripts, limiting the scope and effectiveness of the tool's detection capabilities.

Data Analysis Limitations

In the original implementation, data is stored in log files, making it challenging for users to analyze trends or derive meaningful statistics. This lack of data analysis tools hinders the ability to monitor and assess vulnerabilities effectively.

User Interface Limitations

The tool also suffers from a lack of a user-friendly interface. Users must manually navigate through numerous log files stored across different directories. This process becomes highly inefficient and time-consuming, particularly when monitoring multiple servers and connections.

2.2.4 Goal of this thesis

The main goal of this thesis is to render this tool more scalable and usable, to do so we will address the various areas using different strategies which will be discussed in the next chapters.

Chapter 3

Background and Related Work

In this chapter, we will examine several papers and articles about vulnerabilities and attacks that can compromise TLS security, which our tool is designed to detect. Understanding these attacks is essential for developing an effective monitoring and detection solution.

3.1 Attacks on the Handshake Protocol

The Handshake Protocol facilitates mutual authentication between client and server and negotiates encryption parameters. Attacks in this category often exploit weaknesses in the negotiation process, allowing attackers to downgrade the encryption level or interfere with key exchange

3.1.1 Bleichenbacher Attack

The Bleichenbacher attack [13], named after Daniel Bleichenbacher, is a chosen ciphertext attack against RSA encryption schemes using PKCS #1 v1.5 padding. Introduced in 1998, it exploits an oracle that reveals whether a given ciphertext yields a correctly formatted plaintext under PKCS #1 rules.

The attack targets implementations of RSA that provide feedback about the validity of padding, such as SSL V3.0 and early TLS versions. By adaptively querying the oracle, an attacker can gradually recover the plaintext without needing the private key.

A message m is padded as follows before encryption:

$$0x00\|\|\|0x02\|\|\|PS\|\|\|0x00\|\|\|m$$

where PS is the padding string, which must be at least 8 bytes long. The ciphertext c is computed by encrypting the padded message m^* :

$$c = (m^*)^e \bmod N$$

The Bleichenbacher attack proceeds through four phases:

1. **Blinding Phase:** The attacker computes a new ciphertext $c_A = c \cdot s_0^e \bmod N$, where s_0 is a randomly chosen integer. By querying the oracle, the attacker learns if the plaintext is PKCS #1 conforming, allowing initial bounds for the plaintext to be established.
2. **Range Reduction Phase:** The attacker reduces the range of possible plaintext values by iteratively using different multipliers s_i and querying the oracle. Each valid response updates the bounds, narrowing the possible plaintext range.

3. **Interval Narrowing Phase:** When the plaintext range is reduced to a single interval, small adjustments to s_i are used to further narrow the range until only one possible plaintext value remains.
4. **Solution Extraction Phase:** Finally, the attacker determines the exact plaintext m . Typically, this attack requires around one million oracle queries for a 1024-bit modulus, though fewer queries may suffice in practice.

The attack relies on an oracle O that takes a ciphertext c as input and returns whether the decrypted message has valid PKCS #1 v1.5 padding:

$$O(c) = \begin{cases} 1 & \text{if } c^d N \text{ has valid PKCS\#1 v1.5 padding} \\ 0 & \text{otherwise} \end{cases}$$

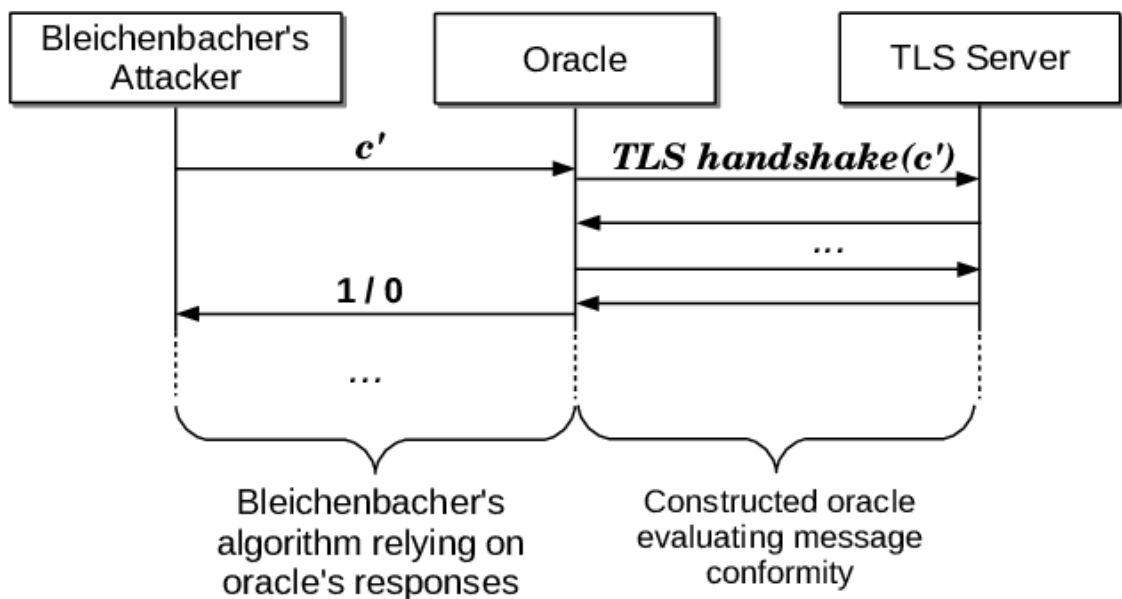


Figure 3.1: Bleichenbacher's attack algorithm [14]

ROBOT Attack

The Return Of Bleichenbacher's Oracle Threat (ROBOT) attack, discovered by Hanno Böck et al. in 2018, is a modern variant of the Bleichenbacher attack that demonstrated persistent vulnerabilities in many RSA implementations using PKCS #1 v1.5. The ROBOT attack leverages server behaviors such as differing error messages or timing responses to construct oracles, enabling decryption or signing with the server's private RSA key. This attack highlighted improper mitigations and affected products from major vendors like F5, Citrix, and Cisco [15].

Countermeasures

Effective countermeasures include:

1. **PKCS #1 v2.0 (OAEP):** Switching to Optimal Asymmetric Encryption Padding (OAEP) makes the scheme resistant to this attack.
2. **Diffie-Hellman Key Exchange:** Replacing RSA key exchange with Diffie-Hellman (DH) or Elliptic Curve Diffie-Hellman (ECDH) prevents padding vulnerabilities.
3. **Uniform Error Messages:** Ensure that decryption errors produce indistinguishable responses to mitigate this attack.

3.1.2 Early ChangeCipherSpec Attack

The Early ChangeCipherSpec (CCS) attack [24], disclosed in 2014 (CVE-2014-0224), exploits a flaw in OpenSSL's handling of the TLS handshake, specifically the sequencing of the ChangeCipherSpec (CCS) message. This allows an attacker in a man-in-the-middle (MITM) position to compromise the TLS session by injecting an early CCS message, leading to weak or predictable encryption keys.

The CCS message in TLS signals that subsequent messages should be encrypted with the negotiated cipher. Due to a flaw in OpenSSL, the CCS message could be processed too early, before the master secret was generated, causing encryption keys to be derived from empty or predictable values. This allowed an attacker to decrypt communication.

The attack works by injecting a CCS message immediately after the ServerHello but before the master secret is generated. This forces the server to derive encryption keys from an empty value, making the communication vulnerable. The vulnerability stemmed from improper validation of the handshake state, allowing CCS processing based on the cipher suite selection rather than the master secret generation.

In a vulnerable implementation, the following occurs:

- The server generates encryption keys using an empty master secret.
- The keys remain fixed and are not recalculated.
- The Finished hash is predictable, making it vulnerable to forgery.

In OpenSSL 1.0.1, changes in the Finished value calculation interacted poorly with the Early CCS flaw, allowing an attacker to align the Finished hashes and compromise the TLS session.

Mitigation Strategies

To mitigate the Early CCS attack:

1. **Patching OpenSSL:** Update to OpenSSL 1.0.1h or later to prevent early CCS processing.
2. **State Validation:** Ensure CCS messages are processed only after the master secret is generated.
3. **Session Integrity Verification:** Verify that the Finished message hash is based on a securely generated master secret.

3.2 Attacks on the Record Protocol

The Record Protocol is responsible for data encryption, fragmentation, and integrity. Attacks targeting this protocol generally aim to compromise data confidentiality or integrity within established TLS connections.

3.2.1 Padding Oracle Attack

The padding oracle attack, described by Serge Vaudenay [16], is a chosen ciphertext attack that exploits vulnerabilities in encryption schemes using Cipher Block Chaining (CBC) mode with padding. Specifically, it targets scenarios where the server reveals whether padding is valid through side channels or error messages, allowing an attacker to decrypt messages without the private key.

In CBC mode, each plaintext block P_i is XORed with the previous ciphertext block C_{i-1} before encryption using the block cipher E :

$$C_i = E(P_i \oplus C_{i-1}) \quad (3.1)$$

where C_0 is the initialization vector (IV). The last block is padded according to a padding scheme such as PKCS#7, to match the block size.

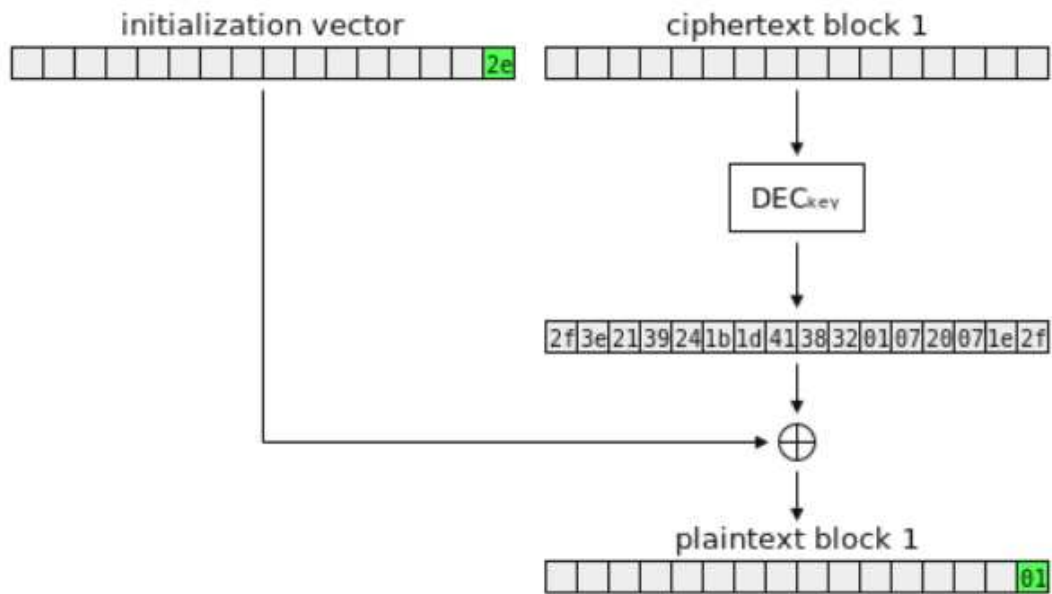


Figure 3.2: Padding Oracle correct guess [17]

The attack leverages the fact that, during decryption, if the padding is incorrect, the server often provides observable feedback. By manipulating the ciphertext and analyzing the server's responses, the attacker constructs an oracle that reveals information about the plaintext in a piecewise manner, exploiting the vulnerability with a computational complexity of approximately $O(NbW)$, where N is the number of ciphertext blocks, b is the block size, and W is the number of possible padding values (typically 256).

Countermeasures

1. **Use AEAD Ciphers:** Employ authenticated encryption (e.g., GCM or CCM) for confidentiality and integrity.

2. **Uniform Error Messages:** Ensure consistent error messages for all decryption failures.
3. **Encrypt-then-MAC:** Use encrypt-then-MAC to prevent padding errors from being detectable.
4. **Constant-Time Padding Verification:** Verify padding in constant time to avoid timing side channels.

3.2.2 POODLE Attack

The POODLE (Padding Oracle On Downgraded Legacy Encryption) attack [18], disclosed in 2014 (CVE-2014-3566), exploits a vulnerability in SSL 3.0 that allows an attacker in a man-in-the-middle (MITM) position to decrypt secure HTTP cookies and other sensitive information by forcing a downgrade from modern TLS protocols to SSL 3.0.

SSL 3.0, although largely replaced by TLS 1.0 and its successors, remains available in many systems for backward compatibility. Attackers can exploit the "downgrade dance" used by clients, where multiple versions of the protocol are attempted to ensure compatibility with legacy servers. By interfering with the handshake, an attacker can force the client and server to use SSL 3.0 instead of a more secure version.

In SSL 3.0, encryption is performed using either RC4 or a block cipher in Cipher Block Chaining (CBC) mode. The POODLE attack targets the latter, exploiting a flaw in how SSL 3.0 handles CBC padding. In CBC mode, the padding added to complete the final block is not covered by the Message Authentication Code (MAC), making it vulnerable to manipulation by an attacker. By replacing parts of the encrypted message, an attacker can trigger predictable padding errors and use them to gradually decrypt sensitive data, such as cookies.

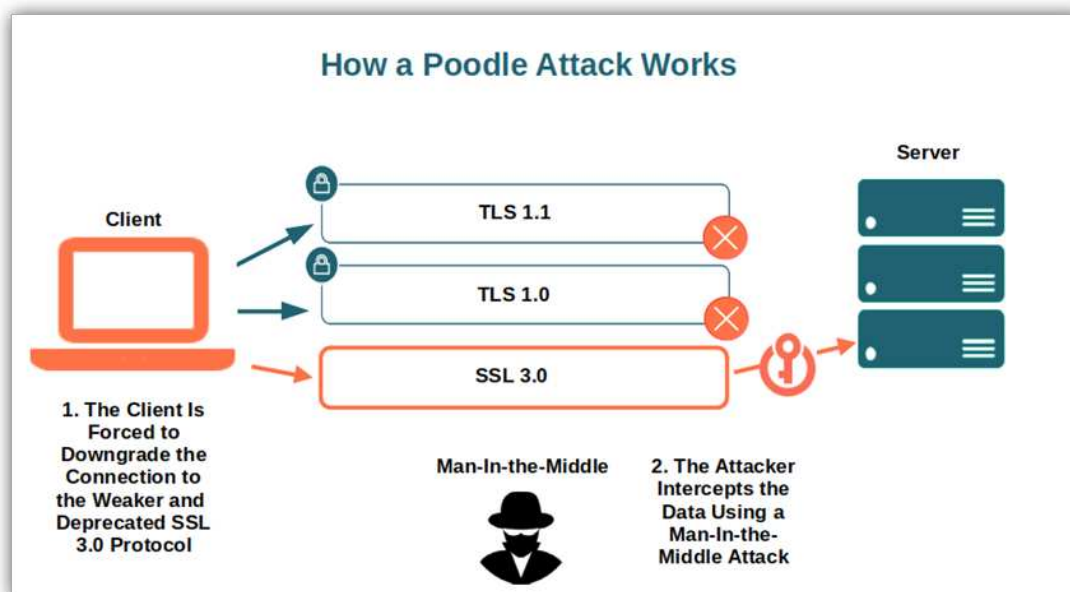


Figure 3.3: Poodle Attack[19]

The POODLE attack typically involves the following steps:

1. The attacker interferes with the connection to force a downgrade from TLS to SSL 3.0, making the communication susceptible to the vulnerabilities of SSL 3.0.
2. The attacker repeatedly modifies the ciphertext, using the server's responses to determine whether the modified padding is accepted. Each successful guess reveals information about the plaintext, enabling the attacker to decrypt the message byte by byte.

3. The process continues until the attacker has decrypted all the sensitive information, such as HTTP cookies, transmitted within the SSL session.

The expected effort for decrypting one byte of data is approximately 256 SSL 3.0 requests, which makes the attack practical under real-world conditions, particularly for long-lived sessions or automated scripts.

Countermeasures

1. **Disable SSL 3.0:** The most effective mitigation is to disable SSL 3.0 entirely on both clients and servers.
2. **Use TLS_FALLBACK_SCSV:** If disabling SSL 3.0 entirely is not feasible, the use of the TLS_FALLBACK_SCSV mechanism to prevent unintended protocol downgrades could be useful. This helps ensure that clients and servers will not fall back to SSL 3.0 when both support a higher version of the protocol.

3.2.3 Lucky13 Attack

The Lucky13 attack [20], introduced by Nadhem J. AlFardan and Kenneth G. Paterson in 2013, targets a vulnerability in the TLS and DTLS protocols when Cipher Block Chaining (CBC) mode encryption is used. This timing attack allows an adversary to potentially recover plaintext from encrypted communications by exploiting subtle differences in the time taken to perform decryption and MAC validation.

The Lucky13 attack takes advantage of the way that TLS and DTLS handle padding in CBC mode. In CBC encryption, the padding added to the plaintext must be removed during decryption, and the resulting plaintext must pass an integrity check using the Message Authentication Code (MAC). The attack leverages timing differences that arise during the MAC verification step if padding is incorrect. Specifically, TLS 1.1 and 1.2 recommend that if the padding is invalid, the MAC should still be computed on some data, assuming a zero-length padding. However, small variations in timing due to the processing of different amounts of data allow an attacker to distinguish between valid and invalid padding, thereby creating a timing side channel that can be used to recover plaintext.

The Lucky13 attack proceeds by repeatedly modifying and injecting ciphertexts, then measuring the response time to infer information about the plaintext. The attack is effective against implementations that do not properly mitigate timing variations during the padding removal and MAC verification stages. Experimental results have demonstrated the feasibility of Lucky13 in realistic environments, including against implementations in OpenSSL.

Mitigation Strategies

To mitigate the Lucky13 attack:

1. **Constant-Time Decryption:** Verify that padding removal, MAC validation, and other decryption operations are performed in constant time, irrespective of the input data, to eliminate timing differences that could be exploited by an attacker.
2. **Use Authenticated Encryption:** Replace CBC mode with authenticated encryption algorithms, such as Galois/Counter Mode (GCM) or ChaCha20-Poly1305, which provide built-in protection against padding and timing attacks by combining encryption and authentication in a secure manner.

3.2.4 Heartbleed Attack



The Heartbleed attack [22], disclosed in 2014 (CVE-2014-0160), is a vulnerability in OpenSSL's implementation of the TLS/DTLS heartbeat extension. This vulnerability allows an attacker to read sensitive information from the memory of a vulnerable server or client, which can include private keys, session cookies, and other sensitive data.

Heartbleed exploits the Heartbeat extension, which is designed to maintain active TLS sessions without renegotiation. During normal operation, the client sends a heartbeat request containing a payload and a specified length, and the server is supposed to respond by echoing back the data. However, in vulnerable versions of OpenSSL (1.0.1 through 1.0.1f), there was insufficient validation of the payload length. An attacker could send a malicious heartbeat request that claimed a larger payload size than what was actually sent. As a result, the server would respond with not only the original payload but also additional memory contents, potentially leaking sensitive information.

The Heartbleed attack can be summarized as follows:

1. The attacker sends a heartbeat request with a small payload but claims a much larger payload length.
2. The server, due to improper validation, reads and returns the requested length of memory, which includes sensitive information beyond the original payload.
3. Repeated requests allow the attacker to gather significant amounts of sensitive data from server memory, including private keys, user credentials, and session information.

One of the critical impacts of Heartbleed is that it allows attackers to recover private keys used for TLS, effectively breaking the encryption for ongoing and future sessions until the keys are replaced.

Mitigation Strategies

To mitigate the Heartbleed attack:

1. **Update OpenSSL:** Upgrade OpenSSL to version 1.0.1g or later, which contains a fix for the vulnerability by ensuring proper validation of heartbeat messages.

3.2.5 BEAST Attack

The BEAST (Browser Exploit Against SSL/TLS) attack [27], demonstrated in 2011 (CVE-2011-3389), is a client-side attack that exploits a vulnerability in the SSL 3.0 and TLS 1.0 protocols, specifically targeting the way Cipher Block Chaining (CBC) mode is used. The attack allows an attacker in a man-in-the-middle (MITM) position to decrypt secure HTTP cookies and other sensitive information.

The BEAST attack exploits a flaw in the way SSL/TLS handles CBC mode encryption. In CBC mode, each block of plaintext is XORed with the previous ciphertext block before encryption. The BEAST attack involves injecting chosen plaintext into the encrypted session to exploit predictable initialization vectors (IVs). By carefully crafting plaintext blocks and observing the server's responses, the attacker can decrypt the session data one block at a time.

The attack works as follows:

1. The attacker uses a network sniffer and a Java applet or JavaScript to control parts of the plaintext being encrypted by the client.

2. The attacker repeatedly sends plaintext blocks, using the predictable nature of the IVs to gain information about the encrypted data.
3. By performing multiple encryption attempts and analyzing the server's responses, the attacker gradually decrypts sensitive information, such as cookies, transmitted within the SSL/TLS session.

The success of the BEAST attack relies on the ability to inject malicious code into the victim's browser and control the flow of the encrypted session. This is often achieved through the use of Java applets or JavaScript to act as an "agent" that communicates with the attacker. Additionally, the attack requires the ability to send two SSL records in the same cookie-bearing request, leveraging a vulnerability known as "blockwise privilege."

Mitigation Strategies

To mitigate the BEAST attack:

1. **Upgrade to TLS 1.1 or TLS 1.2:** These versions of TLS address the vulnerability by changing how initialization vectors are generated, eliminating the predictability that the BEAST attack exploits.
2. **Use Stronger Ciphers:** Prefer authenticated encryption modes like Galois/Counter Mode (GCM) that are not vulnerable to the issues present in CBC mode.

3.3 Compression-Based Attacks

Compression attacks leverage TLS compression to leak sensitive information. These attacks measure the size of compressed data to infer content.

3.3.1 CRIME and BREACH Attacks

The CRIME (Compression Ratio Info-leak Made Easy) and BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) attacks [?] are both compression side-channel attacks targeting encrypted web traffic to recover sensitive information such as authentication cookies or CSRF tokens.

CRIME, presented by Thai Duong and Juliano Rizzo in 2012, exploits vulnerabilities in SSL/TLS compression. By injecting partial chosen plaintext into HTTP requests and observing the size of the resulting encrypted packets, an attacker can infer information about the plaintext. This attack led to the disabling of TLS/SSL-level compression in major web browsers, thereby mitigating its impact.

BREACH, introduced in 2013 by Yoel Gluck, Neal Harris, and Angelo Prado, is a successor to CRIME that exploits HTTP-level compression (such as gzip) rather than TLS-level compression. BREACH targets secrets such as CSRF tokens or session identifiers that are included in HTTP response bodies alongside user-reflected data, which allows the attacker to use the compression characteristics to recover the secret values. Even with TLS compression disabled, the use of HTTP-level compression makes web applications vulnerable to this attack.

The BREACH attack works as follows:

1. The attacker injects chosen plaintext into the victim's HTTP requests, typically using reflected user inputs such as URL parameters.
2. By analyzing the size of the HTTP responses, the attacker can infer whether the guess for the secret is correct, since correct guesses lead to better compression ratios due to repeated strings.
3. The attacker proceeds character by character, gradually recovering the secret by exploiting the compression side channel.

Mitigation Strategies

To mitigate CRIME and BREACH attacks:

1. **Disable Compression:** Disabling HTTP-level compression prevents the attacker from using size differences to infer secret information. However, this may have a negative impact on performance.

Chapter 4

Tools and Technologies Used

Before introducing the proposed work we will first analyze and discuss the tools and technologies used by the tool.

4.1 Packet Sniffers

A packet sniffer, also known as a packet analyzer or network analyzer, is a tool used to monitor network traffic. Packet sniffers capture streams of data packets moving through a network in real time, allowing administrators to examine the data being sent between computers on the network and to the larger internet. These tools can be used in two ways: in unfiltered mode, where all packets are captured for later examination, or in filtered mode, where only packets containing specific data are captured. Packet sniffers are useful for finding network problems, monitoring network usage, and identifying security issues. They can be used on both wired and wireless networks, although their effectiveness depends on network security settings and access to different network segments.

4.1.1 Wireshark



Wireshark is a widely-used, open-source packet analyzer that provides an in-depth view of network traffic. It allows users to capture live data from a network interface and analyze it in detail, facilitating tasks such as troubleshooting, protocol development, and educational purposes.

What can Wireshark be used for

- **Packet Capture:** Wireshark utilizes the `pcap` (packet capture) library to intercept data packets traveling over a network. It can capture traffic from various network types, including Ethernet, Wi-Fi, and loopback interfaces. By placing the network interface into promiscuous mode, Wireshark can capture all packets on the network segment, not just those addressed to the host machine.
- **Filtering:** Wireshark offers filtering capabilities to isolate specific traffic of interest. Display filters enable users to focus on particular protocols, IP addresses, ports, or other criteria.
- **Visualization:** Wireshark allows the user to visualize any packet in detail.

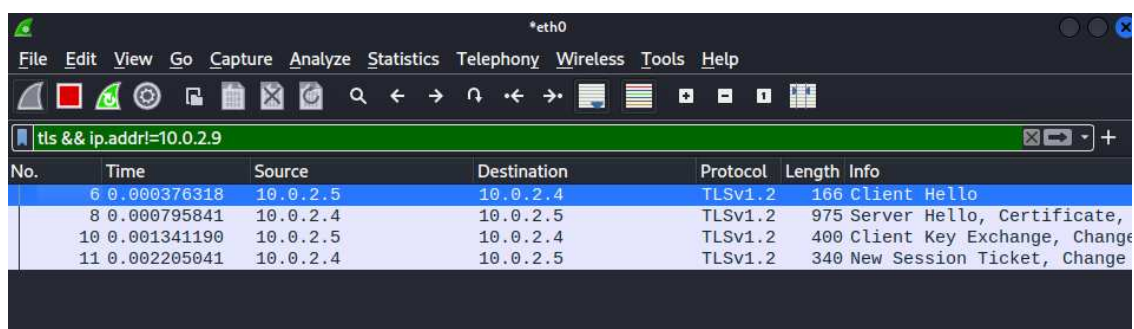


Figure 4.1: Example of Wireshark’s capabilities

4.2 Intrusion Detection Systems (IDS)

Intrusion Detection Systems (IDS) help keep computer networks safe by finding unauthorized access or misuse. Schell and Martin (2006) define intrusion as an action that breaks a system’s security. The main job of an IDS is to watch network traffic or system activities and alert administrators if it finds suspicious behavior. According to Rehman (2003), IDSs use rules and signatures to find unauthorized activity, providing timely alerts so that administrators can respond quickly. IDSs started in the 1980s as academic research, and the first commercial products appeared in the 1990s (Khan Pathan, 2014). Today, IDSs are an important part of network security, helping organizations find and respond to potential attacks before they cause serious problems.

Intrusion Detection Systems can be grouped into different types based on how they monitor and analyze. These types include:

- **Host-based IDS (HIDS):** These systems watch the activity on individual devices, looking at data like operating system logs, application activity, and system changes to find possible intrusions.
- **Network-based IDS (NIDS):** These systems look at network traffic to find suspicious patterns that might mean unauthorized access or attacks on the network. NIDS usually work at key points in a network to monitor all incoming and outgoing traffic.
- **Hybrid IDS:** Hybrid systems combine the features of both HIDS and NIDS, giving full coverage by watching both network traffic and host activity.

4.2.1 Suricata



Suricata is an open-source Network Intrusion Detection System (NIDS) created by the Open Information Security Foundation (OISF) in 2010. It is built with insights from earlier IDS solutions like Snort, and it can use Snort’s rules with minimal modifications, making it a flexible tool for integration into existing setups.

Suricata primarily uses signature-based detection to identify threats by matching network traffic against a library of known attack patterns. One of its key features is a multi-threaded architecture, allowing it to utilize multiple CPU cores for improved performance in high-bandwidth environments. This enables Suricata to efficiently process large volumes of network traffic and analyze multiple packets concurrently.

Suricata also includes deep packet inspection (DPI) and protocol analysis capabilities, enabling it to examine packet contents beyond just the headers to detect sophisticated attacks that may evade simpler detection mechanisms.

Suricata Rules

Suricata relies on rules to detect threats, similar to Snort's rule format. These rules define actions, such as creating alerts or dropping packets, when specific patterns are identified in the network traffic. Users can use predefined rules or create custom ones as needed. Suricata-Update is used to manage and update rulesets, with popular options like the Emerging Threats Open ruleset available for learning and usage.

A typical Suricata rule consists of three parts:

- **Action** - This tells Suricata what to do when a packet matches the rule.
 - **alert** - Create an alert.
 - **pass** - Stop checking the packet any further.
 - **drop** - Drop the packet and create an alert.
 - **reject** - Send an error message to the sender saying that the packet couldn't be delivered.
 - **rejectsrc** - Similar to *reject*, but sends the error only to the sender.
 - **rejectdst** - Sends an error message to the receiver of the packet.
 - **rejectboth** - Sends error messages to both the sender and receiver.
- **Header** - This specifies the protocol, IP addresses, ports, and direction of the traffic.
- **Options** - This contains specific details that must match for the rule to be triggered.

Example Suricata rule:

```
alert tls any any -> any any (msg:"RECORD LAYER +SSLv2+ SERVER HELLO
|VULNERABILITY| #TICKETBLEED#"; flow:from_server; content:"|16 00 02|";
content:"|02|"; distance:2; within:4; content:"|00 23|"; distance:0;
within:2; sid:884;)
```

In this rule:

- The **action** is *alert*, meaning Suricata will generate an alert if the rule matches.
- The **header** specifies TLS traffic from the internal network to any external network.
- The **options** define specific details, in this case bytes in certain positions, that must match for the rule to trigger.

4.2.2 Zeek



Zeek, formerly called Bro, was created by Vern Paxson in 1995. Unlike Snort and Suricata, Zeek is not just about signature-based detection. Instead, it uses anomaly-based detection and network traffic analysis to look for strange behaviors and gather detailed information.

Zeek was mostly developed through academic research, which makes it different from other IDS tools. It uses a powerful scripting language, allowing users to customize it to fit their needs. This makes Zeek very flexible and great for organizations that need a deep understanding of their network traffic.

Zeek is especially good for long-term analysis, as it collects detailed logs of network activities. This helps understand how an attack happened and spot unusual behaviors. However, configuring Zeek requires more expertise compared to traditional signature-based IDS tools.

Logging Framework of Zeek

Zeek’s logging framework provides a flexible, key-value based logging interface that allows users to control what gets logged and how it is logged. This framework revolves around three main abstractions: streams, filters, and writers.

- **Streams** - A log stream in Zeek represents a specific type of log, defining the fields it contains, such as connection summaries (conn stream) or HTTP activity (http stream). Each log stream defines a set of fields with names and types that determine the content of the log.
- **Filters** - Each stream can have multiple filters to control what information gets logged and how it is logged. By default, each stream has a filter that logs all data directly to disk. Additional filters can be added to customize the output, such as logging only certain fields, writing to different outputs, or setting a specific rotation interval for the log files.
- **Writers** - Filters use writers to define the output format of the log data. The default writer is the ASCII writer, which creates tab-separated ASCII files. Other writers, like those for binary output or direct database logging, are also available.

4.3 Security Auditing tools

Let’s now analyze the security auditing tools and frameworks integrated into our project. These tools will help verify potential vulnerabilities identified by the IDS, offering an assessment of threats. After the verification is complete, the results are documented in log files for further analysis.

4.3.1 TLS-Attacker

TLS-Attacker is an open-source framework for evaluating the security of TLS (Transport Layer Security) libraries. Developed by researchers at Ruhr University Bochum[40], it provides a flexible platform for creating custom TLS message flows and analyzing the behavior of TLS libraries under various conditions. TLS-Attacker is widely used to identify cryptographic vulnerabilities and boundary issues in TLS implementations.

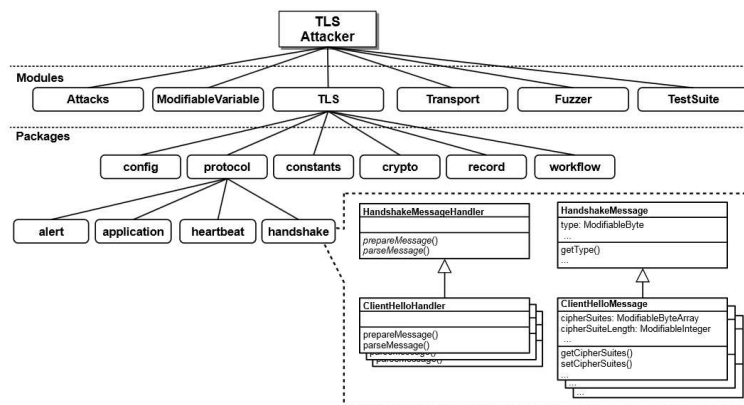


Figure 4.2: TLS Attacker’s Maven modules

Key functionalities of TLS-Attacker:

- **Custom Protocol Flows** - Users can construct and manipulate TLS protocol flows to test for weaknesses in the TLS state machine.
- **Modifiable Variables** - Supports dynamic changes to data types, such as integers and byte arrays, useful for fuzzing operations like addition, XOR, and random insertion.
- **Two-Stage Fuzzing** - Combines cryptographic fuzzing (e.g., padding oracle attacks) with systematic protocol fuzzing to expose memory and boundary issues.
- **Test Suite for TLS Libraries** - Enables the creation of positive and negative test cases to evaluate the robustness and error handling of TLS libraries.

TLS-Attacker is used to test well-known TLS vulnerabilities, such as Bleichenbacher's attack, padding oracle attacks, and invalid curve attacks. For example, testing for Bleichenbacher's attack involves setting a custom premaster secret in the `ClientKeyExchange` message to see if the TLS library improperly distinguishes between valid and invalid RSA padding.

Example command to initiate a test:

```
java -jar TLS-Attacker.jar -config config.xml -attack Bleichenbacher
```

Systematic Fuzzing with TLS-Attacker

TLS-Attacker's fuzzing capabilities follow three phases:

- **Phase 1: Variable Detection:** Identifies critical variables within TLS messages for fuzzing.
- **Phase 2: Variable Fuzzing:** Systematically modifies identified variables, such as message lengths or splitting protocol messages across multiple records.
- **Phase 3: Random Protocol Flows:** Generates random variations of protocol flows to observe system responses.

4.3.2 O-Saft



O-Saft is an advanced forensic tool developed by OWASP (Open Web Application Security Project) to analyze SSL/TLS configurations of remote servers. The primary focus of O-Saft is to provide detailed information about SSL certificates and to test a server's compliance with various SSL protocols and ciphers. It is a useful tool for penetration testers and administrators who need a thorough analysis of SSL configurations. O-Saft has several unique features:

- **Closed Environment Compatibility:** O-Saft is designed to work effectively even in closed environments that have no internet connection.
- **Cipher Availability Testing:** It can test the availability of ciphers independently of the installed SSL library, allowing for broader and more consistent analysis.
- **Minimal Dependencies:** The tool is implemented in Perl, and it does not require the installation of additional Perl modules to perform basic cipher and protocol checks.

The O-Saft command follows this structure:

```
o-saft.pl +[option] [target]
```

- **+ [option]:** Specifies the action or test to be performed. For example, `+check` tests the SSL/TLS configuration, `+info` retrieves certificate details, `+cipher` lists supported ciphers, and `+vulns` checks for specific vulnerabilities.
- **[target]:** The IP address or domain name of the server to be analyzed.

4.3.3 Metasploit



The Metasploit Framework^[38] is a widely used open-source penetration testing platform developed to assist security professionals in identifying, testing, and exploiting vulnerabilities in a variety of systems. Owned by Rapid7, Metasploit provides a comprehensive suite of tools for tasks such as network reconnaissance, exploitation, and post-exploitation activities. It supports multiple modules and offers a flexible platform for scripting custom exploits, payloads, and auxiliary functions, making it a valuable asset for penetration testing and vulnerability assessment.

Metasploit's capabilities are divided into different modules:

- **Exploit Module:** Used to launch attacks on target systems by exploiting specific vulnerabilities.
- **Payload Module:** Manages payloads, which allow attackers to interact with a target system after a successful exploit.
- **Auxiliary Module:** Includes functions for scanning and enumeration, useful for gathering information without deploying a payload.
- **Encoder Module:** Helps evade detection by encoding payloads to bypass antivirus or firewall protections.

Metasploit supports single-line commands using the `-x` option in `msfconsole`, allowing the execution of attack scripts with all necessary parameters defined inline. Below is an example command targeting the Bleichenbacher vulnerability in TLS:

```
msfconsole -x "use auxiliary/scanner/ssl/bleichenbacher_oracle; \  
set RHOST [ipAddress]; set RPORT [port]; set TARGETURI /; \  
set SSL true; check; exit"
```

In this example:

- `use auxiliary/scanner/ssl/bleichenbacher_oracle` - Loads the Bleichenbacher vulnerability scanner.
- `set RHOST [ipAddress]` - Specifies the target's IP address.
- `set RPORT [port]` - Defines the target's port, typically port 443 for HTTPS.
- `set TARGETURI /` - Sets the URI to the root path.
- `set SSL true` - Configures SSL for the connection to handle encrypted communications.
- `check` - Verifies if the target is vulnerable to the Bleichenbacher attack.
- `exit` - Closes the `msfconsole` session.

4.3.4 Nmap



Nmap (Network Mapper) [37] is a powerful open-source tool used for network discovery and security auditing. Its primary function is to map out networks, identify live hosts, open ports, and detect operating systems. Nmap is widely used by network administrators and penetration testers to gain insight into network security and inventory management. While it can be used as a scanner, nmap includes also attack scripts which will be used by Threat-TLS.

Nmap has several features:

- **Comprehensive Network Scanning:** Nmap can identify live hosts, open ports, and running services on a network, providing a detailed view of the network's infrastructure.
- **Port Scanning Techniques:** It offers multiple scanning techniques, including TCP SYN, TCP connect, and UDP scans, allowing for flexible and efficient analysis.
- **OS and Version Detection:** Nmap can detect the operating system and software versions of the devices on the network.
- **Nmap Scripting Engine (NSE):** The tool includes an extensive scripting engine that allows users to automate various networking tasks, such as vulnerability detection and brute-force attacks.

The Nmap command follows this structure:

```
nmap [options] [target]
```

- **[options]:** Specifies the action or test to be performed. For example, `-sS` performs a TCP SYN scan, `-O` detects the operating system, `-sV` detects service versions, and `-A` enables aggressive scanning with multiple tests.
- **[target]:** The IP address, domain name, or IP range of the target to be analyzed.

For example, to check a host for the POODLE vulnerability using Nmap's scripting engine, the following command can be used:

```
nmap --script ssl-poodle -p 443 <target-host>
```

This command uses the `ssl-poodle` script to test for the POODLE vulnerability on port 443 of the specified target host.

4.3.5 Testssl.sh

testssl.sh[\[41\]](#) is a versatile, open-source command-line utility that scans any server port to identify services supporting SSL/TLS protocols and ciphers. It is commonly used to conduct both comprehensive and targeted security assessments of servers.

For a broad evaluation of a server's SSL/TLS configuration, the command:

```
testssl [host]:[port]
```

initiates a scan, testing various aspects of the server's cryptographic setup. The tool highlights each test result using color-coded indicators: green for secure settings, yellow for moderate settings, and red for insecure or problematic settings. This visual feedback simplifies the identification of strengths and weaknesses in the server's SSL/TLS configuration.

For scenarios where a specific vulnerability or feature needs to be examined, *testssl.sh* offers targeted scanning options. Running a focused scan can save time compared to a full scan by limiting the test scope. The following command allows a specific check:

```
testssl [host]:[port] --[test_option]
```

where `[test_option]` is replaced with the particular test of interest, such as specific cipher checks or protocol tests.

4.4 Vulnerability Scanners

Vulnerability scanners are tools designed to identify, classify, and assess potential security weaknesses in systems, networks, and applications. They play an important role in proactive security management by highlighting vulnerabilities before they can be exploited by malicious actors. These tools automate the process of checking for known vulnerabilities and misconfigurations. Vulnerability scanners typically operate by comparing the configuration and software versions of a target system against a database of known vulnerabilities, including Common Vulnerabilities and Exposures (CVEs). They are useful for our project because of their capabilities of extracting Common Platform Enumeration (CPEs)

4.4.1 OpenVAS



The Open Vulnerability Assessment System (OpenVAS)[\[43\]](#), now part of the Greenbone Community Edition (GCE), is a comprehensive open-source vulnerability scanner designed to detect security weaknesses in IT systems. Originally developed as a fork of the Nessus vulnerability scanner when it transitioned to a proprietary model in 2005, OpenVAS has since evolved into a highly reliable and feature-rich tool. It integrates multiple components to provide a full-featured vulnerability management solution

Key Features of OpenVAS

- **Comprehensive Scanning:** OpenVAS performs deep scans of systems and networks, detecting misconfigurations, outdated software, and security vulnerabilities.
- **Regular Updates:** It relies on the Greenbone Community Feed or the commercial Greenbone Enterprise Feed, which provide daily updates of vulnerability tests (VTs).
- **Integrated Management:** It is integrated with the Greenbone Vulnerability Manager (gvm), providing centralized management of scan configurations, schedules, and results.

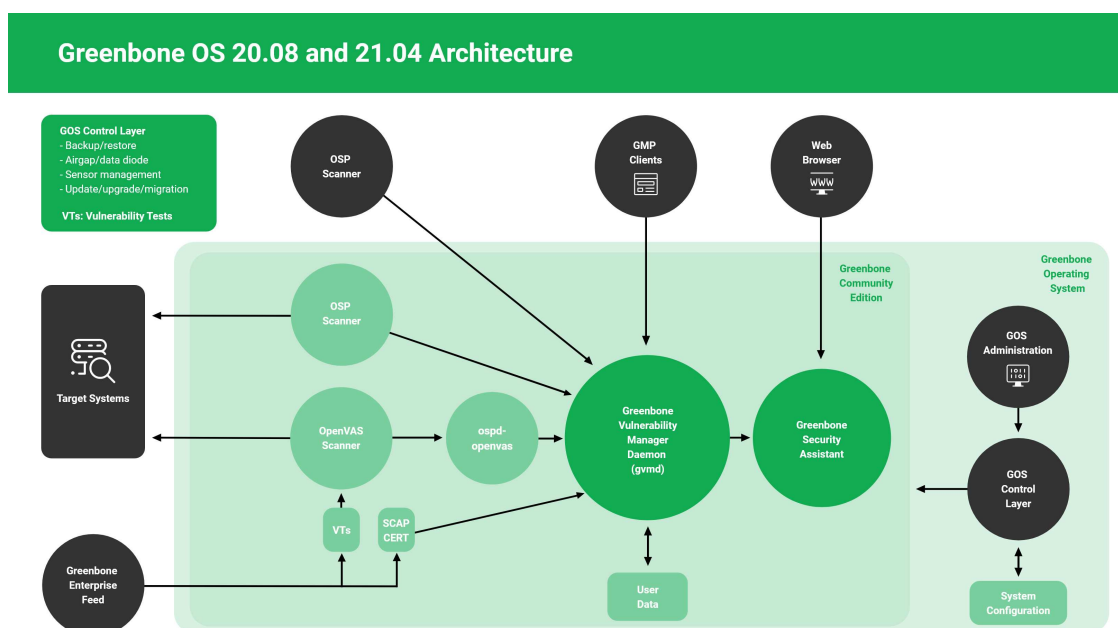


Figure 4.3: OpenVAS Architecture [43]

OpenVAS Scanning Workflow

OpenVAS follows a structured workflow:

1. **Target Definition:** Users specify the systems or network segments to scan.
2. **Scan Configuration:** Predefined scan configurations can be used, or users can create custom configurations to focus on specific vulnerabilities.
3. **Execution:** OpenVAS scans the defined targets and collects data on potential vulnerabilities.
4. **Analysis:** Scan results are analyzed within the Greenbone Security Assistant (GSA), offering detailed insights and recommendations.
5. **Remediation:** Users can address identified vulnerabilities by applying patches, reconfiguring systems, or implementing additional security measures.

4.4.2 NIST NVD Database and API

The National Institute of Standards and Technology (NIST) National Vulnerability Database (NVD)[44] is the United States government's repository of standards-based vulnerability management data. It provides a comprehensive database of known vulnerabilities, each identified by a unique Common Vulnerabilities and Exposures (CVE) identifier. The NVD is widely used for vulnerability assessments and risk management. Its RESTful API enables automated access to vulnerability data, allowing tools to query CVEs, severity metrics (e.g., CVSS scores), and associated remediation steps.

Chapter 5

Threat TLS

As mentioned in the previous chapter, modern TLS scanning solutions provide a good initial assessment of a server's security, but they lack continuous monitoring. Without ongoing checks, a TLS server could become vulnerable over time as new threats and weaknesses are discovered. This is why Monitor for TLS Attacks was developed. With this thesis we want to introduce an evolution of the original tool with the goal of making it more scalable, that will be named Threat-TLS.

5.1 Architecture of Threat-TLS

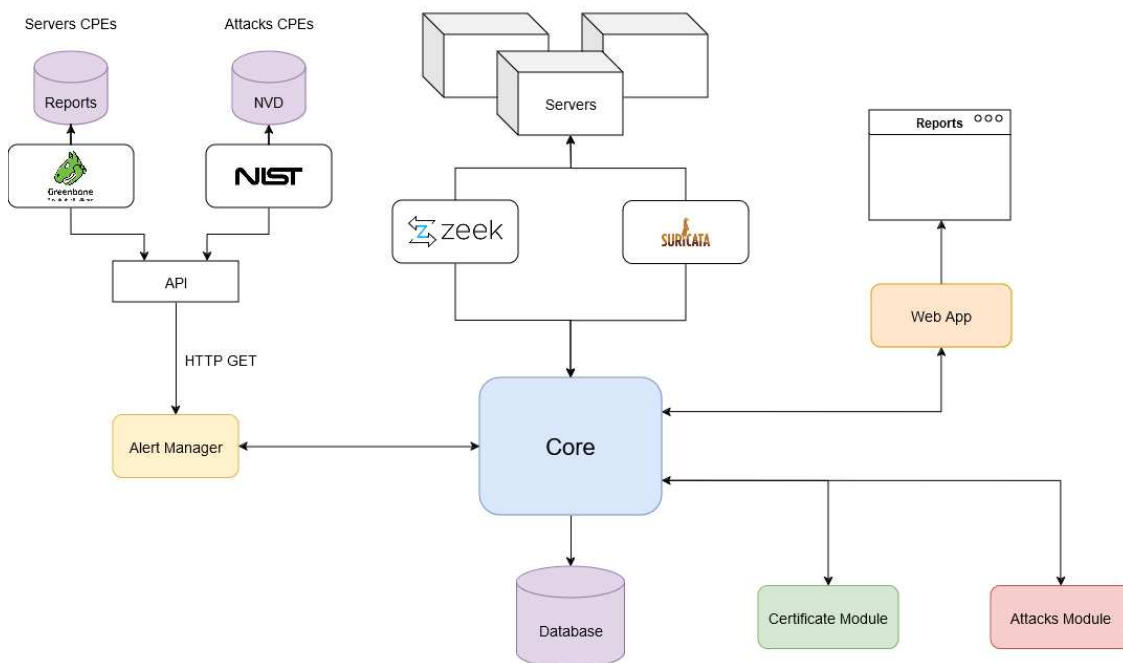


Figure 5.1: Threat-TLS architecture

5.1.1 Design Choices

The architecture of the new monitor is based on the previous tool but has been redesigned to accommodate new features that we want to introduce. The main idea is to scale the architecture to make it more robust and efficient. Our goal with the redesign was to address limitations found within the original version. For example, we found that performance was a limiting factor, which is why we changed the threading structure of the program and added mechanisms to prevent attacks from starting unless relevant. This is why we've added a CPE extraction pipeline. Another limitation that we wanted to address is the lack of analysis and interface. Analyzing the results of the logs was time-consuming and, if deployed in a real scenario, would require a lot of effort from the user. This is why we've added an SQLite database which, for the scope of the application, is well-suited, and a web page. In this way, the user is guided through the results and receives feedback on what steps to take next, thanks to some proposed mitigations.

5.1.2 Description of the Components

- **Core** - The Core is the central unit of the application, acting as the primary orchestrator. It monitors network traffic logs from Suricata or Zeek and processes these logs through various scanning and testing tools, while interacting with the attack and certificate modules. The Core utilizes threading to efficiently manage concurrent tasks, including real-time log parsing, vulnerability detection, and executing targeted scans.
- **Intrusion Detection Systems (IDS)** - The tool employs two IDS: Suricata and Zeek. These systems continuously monitor network activity and generate alerts based on predefined security rules.
- **Alert Manager** - The Alert Manager is a new addition aimed at enhancing the tool's performance by managing which attacks and tests are executed. It communicates with OpenVAS and the National Vulnerability Database (NVD) via APIs to populate the application's database with CPEs related to both servers and known attacks. The Alert Manager then compares these CPEs to determine whether an attack should be launched on a specific server.
- **Attacks Module** - This module is responsible for executing vulnerability attacks using tools like nmap and testssl.sh. The attacks are launched as threads, enabling them to run in parallel, where feasible. To avoid overwhelming the system, attacks are often launched in batches, reducing the load from frequent console invocations.
- **Certificate Module** - The Certificate Module is dedicated to validating TLS certificates, independent of the monitored systems. It verifies the integrity of certificates using Certificate Revocation Lists (CRL), Online Certificate Status Protocol (OCSP), and Signed Certificate Timestamps (SCT) to verify the certificates are trustworthy and have not been revoked or compromised.
- **Web App** - The Web App is built using Flask, which allows for the creation of APIs that enables communication between the HTML pages and the Core application. The web interface is used to display reports generated by the attacks and certificate modules, as well as provide statistics about the application and visualize them.
- **Database** - The tool utilizes an SQLite database to store information on CPEs, servers, and generated reports, providing a central repository for all key data.

5.2 Threat-TLS high-level workflow

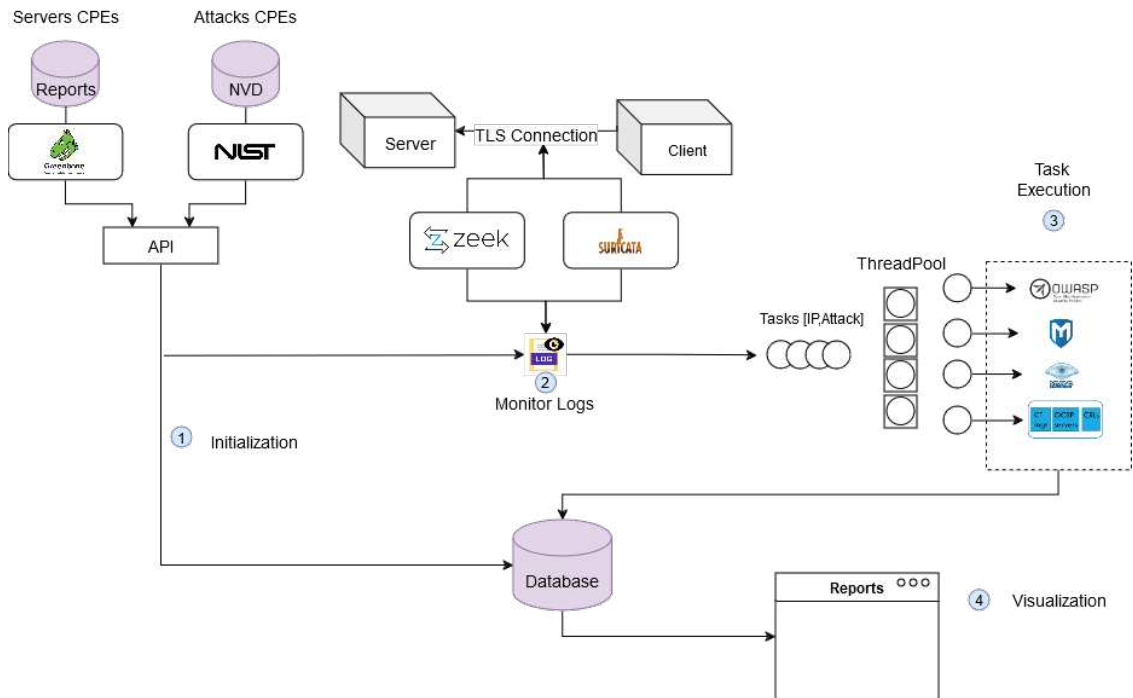


Figure 5.2: Threat-TLS multi-thread design

5.2.1 First Phase: Initialization

The application starts by executing the `app.py` file, which serves as the entry point to the program. This file contains the Flask application responsible for opening the web page upon launch. The following parameters can be used to customize the behavior of the tool:

- `--IDS=Zeek`: Specifies the Intrusion Detection System (IDS) to use, either Zeek or Suricata.
- `--attack`: Specifies a single attack that can be executed on its own.
- `--host`: Specifies the host who will be targeted by the attack.

Once the parameters are set and the IDS is selected, the program connects to the Greenbone Vulnerability Manager (GVM) API. It extracts all Common Platform Enumerations (CPEs) associated with the monitored servers, converts them to version 2.3 format, and populates the tool's database. This step requires that an OpenVAS scan has been previously conducted on the servers within the monitored network.

The purpose of extracting CPEs is to create a full inventory of the software products installed on the monitored machines, enhancing the tool's monitoring efficiency. CPEs provide unique identifiers for products, software, or hardware, including their versions.

To determine whether a particular attack applies to a server, the tool utilizes the National Vulnerability Database (NVD). During initialization, the tool checks its database for attacks without associated CPEs. For such cases, it sends API requests to the NVD, retrieves the relevant CPEs, and updates its database with the newly associated attacks.

To make sure that server configurations remain consistent during monitoring, OpenVAS periodically scans the network and notifies the tool via HTTP GET requests containing the updated CPEs. This mechanism enhances confidence that the servers' configurations have not been tampered with.

5.2.2 Second Phase: Monitoring and Task Creation

Once the CPE extraction is complete, the main function begins execution. The program spawns two threads:

- **Log Reader Thread:** Executes the `log_file` function, responsible for continuously reading the IDS log files.
- **Scheduler Thread:** Manages the processing and assignment of tasks generated from the IDS logs.

When the IDS detects a network event that matches its rules or scripts, the details are written to the log file. The log reader parses these entries (using a parser specific to the selected IDS) and generates a task. Each task is a dictionary containing:

- **Key:** IP addresses and ports of the source and destination.
- **Values:** Detected vulnerability, TLS version, cipher information, and other relevant metadata.

The tasks are then added to a shared queue.

The scheduler checks each new task and determines whether it should be executed by consulting a cache, which resets whenever OpenVAS completes a new scan.

If the condition is satisfied, the scheduler assigns the task to a worker thread from a **Thread Pool**. The maximum number of workers is defined at the start of the program to avoid resource contention. Assigned workers then execute the required attacks and certificate validations using the appropriate modules. The workers in the case of an attack first check if the vulnerability applies to the targeted server by using a function from the Alert Manager to compare CPEs, if applicable they assign new tasks to workers from another Threadpool in charge of executing the attacks.

5.2.3 Third Phase: Task Execution

The workers are now ready to execute the assigned tasks. The tasks involve performing certificate validations and launching attacks:

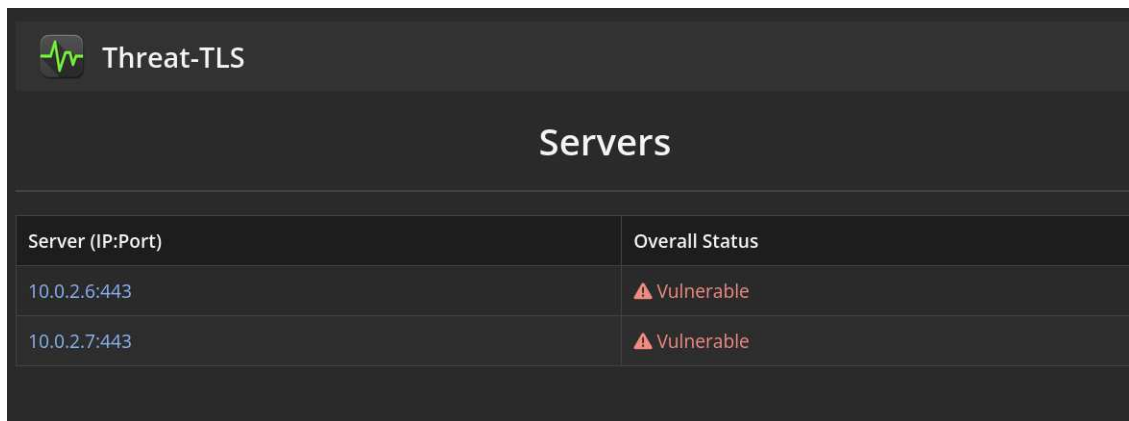
- **Certificate Validation:** The `CertificateValidator` class performs SSL/TLS certificate validation for a given hostname and port. This includes:
 - Fetching the certificate.
 - Verifying the certificate's chain of trust.
 - Checking the revocation status using Online Certificate Status Protocol (OCSP) and Certificate Revocation Lists (CRL).
 - Validating Certificate Transparency using Signed Certificate Timestamps (SCTs).
- **Vulnerability Exploits:** The program executes relevant attacks, batching them when possible to reduce computational overhead (e.g., reusing open consoles for multiple attacks instead of launching new ones).

Once a task is completed, the results are saved to the database. The Flask application is notified of the update, and the worker thread becomes available for new tasks.

5.2.4 Fourth Phase: Visualization

The final phase involves visualizing the results on the web page. The monitored servers are displayed, along with a table for each server that contains:

- Logs of all detected vulnerabilities.
- Statistics summarizing the vulnerabilities found.
- A list of mitigations, informing the user about the steps needed to address the identified weaknesses.



The screenshot shows the main page of the Threat-TLS application. At the top left, there is a logo with a green pulse line and the text "Threat-TLS". Below this, the word "Servers" is centered in a large font. Underneath, there is a table with two columns: "Server (IP:Port)" and "Overall Status". The table contains two rows, both showing the IP address "10.0.2.6:443" and "10.0.2.7:443" respectively, with the status "Vulnerable" indicated by a red triangle icon.

Server (IP:Port)	Overall Status
10.0.2.6:443	⚠ Vulnerable
10.0.2.7:443	⚠ Vulnerable

Figure 5.3: Main page

Threat-TLS

Logs for Server: 10.0.2.7:443

Filter by Attack: All Attacks Filter by Tool: All Tools

Attack	Tool	Status	Timestamp	Logs
Bleichenbacher	TLS-Attacker	Secure	12/2/2024, 10:25:38 AM	View Log
CCS Injection	TestSSL	Vulnerable	12/2/2024, 10:25:36 AM	View Log
Heartbleed	Nmap	Vulnerable	12/2/2024, 10:25:35 AM	View Log
Sweet32	TestSSL	Vulnerable	12/2/2024, 10:25:35 AM	View Log
CCS Injection	TestSSL	Vulnerable	12/2/2024, 10:25:34 AM	View Log
Heartbleed	TestSSL	Vulnerable	12/2/2024, 10:25:32 AM	View Log
Heartbleed	OSAF	Secure	12/2/2024, 10:25:30 AM	View Log
Bleichenbacher	TLS-Attacker	Secure	12/2/2024, 10:25:29 AM	View Log
Heartbleed	TestSSL	Vulnerable	12/2/2024, 10:25:28 AM	View Log
Heartbleed	Nmap	Vulnerable	12/2/2024, 10:25:28 AM	View Log

Previous Showing 221-230 of 297 logs Next

Figure 5.4: Server Details Page

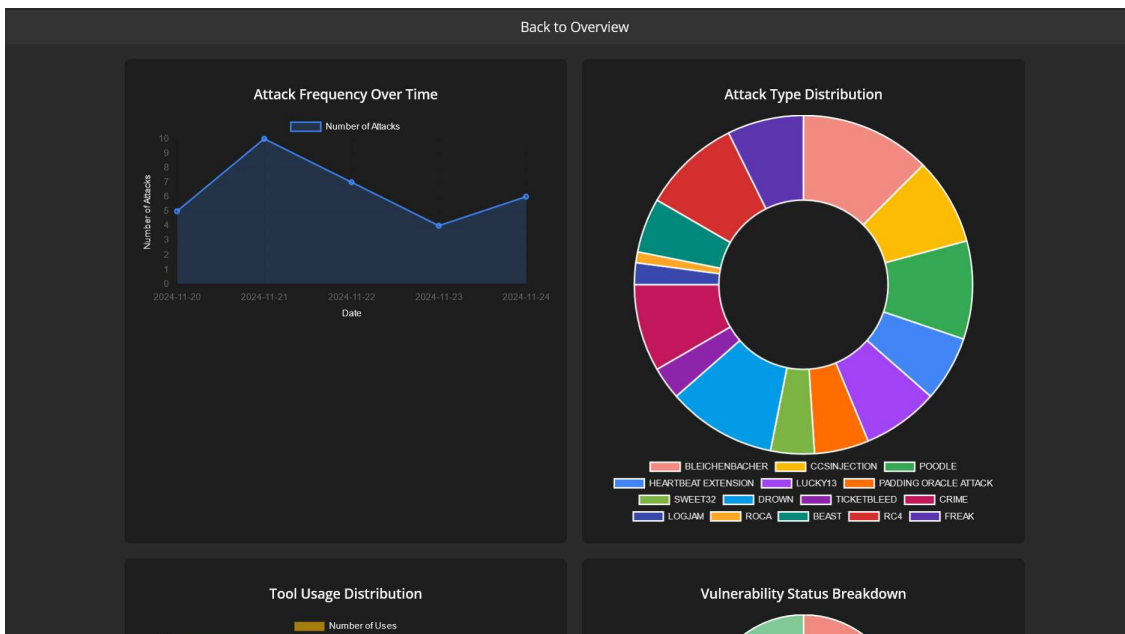


Figure 5.5: Analytics Page

5.3 Understanding how to write Rules/Scripts

It is possible to notice anomalies or patterns corresponding to potential attacks by studying the structure of the TLS handshake messages. Tools like Wireshark allow analyzing the structure of every packet; the insights gained are useful to create alerts in IDS such as Suricata and Zeek.

TLS messages have a fixed structure specified using hexadecimal values. The protocol is divided into phases, each with its own set of messages. At the core of the TLS message is the Record Layer, which encapsulates other TLS components.

The structure of the TLS Record Layer is relatively straightforward:

- **Content Type (1 byte):** Specifies the type of the message; for handshake messages, this is 22 (hexadecimal 16).
- **Protocol Version (2 bytes):** Indicates the TLS version, such as TLS 1.0 (03 01).
- **Record Length (2 bytes):** Specifies the length of the Record Layer payload.
- **Payload:** Contains the encapsulated protocol data, which has its own fixed structure.

By inspecting the protocol version field in the TLS messages, one can identify the TLS version being used. Table 5.1 lists the hexadecimal values corresponding to different TLS versions, which can be observed using Wireshark.

TLS Version	Hex
SSLv2	02 00
SSLv3	03 00
TLSv1.0	03 01
TLSv1.1	03 02
TLSv1.2	03 03
TLSv1.3	03 04

Table 5.1: TLS protocol versions and their corresponding hexadecimal values.

Other components with fixed lengths in the TLS packet structure can also be analyzed:

- **Compression Method**
- **Cipher Suite**
- **Session ID**
- **Extensions**

5.3.1 Inspecting with Wireshark

Using Wireshark, we can identify patterns to write effective rules for traffic analysis. For example, in a TLS 1.2 handshake message, the packet starts with 16 03 03. Here, 16 represents the **content type** for handshake messages, and 03 03 specifies the **protocol version** (TLS 1.2). These patterns can be visualized in Wireshark as shown below:

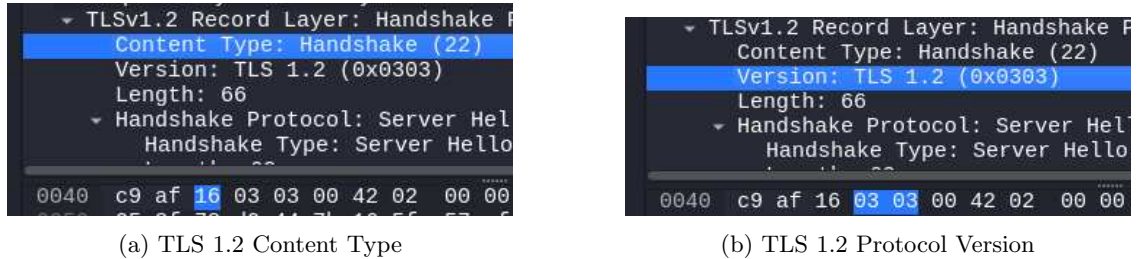


Figure 5.6: Analysis of TLS Content Type and Protocol Version in Wireshark

To distinguish between client messages and server responses, such as identifying the **Server Hello**, we notice that the hex value for the **Server Hello** is 02, which remains constant regardless of the protocol version.

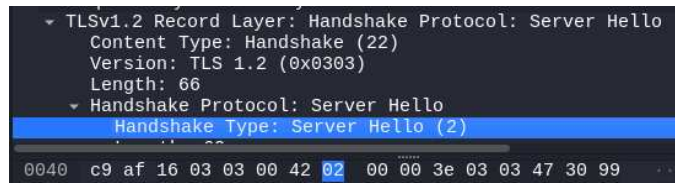


Figure 5.7: TLS Server Hello Identification

Finally, for most vulnerability checks, we must identify the cipher version, weak ciphers indicate potential vulnerabilities. The cipher suite information is located further into the packet:

- **For SSLv3 and SSLv2:** 71 bytes after the content type and protocol version.
- **For TLSv1.0, TLSv1.1, and TLSv1.2:** 41 bytes after the content type and protocol version.

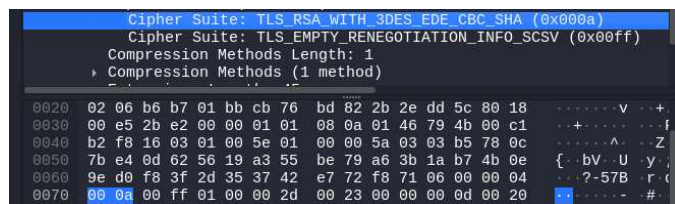
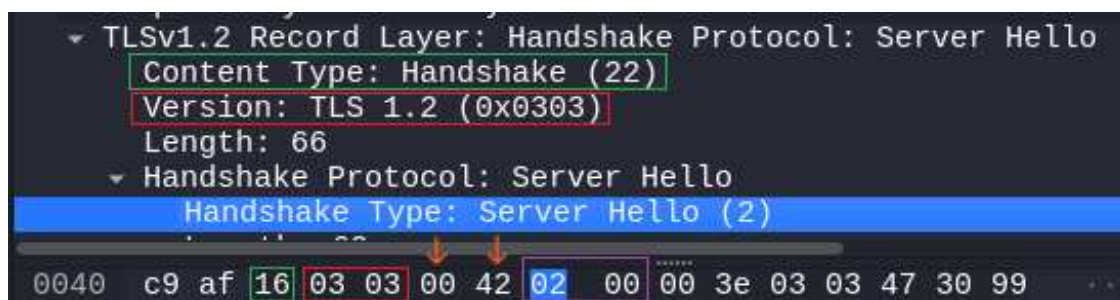


Figure 5.8: TLS Cipher Suite Information

5.3.2 Writing Suricata Rules

With the analysis made with Wireshark, we can now write Suricata rules to detect potential vulnerabilities. Consider the example where a TLS 1.2 connection uses the cipher TLS_RSA_WITH_3DES_EDE_CBC_SHA, which is an RSA cipher that may expose the server to a Bleichenbacher attack.

First Part of the Rule: Identifying TLS 1.2 with Server Hello



To detect the handshake for TLS 1.2, we write the following rule:

```
alert tls any any -> any any (msg:"|VULNERABILITY| #BLEICHENBACHER#
  %TLS_RSA_WITH_3DES_EDE_CBC_SHA%"; flow:from_server; content:"|16 03 03|";
  content:"|02|"; distance:2; within:2; ...)
```

Explanation:

- `alert tls`: The rule applies specifically to TLS traffic.
- `any any -> any any`: Matches traffic from any source IP and port to any destination IP and port.
- `msg:"|VULNERABILITY| #BLEICHENBACHER# %TLS_RSA_WITH_3DES_EDE_CBC_SHA%"`: A log message indicating that an RSA cipher is detected, suggesting potential vulnerability to Bleichenbacher.
- `flow:from_server`: Restricts the rule to traffic originating from the server.
- `content:"|16 03 03|"`: Matches the TLS handshake message for version 1.2:
 - `16`: Denotes a handshake message.
 - `03 03`: Specifies the TLS 1.2 protocol version.
- `content:"|02|"`: Matches the `Server Hello` message type in the handshake.
- `distance:2 within:2`: Specifies where to look for the `Server Hello` marker:
 - `distance:2`: Indicates that the ‘Server Hello’ marker appears 2 bytes after the protocol version.
 - `within:2`: Ensures the marker is found within the next 2 bytes. This generalizes the rule to handle variations in SSLv3 (`distance:3`) and TLS 1.0 to 1.2 (`distance:2`).

Second Part of the Rule: Identifying Vulnerable Cipher

The second rule continues by checking the cipher suite used in the connection. This requires calculating the offset to the cipher suite based on the protocol version.

```
...content:"|16 03 03|"; content:"|00 0a|"; distance:41; within:2; sid:1357;)
```

Explanation:

- `content:"|16 03 03|"`: Matches TLS 1.2 handshake messages.
- `content:"|00 0a|"`: Detects the specific cipher suite `TLS_RSA_WITH_3DES_EDE_CBC_SHA`.
- `distance:41; within:2;`: Just like before, we make sure the cipher suite is located at the correct offset. Instead of just looking for the cipher bytes directly, we calculate the offset relative to the `Server Hello`. This is done to make sure that the content we match is indeed the cipher suite and not some other data elsewhere in the packet.
 - For TLS 1.2 (as in this example), the cipher suite information begins 41 bytes after the protocol version.
 - For SSLv3 and SSLv2, this distance would instead be 71 bytes.

Coupling the Rules

These two rules can be connected using logical operators (e.g., AND) in the Suricata configuration to ensure both conditions are satisfied before triggering an alert.

```
alert tls any any -> any any (msg:"|VULNERABILITY| #BLEICHENBACHER#
  %TLS_RSA_WITH_3DES_EDE_CBC_SHA%"; flow:from_server; content:"|16 03 03|";
  content:"|02|"; distance:2; within:2; content:"|16 03 03|"; content:"|00
  0a|"; distance:41; within:2; sid:1357;
```

5.3.3 Writing Zeek Scripts

Zeek, unlike Suricata, uses scripts to analyze traffic. Using similar logic to that of Suricata rules, we can derive scripts to detect vulnerabilities. For example, to detect vulnerabilities like BleichenBacher/ROBOT, the following script can be used:

```
1 event ssl_server_hello(c: connection, version: count, record_version: count, possible_ts: time,
2   server_random: string, session_id: string, cipher: count, comp_method: count)
3 {
4   # Map cipher codes to their descriptions
5   local cipher_name = SSL::cipher_desc[cipher];
6
7   # Detect non-PSK RSA cipher suites
8   if ( /_RSA_/ in cipher_name && ! ( /PSK/ in cipher_name ) )
9   {
10      NOTICE([$note=ROBOT_NonPSK_RSA_Cipher_Detected,
11        $conn=c,
12        $msg=fmt("Non-PSK RSA cipher suite detected: %s. Potential vulnerability to
13        ROBOT attack.", cipher_name)]);
14   }
```

- `event ssl_server_hello`: This is the event handler in Zeek that triggers when the server sends a `Server Hello` message during the SSL/TLS handshake. It includes parameters like the connection (`c`), protocol version, server random, session ID, and cipher information.
- `local cipher_name = SSL::cipher_desc[cipher];`: This maps the numeric cipher identifier (`cipher`) to its human-readable description using the Zeek `SSL::cipher_desc` table.
- `if (/_RSA_/ in cipher_name && ! (/PSK/ in cipher_name))`: This conditional checks if the detected cipher suite contains `RSA` but does not include `PSK` (Pre-Shared Key). Ciphers that match this condition are potentially vulnerable to BleichenBacher or ROBOT attacks.
- `NOTICE([$note=ROBOT_NonPSK_RSA_Cipher_Detected, ...])`: If the condition is met, Zeek raises a `NOTICE` log entry with a custom message that includes the detected cipher suite and explains the potential vulnerability.
- `fmt`: This formats the output string for the `NOTICE` log, dynamically including the name of the detected cipher suite.

5.4 Implementation of the Rules/Scripts

Now that we have understood how we can write rules and we know the mechanisms of the attacks, can define the markers for each vulnerability. To do so its possible to inspect the packets with Wireshark after launching an attack using the TLS attack tools. Here we have a table with all the markers found:

Table 5.2: Attacks integrated in Threat TLS

Attack	Marker	Attack Tools
Heartbleed	Heartbeat Extension and Message	Metasploit, Nmap, TestSSL, TLS-Attacker, O-saft
CRIME	Compression enabled	TestSSL,O-saft
DROWN	SSLv2 enabled	TestSSL, TLS-Attacker
Bleichenbachers/ROBOT	Non-PSK RSA Ciphersuites	TestSSL, TLS-Attacker, Metasploit
Sweet32	DES/3DES Ciphersuite	TestSSL
LogJam	Export Ciphersuite with weak prime DH	TestSSL, Nmap
Lucky13	CBC Ciphersuite	TestSSL
Padding Oracle Attack	RSA_CBC Ciphersuite	TLS-Attacker
POODLE	SSLv3 enabled and RSA_CBC Ciphersuite	TestSSL, Nmap, TLS-Attacker
MITM	Certificate Self Signed/Revoked/Compromised	*
Ticketbleed	Session Ticket Enabled	TestSSL,Nmap
Change Cipher Spec injection	*	TestSSL,Metasploit, Nmap
Roca Vulnerability	RSA Ciphersuites	Nmap
Beast	SSLv3, TLSv1 + CBC Ciphersuites	TestSSL,O-saft
RC4	RC4 Ciphersuites	TestSSL
Freak	Export Ciphersuites	TestSSL

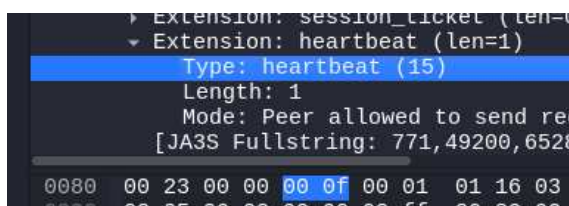
With all these markers defined we can start writintg the rules/scripts, to avoid redundancy we will avoid explaining parts of the rules already discussed previously and focus on the specifics of the vulnerability itself,

5.4.1 Heartbleed

Suricata

Heartbleed's primary markers are:

- The presence of the Heartbeat extension in the initial handshake:

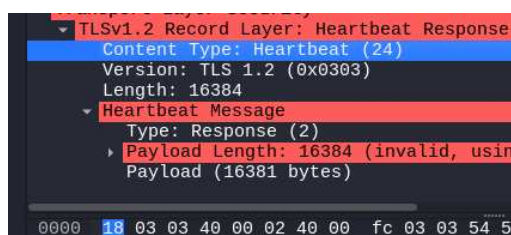


```
alert tls any any <> any any (msg:"|VULNERABILITY| #HEARTBEAT EXTENSION#
$1.2$"; flow:from_server; content:"|16 03 03|"; content:"|02|";
distance:2; within:4; content:"|00 0f|"; content:"|01|"; distance:2;
within:4; sid:13;)
```

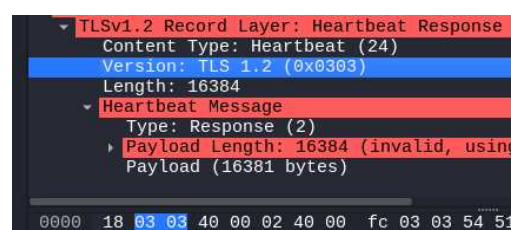
We will omit explaining the first part of the rule as it is similar to what we explained before:

- `content:"|00 0f|"`: Matches the length field of the Heartbeat request
- `content:"|01|"`: Matches the Heartbeat extension length, indicating that is the Heartbeat extension is being used.

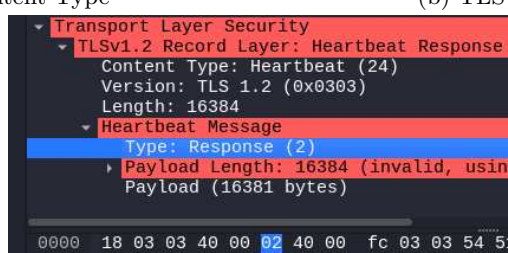
- The presence of a Heartbeat reply message, indicating that the extension is actively used:



(a) Heartbeat Content Type



(b) TLS 1.2 Protocol Version



(c) Heartbeat Response

```

alert tls any any <> any any (msg:"|VULNERABILITY| #HEARTBEAT REPLY# $1.2$";
  flow:from_server; content:"|18 03 03|"; content:"|01|"; distance:3;
  within:1; sid:23;)

```

This rule detects a Heartbeat reply sent by the server.

- `content:"|18 03 01|"`: Identifies the Heartbeat reply protocol and TLS version.
- `content:"|01|"`: Matches the payload of the Heartbeat reply.

Zeek

Similarly in Zeek the script can be written like this:

```

1 event ssl_extension(c: connection, is_orig: bool, code: count, val: string) &priority=5
2 {
3     # 15 is the extension code for Heartbeat(0x0f in hexadecimal)
4     if ( code == 15 && !is_orig )
5     {
6         # Server advertised Heartbeat extension
7         NOTICE([$note=Heartbleed,
8             $conn=c,
9             $msg="Server supports Heartbeat extension. Potential Heartbleed vector."]);
10    }
11 }
12
13 event ssl_record(c: connection, is_orig: bool, record_version: count, content_type: count, data:
14     string)
15 {
16     # 24 is the Content Type for Heartbeat (0x18 in hexadecimal)
17     if ( !is_orig && content_type == 24 )
18     {
19         NOTICE([$note=Heartbleed,
20             $conn=c,
21             $msg="Server sent a Heartbeat response. Potential Heartbleed vulnerability."]);
22    }

```

This Zeek script consists of two events:

1. `ssl_extension`: Detects if the server advertises the Heartbeat extension (`code == 15`) in its SSL handshake, which could indicate support for the Heartbeat feature.
2. `ssl_record`: Monitors SSL/TLS records and triggers a notice when the server sends a Heartbeat response (`content_type == 24`), suggesting potential vulnerability to Heartbleed.

5.4.2 CRIME

For Crime we must check the use of compression from the server.

Suricata

The Suricata rule to detect CRIME vulnerabilities is as follows:

```
alert tls any any -> any any (msg:"|VULNERABILITY| #CRIME#
  TLSv1.1";flow:from_server;content:"|16 03 02|";content:"|02|";
  distance:2;within:3; content:"|16 03 02|";content:"|01|";
  offset:46;depth:1; sid:8;)
```

This rule specifically checks for the compression method in the Server Hello message:

- `content:"|01|"; offset:46;depth:1;`: Matches the compression method field. A non-null value indicates that TLS compression is enabled, signaling a potential CRIME vulnerability.

Zeek

```
1 event ssl_server_hello(c: connection, version: count, record_version: count, possible_ts: time,
2                       server_random: string, session_id: string, cipher: count, comp_method:
3                       count)
4 {
5   # comp_method == 0 indicates "null" compression, which is safe.
6   # Any other value means compression is enabled, which may indicate CRIME vulnerability.
7   if ( comp_method != 0 )
8   {
9     NOTICE([$note=CRIME,
10            $conn=c,
11            $msg="TLS compression is enabled. Possible CRIME vulnerability."]);
12  }
```

This script triggers a NOTICE event if the `comp_method` field in the Server Hello message has a non-zero value, indicating that TLS compression is enabled and the server might be vulnerable to CRIME.

5.4.3 BEAST

For Beast we must check the use of SSLv3/TLS 1.0 + CBC

Suricata

An example of a Suricata rule for detecting BEAST vulnerability is:

```
alert tls any any -> any any (msg:"|VULNERABILITY| #BEAST#
  %TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256% ";flow:from_server;content:"|16
  03 01|";content:"|02|"; distance:2;within:4; content:"|16 03
  01|";content:"|C0 27|"; distance:41;within:2; sid:2214;)
```

- `|16 03 01|`: Indicates the use of TLS 1.0 protocol.
- `|C0 27|`: Identifies a CBC-based cipher suite.

Zeek

The Zeek script to detect BEAST vulnerability is:

```
1 # Detect SSLv3 or TLS 1.0 with CBC cipher suites
2 if ( (version_name == "SSLv3" || version_name == "TLSv1.0") && /_CBC_/ in cipher_name )
```

Similarly this script trigger an alert when a Server Hello containing SSLv3/TLS 1.0 + CBC is detected

5.4.4 Logjam

The markers are the use of Export ciphers with weak DH (≤ 1024 bits)

Suricata

```
alert tls any any -> any any (msg:"|VULNERABILITY| #LOGJAM#
  %TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA%";flow:from_server;content:"|16 03
  01|";content:"|02|";distance:2;within:4;content:"|16 03 01|";content:"|00
  b3|";distance:41;within:2;sid:2739;)
```

|00 b3| Identifies a Diffie-Hellman EXPORT cipher suite

Zeek

```
1 event ssl_server_hello(c: connection, version: count, record_version: count, possible_ts: time,
2     server_random: string, session_id: string, cipher: count, comp_method:
3     count)
4 {
5     # Map cipher codes to their descriptions
6     local cipher_name = SSL::cipher_desc[cipher];
7
8     # Store potential export ciphers for further validation
9     if ( /EXPORT/ in cipher_name && /_DHE_/ in cipher_name )
10    {
11        c$has_export_cipher = T;
12        c$cipher_name = cipher_name;
13    }
14 }
15 event ssl_server_key_exchange(c: connection, dh_p: string, dh_g: string, dh_public: string,
16     dh_bits: count)
17 {
18     # Check for weak Diffie-Hellman keys (1024 bits or fewer) and previously flagged export
19     cipher
20     if ( c$has_export_cipher && dh_bits <= 1024 )
21     {
22         NOTICE([$note=Logjam,
23             $conn=c,
24             $msg=fmt("Logjam vulnerability detected: %s with weak %d-bit Diffie-Hellman keys
25             .", c$cipher_name, dh_bits)]);
26     }
27 }
```

- `ssl_server_hello`: Identifies a DH EXPORT cipher suite (`/EXPORT/` in `cipher_name` and `/_DHE_/` in `cipher_name`) and stores this information in connection-specific variables (`c$has_export_cipher` and `c$cipher_name`).
- `ssl_server_key_exchange`: Validates the presence of a weak Diffie-Hellman key (≤ 1024 bits) and checks if the connection was flagged as using an export cipher. If both conditions are satisfied, it raises a NOTICE.

5.4.5 Lucky13 and Padding Oracle

The marker, for both attacks, is the use of CBC

Suricata

```
alert tls any any -> any any (msg:"|VULNERABILITY| #LUCKY13_Padding Oracle#
  %TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256% ";flow:from_server;content:"|16
  03 00|";content:"|02|"; distance:2;within:4; content:"|16 03
  00|";content:"|C0 27|"; distance:73;within:2; sid:349;)
```

|C0 27| corresponds to a CBC cipher

Zeek

```

1 # Detect CBC cipher suites
2 if ( /_CBC_/ in cipher_name )

```

Again, we check for the use of CBC

5.4.6 POODLE

The marker is the use of SSLv3 with a CBC cipher suite.

Suricata

```

alert tls any any -> any any (msg:"|VULNERABILITY| #POODLE#
  %TLS_DHE_DSS_WITH_AES_128_CBC_SHA% ";flow:from_server;content:"|16 03
  00|";content:"|02|"; distance:2;within:4; content:"|16 03
  00|";content:"|00 32|"; distance:73;within:2; sid:2504;)

```

|00 32| corresponds to a CBC cipher suite.

Zeek

```

1 #Detect SSLv3
2 if ( version_name == "SSLv3" )
3 {
4     #Detect CBC
5     if ( /_CBC_/ in cipher_name )

```

This Zeek script identifies connections using SSLv3 and CBC cipher suites

5.4.7 RC4

The marker is the use of an RC4 cipher suite, which is deprecated due to its known vulnerabilities.

Suricata

```

alert tls any any -> any any (msg:"|VULNERABILITY| #BLEICHENBACHER#
  %TLS_RSA_EXPORT_WITH_RC4_40_MD5% ";flow:from_server;content:"|16 03
  00|";content:"|02|"; distance:2;within:4; content:"|16 03
  00|";content:"|00 03|"; distance:73;within:2; sid:942;)

```

|00 03| corresponds to the RC4 cipher suite.

Zeek

```

1 # Detect RC4 cipher suites
2 if ( /_RC4_/ in cipher_name )

```

This Zeek script identifies connections using RC4.

5.4.8 Sweet32

The marker is the use of DES or 3DES cipher suites.

Suricata

```

alert tls any any -> any any (msg:"|VULNERABILITY| #SWEET32#
  %TLS_RSA_WITH_DES_CBC_SHA% ";flow:from_server;content:"|16 03
  00|";content:"|02|"; distance:2;within:4; content:"|16 03
  00|";content:"|00 09|"; distance:73;within:2; sid:223;)

```

|00 09| corresponds to the DES cipher.

Zeek

```

1 # Detect DES or 3DES cipher suites
2 if ( /_DES_/ in cipher_name )

```

This Zeek script identifies the use of DES or 3DES.

5.4.9 TicketBleed

The marker is the support for the Session Ticket extension.

Suricata

```

alert tls any any -> any any (msg:"RECORD LAYER +TLSv12+ SERVER HELLO
|VULNERABILITY| #TICKETBLEED#"; flow:from_server; content:"|16 03 03|";
content:"|02|"; distance:2; within:4; content:"|00 23|"; distance:0;
within:2; sid:2484;)

```

|00 23| corresponds to the Session Ticket extension in the TLS handshake.

Zeek

```

1 event ssl_extension(c: connection, is_orig: bool, code: count, val: string) &priority=5
2 {
3   # 35 is the extension code for the Session Ticket extension
4   if ( code == 35 && !is_orig )
5   {
6     NOTICE([$note=TicketBleed_Session_Ticket_Supported,
7             $conn=c,
8             $msg="Server supports the session ticket extension. Possible TicketBleed
9             vulnerability."]);
10  }

```

This Zeek script identifies servers that advertise the Session Ticket extension (code == 35) in their SSL/TLS handshake.

5.4.10 FREAK

The marker is the use of Export ciphers

Suricata

```

alert tls any any -> any any (msg:"|VULNERABILITY| #FREAK#
|TLS_RSA_EXPORT1024_WITH_RC4_56_MD5% "; flow:from_server; content:"|16 03
03|"; content:"|02|"; distance:2; within:4; content:"|16 03
03|"; content:"|00 60|"; distance:41; within:2; sid:2030;)

```

|00 60| corresponds to an RSA export cipher suite

Zeek

```

1 # Detect Export cipher suites
2 if ( /EXPORT/ in cipher_name )

```

This Zeek script identifies servers using RSA.

5.4.11 Improvements Made to Zeek Scripts

In the original implementation, many vulnerabilities were passed to the program as generic alerts with an associated cipher suite, as shown below:

```
1 c$ssl$cipher = cipher_desc[cipher];
```

This approach required the tool to determine whether a connection was potentially vulnerable. Consequently, the IDS (Zeek) was not being fully utilized, and in high-load scenarios, this could potentially reduce the tool's overall performance.

Chapter 6

Threat-TLS Implementation

We will now analyze the code and implementation of the solution, the tool was developed using Python 3.12

6.1 Threat-TLS low-level workflow

The project is structured in various files, lets analyze them one by one in order of execution :

6.1.1 Initialization

app.py

As previously mentioned, the program starts with the app.py file, which contains our Flask application. This microframework handles the web application and is responsible for opening the GUI.

The file includes various API routes for different HTML pages and one dedicated route to receive **Greenbone** alerts when a report has finished:

```
1 @app.route('/')
2 ...
3
4 @app.route('/server/<ip>:<port>')
5 ...
6
7 @app.route('/api/server_statuses')
8 ...
9
10 @app.route('/api/server_details/<ip>:<port>')
11 ...
12
13 @app.route('/charts/<server_key>')
14 ...
15
16 @app.route('/api/openvas_alert', methods=['GET'])
17 def openvas_alert():
18     cpe_extractor()
19     if scanner:
20         scanner.reset_cache()
21     return jsonify({"message": "CPEs updated successfully!"}), 200
22
23 if __name__ == '__main__':
24     args = parse_args()
25     threading.Thread(target=alertManager, daemon=True).start()
26     threading.Thread(target=core, daemon=True, args=args).start()
27     threading.Timer(1, open_browser).start()
28     socketio.run(app, debug=False)
```

When the program starts, two threads are launched: the **core thread** and the **alertManager** thread. The core thread processes arguments such as the IDS choice.

alert_manager.py

The **alert_manager.py** file contains several functions, with the most notable being **cpe_extractor**, **get_cve_details**, and **compare_cpes**.

The **cpe_extractor** function is responsible for extracting the CPEs (Common Platform Enumerations) from the Greenbone database. It converts the CPEs to the 2.3 format and stores them in the database. This function is executed both when the program starts and whenever OpenVAS finishes a report. The function uses the Greenbone Management Protocol (GMP) to access reports stored in XML format, which are parsed using XPath methods:

```

1 def cpe_extractor():
2     ...
3
4     with Gmp(connection=connection, transform=transform) as gmp:
5         gmp.authenticate(username, password)
6
7         tasks = gmp.get_tasks()
8
9         for task in tasks.xpath('task'):
10            last_report = task.find('./last_report/report')
11            if last_report is not None:
12                last_report_id = last_report.attrib['id']
13                reports.append(last_report_id)
14
15            for report_id in reports:
16                report = gmp.get_report(report_id)
17                results = report.xpath('report/report/results/result/description')
18                for result in results:
19                    cpes_text = result.text
20                    if cpes_text and '|cpe' in cpes_text:
21                        lines = cpes_text.strip().split('\n')
22                        for line in lines:
23                            if '|cpe' in line:
24                                ip_part, cpe_part = line.split('|', 1)
25                                ip = ip_part.strip()
26                                cpe_name = cpe_part.strip()
27            ...
28
29 new_cpes = [CPEModel(server_id=server.id, cpe_name=cpe_name) for cpe_name in cpes]
30 session.bulk_save_objects(new_cpes)
31 session.commit()
32 ...

```

The **get_cve_details** function connects to the NIST NVD API to retrieve all CPEs associated with the vulnerabilities (CVEs) used by the tool. This function only executes at the start of the program and only if the database lacks CPEs for a specific vulnerability. To ensure security, the API key is stored locally in the environment variables and accessed using `os.environ.get()`:

```

1 def get_cve_details(self, cve_id: str) -> Optional[Dict]:
2     """
3     Fetch CVE details from the NVD API.
4     Implements caching to avoid redundant API calls.
5     """
6     cache_key = cve_id
7     with self.cache_lock:
8         if cache_key in self.cve_details_cache:
9             return self.cve_details_cache[cache_key]
10
11     url = 'https://services.nvd.nist.gov/rest/json/cves/2.0'
12     headers = {'apiKey': self.api_key}
13     params = {'cveId': cve_id}
14
15     try:
16         response = requests.get(url, headers=headers, params=params)
17         response.raise_for_status()

```

```

18     cve_data = response.json()
19     with self.cache_lock:
20         self.cve_details_cache[cache_key] = cve_data
21     return cve_data
22 except requests.HTTPError as http_err:
23     logging.error(f"HTTP error occurred while fetching CVE details: {http_err}")
24 except Exception as err:
25     logging.error(f"An error occurred while fetching CVE details: {err}")
26 return None

```

The `compare_cpes` function compares the CPEs extracted from servers with those associated with specific vulnerabilities to determine whether an attack should be launched on a given host.

```

1 def compare_cpes(self, server_cpe: CPE, attack_cpe: CPE, attack_cpe_entry: AttackCPE) -> bool:
2     ...
3     if attack_cpe_entry.version_start_including:
4         try:
5             start_ver = version.parse(attack_cpe_entry.version_start_including)
6             meets_start_including = server_ver >= start_ver
7         except:
8             pass
9
10    if attack_cpe_entry.version_end_including:
11        try:
12            end_ver = version.parse(attack_cpe_entry.version_end_including)
13            meets_end_including = server_ver <= end_ver
14        except:
15            pass
16
17    if attack_cpe_entry.version_start_excluding:
18        try:
19            start_ver = version.parse(attack_cpe_entry.version_start_excluding)
20            meets_start_excluding = server_ver > start_ver
21        except:
22            pass
23
24    if attack_cpe_entry.version_end_excluding:
25        try:
26            end_ver = version.parse(attack_cpe_entry.version_end_excluding)
27            meets_end_excluding = server_ver < end_ver
28        except:
29            pass
30    ...

```

6.1.2 Monitoring and Task Creation

core.py

Once the CPEs have been extracted and the core thread has been launched, we enter the second phase of the program.

The core thread begins by parsing the arguments provided, including configurations for the selected IDS (Intrusion Detection System). It then instantiates the **VulnerabilityScanner** class, where the core functionality of the tool resides. Users can provide a JSON configuration file to specify whitelists for TLS versions, ciphers, or certificate fingerprints.

```

1 def core():
2     ...
3     if args.json:
4         try:
5             with open(args.json, 'r') as f:
6                 config_data = json.load(f)
7                 versions_config = set(config_data.get('versions', []))
8                 ciphers_config = set(config_data.get('ciphers', []))
9                 certificate_fingerprint_config = set(
10                    fp.replace(':', '').lower() for fp in config_data.get('
11                    certificate_fingerprint', []))
12         except Exception as e:

```

```

13     logging.error(f"Failed to load configuration from {args.json}: {e}")
14     sys.exit(1)
15
16     scanner = VulnerabilityScanner(
17         full_version=args.full,
18         ids=args.IDS,
19         versions_config=versions_config,
20         ciphers_config=ciphers_config,
21         certificate_fingerprint_config=certificate_fingerprint_config
22     )
23
24     scanner.run()
25 ...

```

The `run()` function initiates two separate threads: a log reader thread and a scheduler thread.

```

1     def run(self):
2         """
3         Starts the vulnerability scanner.
4         """
5         try:
6             if self.ids.upper() == 'SURICATA':
7                 threading.Thread(target=log_reader, args=('/var/log/suricata/fast.log', self.
8                 parser_suricata),
9                                     daemon=True).start()
10            elif self.ids.upper() == 'ZEEK':
11                threading.Thread(target=log_reader,
12                args=('/usr/local/zeek/logs/current/notice.log', self.
13                parser_zeek),
14                                    daemon=True).start()
15            else:
16                logging.error(f"Unsupported IDS: {self.ids}")
17                sys.exit(1)
18                logging.info('Threat TLS started...')
19                self.scheduler()
20            except (KeyboardInterrupt, SystemExit):
21                logging.info("Threat TLS terminated.")
22                sys.exit(0)

```

The `log_reader` function monitors the IDS log files. It continuously loops and reads new lines as they are appended to the logs. For every new log entry, it calls the appropriate parser based on the IDS selected.

```

1     def log_reader(src_path: str, parser):
2         """
3         Tails a file and processes each new line using the provided file_reader function.
4         """
5         with open(src_path, "rb") as fp:
6             fp.seek(0, os.SEEK_END)
7             while True:
8                 line = fp.readline()
9                 if not line:
10                    time.sleep(0.1)
11                    continue
12                parser(line)

```

The parsers, `parser_suricata()` and `parser_zeek()`, extract the relevant data from the IDS logs, such as the source and destination IPs, the identified vulnerability, the TLS version, and the cipher used. The parsers check if the same vulnerability has already been processed for a given server; if so, the task is skipped. Otherwise, a new task is created and added to a thread-safe queue object.

Here's an example of the Zeek parser:

```

1     task_identifier = (source_dest, new_vulnerability)
2     with self.processed_vulnerabilities_lock:
3         if task_identifier not in self.processed_vulnerabilities:
4             self.processed_vulnerabilities.add(task_identifier)
5             version = ""
6             if new_vulnerability == 'HEARTBEAT EXTENSION':
7                 version = vuln[2].split("$")[1]

```

```

8         logging.info(f'Suricata detected vulnerability {new_vulnerability} for {
source_dest}')
9         task = {
10             'connection': source_dest,
11             'vulnerability': new_vulnerability,
12             'extra_data': {
13                 'tls_version': version,
14                 'cipher_suite': cipher_suite_found,
15             }
16         }
17         self.vuln_queue.put(task)

```

Simultaneously, the scheduler thread runs. Its task is to loop continuously until it detects a new task in the queue. When a task is found, it assigns it to a free worker in the thread pool. Workers are threads that remain idle until a task is assigned to them, allowing for concurrent processing.

```

1     def scheduler(self):
2         """
3         Main loop to check vulnerabilities from the queue using a ThreadPoolExecutor.
4         """
5         logging.info("Vulnerability test thread is started...")
6         with ThreadPoolExecutor(max_workers=self.max_workers) as executor:
7             while True:
8                 try:
9                     task = self.vuln_queue.get(timeout=1)
10                    if task is None:
11                        break
12                    executor.submit(self.process_vulnerability, task)
13                    self.vuln_queue.task_done()
14                except queue.Empty:
15                    continue

```

6.1.3 Task Execution

Once a worker thread is assigned a task, it begins processing it by calling the `process_vulnerability()` function.

This function extracts the relevant data from the task and uses the `compare_cpes()` function to check if the task is applicable to the server. If applicable, the function retrieves the necessary tests for the vulnerability from a dictionary called `vulnerability_tests`. This dictionary maps vulnerabilities to a list of lambda functions that perform specific tests. The worker assigns these tests to another thread pool, which has its own works, with each test executed by a separate one. The original worker then becomes available to take on new tasks.

```

1     def process_vulnerability(self, task: Dict):
2
3         ...
4
5         if self.compare_cpes(server_cpe, attack_cpe, attack_cpe_entry):
6
7             match_found = True
8             break
9
10        if not match_found:
11            logging.info(f"No matching CPEs for vulnerability {vulnerability} on {ip_source}.
Skipping")
12            return
13        logging.info(f"Start test for {ip_source}:{port_source} for {vulnerability} vulnerability")
14
15        vulnerability_tests = {
16
17            Vulnerability.HEARTBLEED.value: [
18                lambda: heartbleed_testssl(ip_source, port_source),
19                lambda: metasploit_heartbleed_process(ip_source, port_source, extra_data.get('
tls_version',
20                lambda: nmap_heartbleed_process(ip_source, port_source),
21                lambda: heartbleed_tls_attacker(ip_source, port_source),
22        ],

```

```

23     ...
24 }
25
26 tests = vulnerability_tests.get(vulnerability, [])
27 if not tests:
28     logging.info(f"No tests defined for vulnerability {vulnerability}. Skipping.")
29     return
30 with ThreadPoolExecutor(max_workers=DEFAULT_MAX_WORKERS) as executor:
31     future_to_test = {executor.submit(test_func): test_func for test_func in tests}
32     for future in as_completed(future_to_test):
33         test_func = future_to_test[future]
34         try:
35             future.result()
36         except Exception as e:
37             logging.error(f"Error during {vulnerability} test on {ip_source}:{
port_source}: {e}")

```

At this stage, workers either execute an attack or perform certificate validation.

If the task to perform is a certificate validation the worker executes the function `process_certificate()`, which thanks to the `asyncio` module launches the `validate_certificate` function asynchronously.

```

1  def process_certificate(self, ip_source: str, port_source: str):
2      """
3      Processes certificate-related vulnerabilities.
4      """
5      try:
6          loop = asyncio.new_event_loop()
7          asyncio.set_event_loop(loop)
8          cert_data = loop.run_until_complete(validate_certificate(ip_source, int(port_source)
9      ))
10         loop.close()
11         ...

```

certificate.py

This file contains the `CertificateValidator` class, which is designed to validate SSL/TLS certificates, including checking the chain of trust, revocation status (via OCSP and CRL), and Certificate Transparency (SCTs). It is initialized with a hostname, port, and an optional timeout value.

It contains different methods to execute these tasks:

The `get_cert_for_hostname()` method fetches the certificate from the server by establishing a connection and wrapping it with the SSL context. It then extracts the certificate in DER format and converts it to PEM format:

```

1  def get_cert_for_hostname(self) -> Tuple[Optional[x509.Certificate], Optional[str]]:
2      ...
3      with socket.create_connection((self.hostname, self.port), timeout=self.timeout) as sock:
4          with self.context.wrap_socket(sock, server_hostname=self.hostname) as ssock:
5              self.der_cert = ssock.getpeercert(True)
6              self.cert_pem = ssl.DER_cert_to_PEM_cert(self.der_cert)
7              self.cert = x509.load_pem_x509_certificate(self.cert_pem.encode('ascii'),
8              default_backend())
9      ...

```

The `get_certificate_fingerprint` method retrieves the fingerprint of the loaded certificate:

```

1  def get_certificate_fingerprint(self) -> Optional[str]:
2      ...
3      fingerprint = self.cert.fingerprint(self.cert.signature_hash_algorithm).hex()
4      ...

```

The `validate_certificate_chain` method uses the `certvalidator` library to validate the certificate chain, including fetching intermediate certificates:

```

1 def validate_certificate_chain(self):
2     ...
3     validator = CV(cert,intermediate_certs=intermediates,validation_context=context)
4     validator.validate_tls(self.hostname)
5     ...

```

The **check_ocsp_revocation** and **check_crl_revocation** methods perform revocation checks using **OCSP** and **CRL**, respectively. They make HTTP requests to the URLs extracted earlier and process the responses to determine the revocation status.

```

1 async def check_ocsp_revocation(self):
2     ...
3     if obsp_resp.response_status == obsp.OCSPResponseStatus.SUCCESSFUL:
4         status = obsp_resp.certificate_status
5         logger.info(f"OCSP revocation status: {status}")
6         if status != obsp.OCSPCertStatus.GOOD:
7             raiseCertificateValidationError(f"Certificate is revoked according to OCSP: {
            status}")
8     ...

```

```

1 async def check_crl_revocation(self):
2     ...
3     async with aiohttp.ClientSession() as session:
4     async with session.get(crl_url, timeout=self.timeout) as response:
5         if response.status != 200:
6             logger.error(f"CRL server returned status {response.status}")
7             continue
8         content = await response.read()
9         crl = x509.load_der_x509_crl(content, default_backend())
10        for revoked_cert in crl:
11            if revoked_cert.serial_number == self.cert.serial_number:
12                logger.error(f"Certificate is revoked according to CRL at {crl_url}")
13                raise CertificateValidationError("Certificate is revoked according to CRL")
14            logger.info(f"Certificate is not revoked according to CRL at {crl_url}")
15        ...

```

Finally the **check_scts** method verifies Signed Certificate Timestamps (SCTs) for Certificate Transparency by parsing the SCTs from the certificate and using OpenSSL to verify them:

```

1 def verify_sct(self, sct, public_key_pem, ee_cert_der):
2     ...
3     ...
4     scts=self.cert.extensions.get_extension_for_oid(ExtensionOID.
        SIGNED_CERTIFICATE_TIMESTAMPS).value
5     for sct in scts:
6         if not self.verify_sct(sct.encode(), public_key_pem, self.der_cert):
7             raise CertificateValidationError(f"SCT verification failed for log
            {
            log_id_b64}.")
8     ...

```

attacks.py

If the task to be executed is instead an attack, we enter the attacks.py file. This file contains all the attack functions. When executing an attack, a worker thread calls a function that looks like this:

```

1 def crime_testssl(ip, port):
2     options = ['--crime']
3     run_testssl(ip, port, options, 'CRIME', 'TestSSL')

```

The file is designed modularly, with wrapper functions that execute attacks based on the parameters provided. In this case, the `run_testssl()` function is called.

```

1 def run_testssl(ip, port, options, attack_name, tool, timeout=None):
2     """
3     Runs a testssl scan with the specified options.
4     """
5     if not is_tool_available('testssl'):

```



```

6     log_message("testssl is not installed or not in PATH", "ERROR")
7     return
8     host = f"{ip}:{port}"
9     command_list = [
10        'testssl',
11        *options,
12        '--color', '0',
13        '--parallel',
14        '--warnings', 'off',
15        host
16    ]
17    run_subprocess(command_list, ip, port, attack_name, tool, timeout)

```

The `run_testssl()` function calls the `run_subprocess()` function, which executes the attack as a subprocess. This function captures the output and checks the logs for vulnerabilities. It then calls `save_attack_results()` to store the results in the database.

```

1 def run_subprocess(command_list, ip, port, attack, tool, timeout=None):
2     """
3     Runs a subprocess with the given command list, capturing the output and errors.
4     """
5
6     try:
7         log_message(f"Starting {attack} ({tool}) on {ip}:{port}", "START")
8         process = subprocess.Popen(command_list, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
9         stdout, stderr = process.communicate(timeout=timeout)
10        stdout = stdout.decode()
11        stderr = stderr.decode()
12        combined_output = stdout + stderr
13
14        vulnerable = check_vulnerability(combined_output)
15
16        save_attack_result(ip, port, combined_output, attack, tool, vulnerable)
17    ...

```

The `save_attack_result` function triggers the Flask application using `socketio.emit`, a function provided by the `socketio` library. This function notifies connected clients of any changes, allowing our web application to dynamically update the user interface without requiring the client to reload the page periodically. This functionality is essential for providing a seamless and interactive user experience.

6.1.4 Visualization

After the results are written to the database and the web application client is notified, the data can be visualized through two main HTML pages, which are served via the Flask APIs defined earlier.

index.html

The first page, `index.html`, displays all the monitored servers and provides a preview of their statuses, either *Secure* or *Vulnerable*. The data is fetched through the function `fetchServerStatuses()`, which communicates with the server API to retrieve the statuses. This function loads the page dynamically updates to reflect the latest status of all monitored servers.

```

1     function fetchServerStatuses() {
2         document.getElementById('loading-spinner').style.display = 'block';
3         fetch('/api/server_statuses')
4             .then(response => response.json())
5             .then(data => {
6                 document.getElementById('loading-spinner').style.display = 'none';
7                 const tableBody = document.querySelector('tbody');
8                 tableBody.innerHTML = ''; // Clear the existing rows
9                 Object.entries(data).forEach(([server, details]) => {
10                    const row = document.createElement('tr');
11                    const serverCell = document.createElement('td');
12                    const statusCell = document.createElement('td');

```

```

13
14     const link = document.createElement('a');
15     link.href = /server/server;
16     link.textContent = server;
17     link.setAttribute('aria-label', View details for server server);
18     link.setAttribute('tabindex', '0');
19
20     serverCell.appendChild(link);
21
22     statusCell.innerHTML = details.overall_status === 'Vulnerable'
23     ? '<i class="fas fa-exclamation-triangle" style="color: #f28b82;"></i>'
i> Vulnerable'
24     : '<i class="fas fa-check-circle" style="color: #81c995;"></i>'
Secure';
25     statusCell.className = details.overall_status === 'Vulnerable' ? '
vulnerable' : 'secure';
26     ...

```

server_details.html

The second page, **server_details.html**, provides a detailed view of the results from previous scans. Users can review logs, examine statistics, and view remediation suggestions. The function **fetchServerDetails()** fetches the necessary data from the server in a manner similar to **fetchServerStatuses()**.

```

1  function fetchServerDetails() {
2  document.getElementById('loading-spinner').style.display = 'block';
3  fetch('/api/server_details/{serverKey}')
4  .then(response => response.json())
5  .then(data => {
6      document.getElementById('loading-spinner').style.display = 'none';
7      document.getElementById('server-title').textContent = Logs for Server: serverKey;
8      const attacks = data.attacks;
9      tableData = Object.values(attacks).map((attack, index) => ({
10         id: index,
11         attack_name: attack.attack_name,
12         tool: attack.tool,
13         vulnerable: attack.vulnerable,
14         timestamp: new Date(attack.timestamp),
15         log_content: attack.log_content
16     }));
17  });
18  ...

```

Chapter 7

Tests and Results

In this chapter we will execute some tests and analyze the results and performance improvements made over the original work

7.1 Testbed

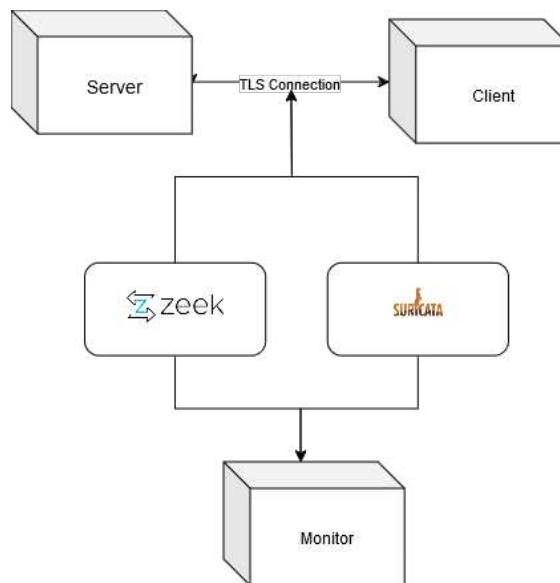


Figure 7.1: Configured Testbed

The tests were conducted in a local environment using three virtual machines running on VirtualBox 7.1.4:

- Two machines, serving as client and server, were running Ubuntu 14.04 LTS.
- One machine was running the Threat-TLS monitor on Kali Linux 2024.4.

The host machine used for the tests is equipped with a 6-core, 12-thread Ryzen 3600X CPU clocked at 4.4 GHz and 16 GB of RAM operating at 3200 MHz.

All virtual machines were connected via a virtual NAT network with IPv4 address space 10.0.2.0/24, configured in promiscuous mode to allow the monitor to sniff network packets.

To conduct these tests, we set up two server machines using different OpenSSL versions. The first server was running Nginx 1.24.0 with OpenSSL 1.0.1f, while the second server was running Apache

2.47.2 with OpenSSL 1.0.2o . This configuration was intended to test the filtering capabilities of our redesign; we expect that when a server is not running vulnerable software, the tool will refrain from launching unnecessary attacks.

7.2 Tests

We are going to test the performance of the tool, the reliability under stress conditions, and for each test we are going to compare to the original tool.

7.2.1 Vulnerability Appendix

The following table outlines the vulnerabilities associated with OpenSSL versions 1.0.1f and 1.0.2o. Each entry lists the attack name, its corresponding CVE code, and whether the respective OpenSSL version is vulnerable.

Table 7.1: Vulnerabilities for OpenSSL Versions 1.0.1f and 1.0.2o

Attack Name	CVE Code	OpenSSL 1.0.1f	OpenSSL 1.0.2o
Bleichenbacher	CVE-2012-0884	No	No
CCS Injection	CVE-2014-0224	Yes	No
POODLE	CVE-2014-3566	Yes	No
Heartbeat Extension	CVE-2014-0160	Yes	No
Lucky13	CVE-2013-0169	Yes	No
Padding Oracle Attack	CVE-2016-2107	Yes	Yes
Sweet32	CVE-2016-2183	Yes	Yes
DROWN	CVE-2016-0800	Yes	Yes
CRIME	CVE-2012-4929	Yes	Yes
Logjam	CVE-2015-4000	Yes	Yes
BEAST	CVE-2011-3389	Yes	Yes
RC4	CVE-2013-2566	Yes	Yes
FREAK	CVE-2015-0204	Yes	No

7.2.2 Limitations

- TLS Attacker, although implemented, is not present in the tests, that is because the tool's version is not supported anymore and runs on legacy software, I was not able to make it run, but if a fix is found then or the new versions support the attacks again, it can be run again
- Some specific attacks also could not be tested as they need specific hardware setups to be run:

Table 7.2: Attacks that can't be tested

TLS Attacks	Specific Hardware required	Referenceces
ROCA	Infineon Trusted Platform Module (TPM) firmware versions before 0000000000000422 - 4.34, before 000000000000062b - 6.43, and before 0000000000008521 - 133.33	CVE-2017-15361
Ticketbleed	F5 BIG-IP LTM, AAM, AFM, Analytics, APM, ASM, GTM, Link Controller, PEM, PSM	CVE-2016-9244
ROBOT	BIG-IP (F5 vendor), Citrix NetScaler,etc..	Different CVEs

7.2.3 Applicability Test

In this test, we aim to verify the capabilities of the new Tool to filter attacks that do not have compatible CPEs according to the NVD database. For this test, we used two servers running different versions of OpenSSL: Apache2 with OpenSSL 1.0.2o and Nginx with OpenSSL 1.0.1f. We send packets that execute various attacks to observe the behavioral differences between the old and the new tools.

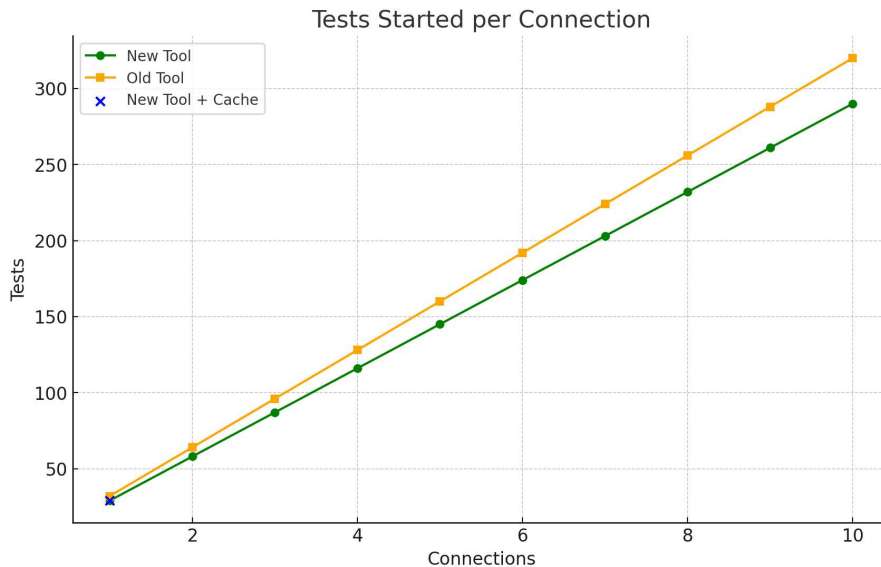


Figure 7.2: Applicability test for Nginx OpenSSL 1.0.1f

For the Nginx server running OpenSSL 1.0.1f, we notice a smaller improvement. Both tools execute a similar number of attacks due to the server's older OpenSSL version having a broader range of vulnerabilities. The new tool still filters out some inapplicable attacks, but the overall difference is less pronounced compared to the Apache2 test.

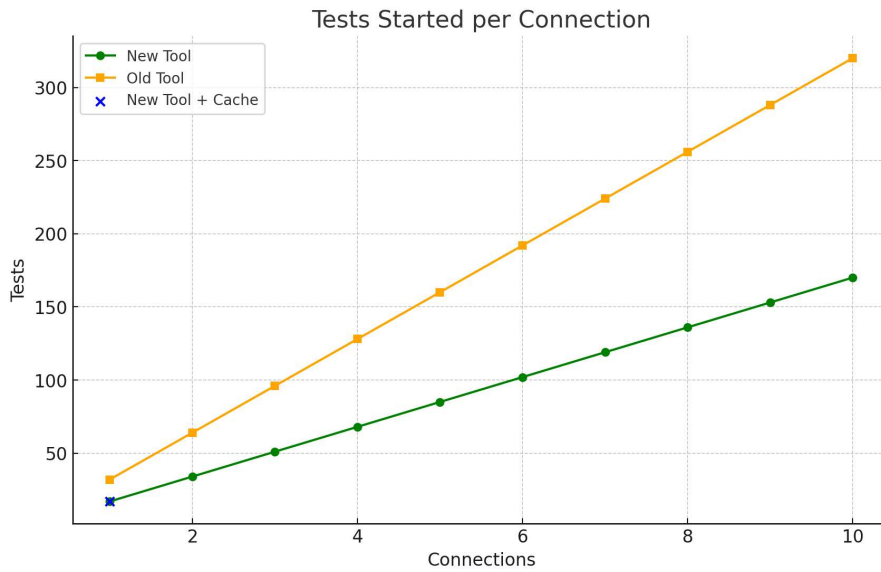


Figure 7.3: Applicability test for Apache2 OpenSSL 1.0.2o

For the Apache2 server running OpenSSL 1.0.2o, the improvement is more significant. The old tool starts 32 attacks for each new attack, whereas the new tool filters out inapplicable attacks and executes only 17 attacks. Additionally, with the cache mechanism of the new tool, attacks stop at 17 because it does not re-execute attacks on the same server if they have already been performed. The cache resets whenever a new alert from OpenVAS is received.

7.2.4 Performance Test

This test compares the raw performance of the original tool with the new one. To make it an accurate comparison, we disabled the alert manager and caching mechanisms in the new tool. Both tools were subjected to stress tests, where the number of concurrent connections was incremented in steps of 10, starting from 10. Both tools had logging enabled, as the original tool relies heavily on logs, which are known to slightly slow down execution.

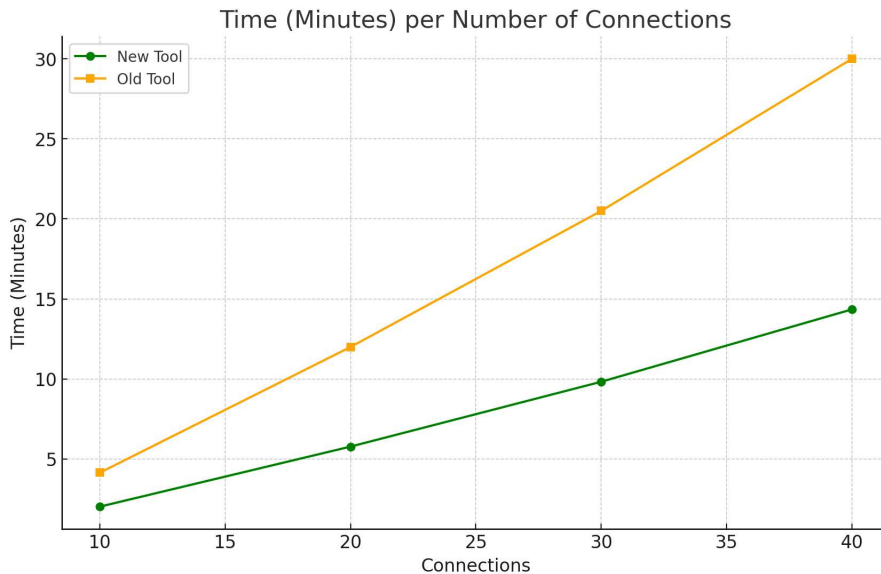


Figure 7.4: Raw performance test

Table 7.3: Comparison of Tools for Time per Number of Connections

Connections	Old Avg (s)	New Avg (s)	Improvement Factor
10	4.15	2.03	2.04
20	12.00	5.68	2.11
30	20.50	9.70	2.11
40	30.00	14.13	2.12

In this test, the command `openssl s_client -connect <server_ip>:443 -cipher DES-CBC3-SHA` was executed multiple times. Regardless of the server chosen, the program’s execution behavior remains consistent.

By analyzing the graph, we observe that the new tool scales almost linearly with the number of connections and completes tasks in about half the time of the original tool. This performance improvement is primarily because the original tool executes all tests in parallel for a single connection, and new jobs need to wait until the tests being executed. which scales poorly when numerous vulnerabilities must be processed. The new tool, in contrast, queues the tests and waits for their completion, leading to better scaling.

This behavior can also be observed in CPU usage during the test execution. The **old tool** exhibits high and erratic CPU usage, while the **new tool** demonstrates more consistent and efficient CPU utilization:

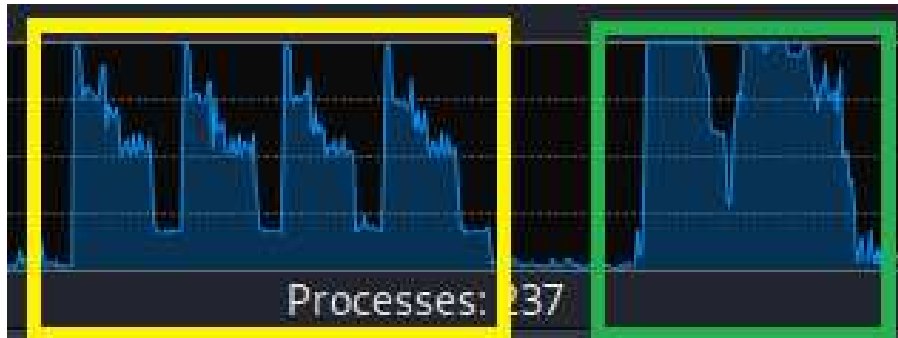


Figure 7.5: CPU usage comparison

When the alert manager is enabled in the new tool, we observe an even greater performance improvement of a factor of about 2.8, this factor obviously depends on the nature of the connections and how many tests get executed:

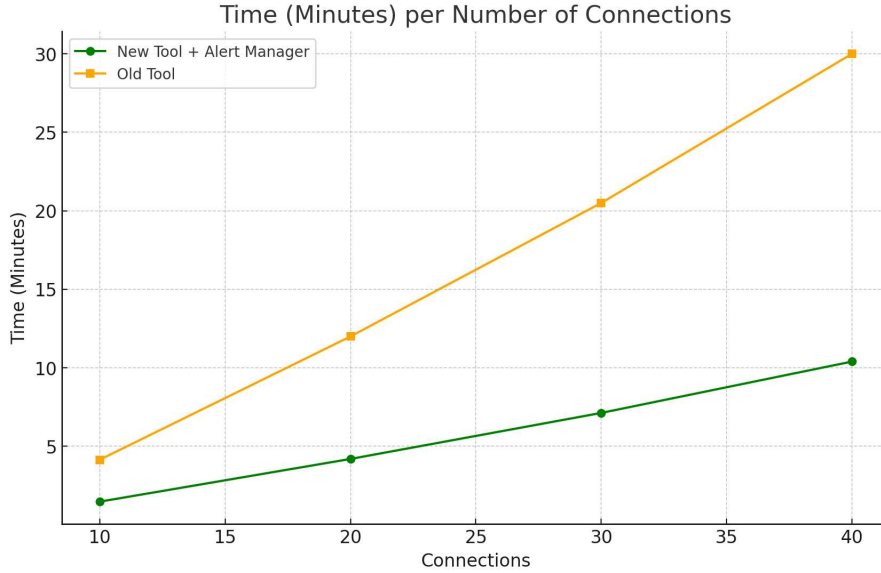


Figure 7.6: Alert manager performance test

Table 7.4: Comparison of Tools for Time per Number of Connections (with Alert Manager)

Connections	Old Avg (s)	New+Alert Manager Avg (s)	Improvement Factor
10	4.15	1.48	2.80
20	12.00	4.20	2.86
30	20.50	7.13	2.88
40	30.00	10.40	2.88

Finally if we also enable the caching, the tool becomes a lot faster, for obvious reasons, many tasks that have been already executed get discarded by the scheduler in the future, as already mentioned this behaviour gets reset when the tool receives an OpenVas alert.

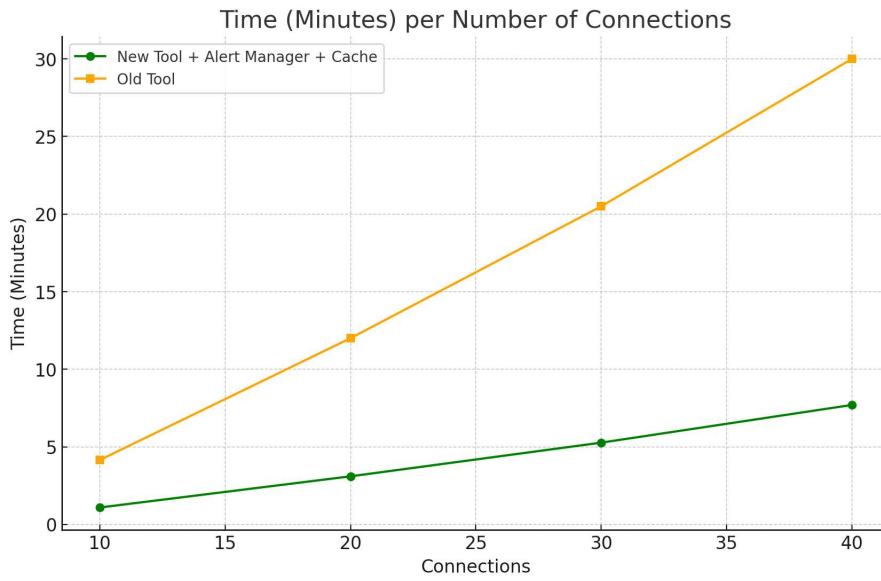


Figure 7.7: Alert manager + Cache performance test

Table 7.5: Comparison of Tools for Time per Number of Connections (With Cache)

Connections	Old Avg (s)	New Avg (Cache) (s)	Improvement Factor
10	4.15	1.09	3.81
20	12.00	3.10	3.87
30	20.50	5.27	3.89
40	30.00	7.70	3.90

7.2.5 IDS Performance

We can compare the performance of the two tools across different IDS systems (Suricata and Zeek), the tests consist in analyzing how long it takes from the generation of the alert to it being detected by the `log_reader` thread.

We notice that both IDS perform similarly and it doesn't impact much the performance of the tool.

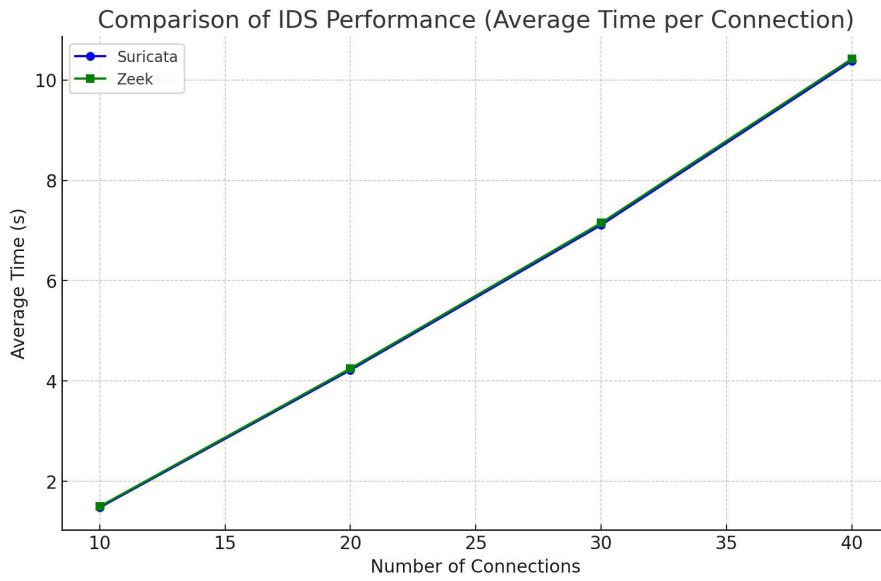


Figure 7.8: Performance comparison of IDS

Table 7.6: Comparison of IDS Performance for Each Number of Connections

Connections	Suricata Avg Time (s)	Suricata Std Dev (s)	Zeek Avg Time (s)	Zeek Std Dev (s)
10	1.48	0.20	1.50	0.22
20	4.22	0.32	4.25	0.35
30	7.11	0.40	7.15	0.42
40	10.38	0.52	10.42	0.54

7.2.6 The Importance of Running Multiple Tests

One of the key advantages of Threat-TLS is its ability to increase the reliability of vulnerability detection through repeated testing with multiple tools. For instance, during stress testing, it became evident that relying solely on a single tool, such as O-Saft, could lead to incorrect conclusions. O-Saft for example, is not able to detect the FREAK vulnerability correctly in some cases.

```
Testing for FREAK attack
FREAK (CVE-2015-0204) VULNERABLE (NOT ok), uses EXPORT RSA ciphers
```

(a) testssl result of FREAK scan on a vulnerable server

```
Connection is safe against FREAK attack: yes
```

(b) o-saft result of FREAK scan on a vulnerable server

Chapter 8

Conclusions

Threat-TLS manages to improve the way we monitor TLS vulnerabilities. By addressing the limitations of the original tool, this implementation aims to be a more performant and usable version. With features like the CPE extraction pipeline, multi-threaded architecture, and the Alert Manager, Threat-TLS improves the process of identifying and prioritizing vulnerabilities, reducing unnecessary scans and saving valuable time and resources.

The tool's modular design makes it flexible and adaptable in case of wanted changes. Its use of IDS systems like Suricata and Zeek allows it to monitor the network continuously. Additionally, the inclusion of an SQLite database and a web-based interface makes analyzing results and taking action much simpler for users.

One limitation of the tool is that, in certain cases, a server may still be vulnerable to an attack despite its CPEs suggesting otherwise. This can occur due to misconfigurations or overlooked settings. For example, the POODLE attack demonstrates this issue: according to the NVD database, a particular OpenSSL version may not appear vulnerable. However, during our tests on an Apache2 server, we observed that SSLv3 was enabled in its configuration. This misconfiguration allowed the server to remain susceptible to downgrade attacks, illustrating the importance of thorough configuration audits alongside vulnerability scanning. Another limitation is the fact that network scans, no matter the optimizations, are external processes and making them run faster is a challenge. A possible solution to this problem would be to split the load in a cluster of monitors, this would improve the parallel execution of the external tools.

Another key takeaway is the value of running multiple tests with different tools. Our tests showed that relying on a single tool can sometimes lead to incorrect conclusions.

Potential enhancements to the could include integrating additional scanners to support hardware CPEs, finding detection of anomalies in TLS 1.3 traffic, and enabling the tool to run across multiple systems for faster processing.

Appendix A

User's Manual

A.1 Requirements

To use Threat-TLS, some tools are needed

- Threat-TLS
- Zeek
- Suricata
- Greenbone Community Edition
- Metasploit
- Nmap
- TLSAttacker
- TestSSL

A.2 Installation

A.2.1 Suricata Installation

1. **Install Dependencies:** The appropriate command needs to be executed based on the operating system:

- **Ubuntu/Debian:**

```
sudo apt-get install libpcre3 libpcre3-dbg libpcre3-dev
build-essential libpcap-dev libyaml-0-2 libyaml-dev
pkg-config zlib1g zlib1g-dev make libmagic-dev libjansson
libjansson-dev
```

- **MacOS:**

```
brew install pkg-config libmagic libyaml nss nspr jansson libnet
lua pcre
```

- **Windows:** The installer needs to be downloaded from the official Suricata download page.

2. **Download and Install Suricata:** The latest Suricata release (`.tar.gz` file) must be downloaded from the Suricata download page. The following steps need to be executed:

```
tar xzvf suricata-7.0.2.tar.gz
cd suricata-7.0.2
./configure
make
sudo make install
```

The default installation path is `/usr/local/bin/`. The configuration file `suricata.yaml` and rules directory will be located in `/usr/local/etc/suricata/`.

A.2.2 Zeek Installation

1. **Install Dependencies:** The following commands must be executed based on the operating system:

- **Ubuntu/Debian:**

```
sudo apt-get install cmake make gcc g++ flex libfl-dev bison
libpcap-dev libssl-dev python3 python3-dev swig zlib1g-dev
```

- **RPM/RedHat-based Linux:**

```
sudo dnf install cmake make gcc gcc-c++ flex bison libpcap-devel
openssl-devel python3 python3-devel swig zlib-devel
```

- **macOS:** The following commands must be executed:

```
xcode-select --install
```

Then, the necessary dependencies can be installed using Homebrew:

```
brew install cmake swig bison openssl
```

2. **Download and Install Zeek:** The latest Zeek release (`.tar.gz` file) must be downloaded from the Zeek download page. The following steps need to be executed:

```
tar xzvf zeek-7.0.1.tar.gz
cd zeek-7.0.1
./configure
make
sudo make install
```

The default installation path is `/usr/local/zeek/`. To change this, the `--prefix` option can be used with the `./configure` command.

A.2.3 Greenbone Community Edition (GVM) Installation

1. **Install GVM:** The necessary command must be executed depending on the operating system:

- **Ubuntu/Debian:**

```
sudo apt-get install gvm
```

- **Other Operating Systems:** The installer can be downloaded and installed using the instructions provided on the official Greenbone Community Edition website: <https://www.greenbone.net/en/greenbone-free/>.

2. **Initialize GVM:** The following command must be executed to initialize and configure GVM:

```
sudo gvm-setup
```

This command performs the following tasks:

- Configures the Greenbone Vulnerability Manager (GVM).
 - Downloads the required vulnerability database.
 - Sets up a default administrator user.
3. **Set Up User and Password:** During the setup process, a default admin user and a randomly generated password are created. These credentials are displayed at the end of the setup process:

```
User: admin
Password: <random_password>
```

If the password needs to be reset or a new user created, the following commands can be used:

- Reset password for the existing `admin` user:

```
sudo gvmc --user=admin --new-password=<your_new_password>
```
- Create a new user:

```
sudo gvmc --create-user=<username>
sudo gvmc --user=<username> --new-password=<your_password>
```

4. **Start GVM:** The GVM services need to be started with the following command:

```
sudo gvm-start
```

5. **Access the Web Interface:** A web browser must be used to navigate to `https://127.0.0.1:9392`. The credentials displayed during the setup or the ones configured manually can be used to log in.

A.2.4 Metasploit Installation

1. **Install Dependencies:** The system needs to be updated, and the required packages installed:

```
sudo apt-get update
sudo apt-get install curl gpg gnupg2
```

2. **Install Metasploit:** The following command must be executed:

```
curl
https://raw.githubusercontent.com/rapid7/metasploit-framework/master/msfupdate
| sudo bash
```

Alternatively, the installer can be downloaded from <https://www.metasploit.com/>.

3. **Start Metasploit Console:** After installation, the following command must be executed:

```
msfconsole
```

A.2.5 Nmap Installation

1. **Install Nmap:** The required command must be executed based on the operating system:

- **Ubuntu/Debian:**

```
sudo apt-get install nmap
```
- **RPM/RedHat-based Linux:**

```
sudo dnf install nmap
```
- **macOS:**

```
brew install nmap
```

A.2.6 TestSSL Installation

1. **Install Dependencies:** The required dependencies must be installed. The following command needs to be executed based on the operating system:

- **Ubuntu/Debian:**

```
sudo apt-get install openssl
```

- **macOS:**

```
brew install openssl
```

2. **Download TestSSL:** The TestSSL repository needs to be cloned:

```
git clone https://github.com/drwetter/testssl.sh.git
```

3. **Run TestSSL:** The following commands need to be executed to navigate to the directory and run the tool:

```
cd testssl.sh
./testssl.sh --help
```

A.2.7 Threat-TLS Installation

The repository must be cloned using the following command:

```
git clone https://github.com/lithekevin/Threat-TLS.git
```

A.3 How to Run the Tool

A.3.1 Setup Greenbone

To start the tool, the Greenbone client must be launched using the following command:

```
sudo gvm-start
```

Once the client has started, the web browser will automatically open at the address <https://127.0.0.1:9392>. Log in using the username and password created during the setup process.

Creating a Target with SSH Credentials

A target represents the system to be scanned. To create a target with SSH credentials:

1. Navigate to **Configuration** → **Targets**.
2. Click on the **New Target** button.
3. Fill in the target name in the **Name** field (e.g., "My Target").
4. In the **Hosts** field, specify the IP address or range of the target (e.g., 192.168.1.1).
5. Under the **Credentials for authenticated checks** section:
 - (a) Click **Add Credential**.
 - (b) Select **SSH** as the credential type.
 - (c) Fill in the required information:
 - **Username:** The SSH username.

- **Password:** The SSH password or private key (if using key-based authentication).
 - **Private Key Passphrase:** If applicable, specify the passphrase for the private key.
- (d) Click **Save** to attach the SSH credential to the target.
6. Save the target by clicking the **Create Target** button.

Creating a Schedule

A schedule defines when scans should occur. To create a schedule:

1. Navigate to **Configuration** → **Schedules**.
2. Click on the **New Schedule** button.
3. Fill in the schedule name in the **Name** field (e.g., "Weekly Scan").
4. Set the **First Time** and **Duration** fields to define the start time and duration of the scan.
5. Specify the recurrence in the **Period** field (For the purposes of the tool it would be preferable to execute a daily scan).
6. Click **Create Schedule** to save.

Creating an Alert

An alert specifies actions to be taken when specific scan conditions are met. To create an alert:

1. Navigate to **Configuration** → **Alerts**.
2. Click on the **New Alert** button.
3. Fill in the alert name in the **Name** field (e.g., "Critical Issues Alert").
4. In the **Condition** section, select Always.
5. Under the **Method** section:
 - (a) Select the HTTP GET notification method.
 - (b) Set the HTTP GET URL as `http://127.0.0.1:5000/api/openvas_alert`
6. Click **Create Alert** to save.

Creating a Task Using the Schedule and Alert

A task specifies the actual scan and uses the previously created target, schedule, and alert. To create a task:

1. Navigate to **Scans** → **Tasks**.
2. Click on the **New Task** button.
3. Fill in the task name in the **Name** field (e.g., "Network Scan").
4. Under the **Target** field, select the target created earlier.
5. Under the **Schedule** field, select the schedule created earlier.
6. Under the **Alerts** field, select the alert created earlier.
7. Click **Create Task** to save.

Once the task is created, it will execute according to the schedule and trigger alerts based on the conditions defined.

A.3.2 Create a NVD API Key

A NVD API Key is necessary to make requests to NVD CVE database, to do so we must visit this page <https://nvd.nist.gov/developers/request-an-api-key> and request a key.

Once received we must insert it in our shell's configuration's file, we open the file, we insert the export string, save and then reload:

```
nano ~/.bashrc
export NVD_API_KEY='api_key_received'
source ~/.bashrc
```

A.3.3 Running the IDS

Threat-TLS relies on an Intrusion Detection System (IDS) to function. Before starting the monitoring process, one of the supported IDS tools must be launched.

- To launch **Zeek**, the following command needs to be executed:

```
sudo zeekctl deploy
```

- To launch **Suricata**, the following command must be used:

```
sudo systemctl start suricata.service
```

A.3.4 Running Threat-TLS

Once all prerequisites are installed, the tool can be started. To launch the tool, navigate to the Threat-TLS directory and execute the following command with **sudo**:

```
cd Threat-TLS
sudo python3 app.py
```

Tool Options

The tool accepts several options that can be specified at runtime to customize its behavior:

- **--IDS=Suricata/Zeek**: This option specifies the Intrusion Detection System (IDS) to be used. The default IDS is Suricata, but this option allows switching to Zeek. For example:

```
sudo python3 app.py --IDS=Zeek
```

- **--json ./pathToConfigJSONFile**: This option allows specifying a whitelist file in JSON format. The JSON file contains configurations or exceptions for Threat-TLS. For example:

```
sudo python3 app.py --json ./config.json
```

- **--attack --host**: This option launches a specific attack against a target server. The **--attack** parameter specifies the type of attack, and the **--host** parameter provides the target server's IP and port in the format IP:PORT.

Supported attack types include: `heartbleed`, `crime`, `padding_oracle_attack`, `lucky13`, `drown`, `sweet32`, `logjam`, `roca`, `bleichenbacher`, `ticketbleed`, `ccs_injection`, `certificate`, `self_signed`, `expired`, `poodle`, `rc4`, `beast`, and `freak`.

For example, to launch a Heartbleed attack against a target server:

```
sudo python3 app.py --attack heartbleed --host 192.168.1.1:443
```

A.4 Testbed Installation

This section provides a step-by-step guide for installing and configuring OpenSSL, Apache2, and Nginx to set up the testbed environment. The instructions include downloading specific versions of the required software and ensuring the proper setup for SSL/TLS testing.

A.4.1 OpenSSL

To install a specific version of OpenSSL, the following prerequisites must first be installed on the system:

```
sudo apt update
sudo apt upgrade
sudo apt install build-essential checkinstall zlib1g-dev -y
```

To download a specific OpenSSL version, visit <https://www.openssl.org/source/old/> or use the following command:

```
sudo wget https://www.openssl.org/source/openssl-1.1.1s.tar.gz
```

This example downloads version 1.1.1s, but the version can be modified to download any available release. Extract the downloaded file with:

```
sudo tar -xf openssl-1.1.1s.tar.gz
```

To build and install the library, execute the following commands:

```
cd openssl-1.1.1s
sudo ./config -fPIC no-shared --prefix=/usr/local/ssl/
--openssldir=/usr/local/ssl/
sudo make clean
sudo make
```

Depending on the OpenSSL version, finalize the installation with one of these commands:

```
sudo make install
sudo make install_sw
```

To enable the installed library for command-line use, add `/usr/local/ssl/lib` to the configuration file by running:

```
sudo nano /etc/ld.so.conf.d/openssl-1.1.1s.conf
```

Reload the configuration with:

```
sudo ldconfig -v
```

Update the environment variable by appending `:/usr/local/ssl/bin` to `/etc/environment`, then reload it with:

```
source /etc/environment
```

To verify the installed OpenSSL version, use:

```
openssl version -a
```

A.4.2 Apache2

To install and configure Apache2 with SSL support, first install the required dependencies:

```
sudo apt update
sudo apt upgrade
sudo apt install libpcre3-dev libaprutil1 libaprutil1-dev libapr1 libapr1-dev
-y
```

Download the desired Apache2 version from <https://archive.apache.org/dist/httpd/> or using:

```
sudo wget https://archive.apache.org/dist/httpd/httpd-2.4.51.tar.gz
```

Extract the downloaded archive:

```
sudo tar -xf httpd-2.4.51.tar.gz
```

Navigate to the extracted directory and build Apache2 with SSL support:

```
cd httpd-2.4.51
sudo ./configure --enable-ssl --with-ssl=/usr/local/ssl/
--prefix=/usr/local/apache2/
sudo make
sudo make install
```

Enable SSL mode by editing the Apache2 configuration file `/usr/local/apache2/conf/httpd.conf` and uncommenting the following lines:

```
Include conf/extra/httpd-ssl.conf
LoadModule ssl_module modules/mod_ssl.so
```

A.4.3 Nginx

To set up and configure Nginx for SSL/TLS testing, start by installing the necessary dependencies:

```
sudo apt update
sudo apt upgrade
sudo apt install build-essential libpcre3 libpcre3-dev zlib1g zlib1g-dev -y
```

Download the desired version of Nginx from <http://nginx.org/en/download.html> or using:

```
sudo wget http://nginx.org/download/nginx-1.24.0.tar.gz
```

Extract the downloaded archive:

```
sudo tar -xf nginx-1.24.0.tar.gz
```

Navigate to the extracted directory and build Nginx with SSL support:

```
cd nginx-1.24.0
sudo ./configure --with-http_ssl_module --with-openssl=/usr/local/ssl/
--prefix=/usr/local/nginx/
sudo make
sudo make install
```

After installation, configure Nginx to enable SSL. Edit the configuration file `/usr/local/nginx/conf/nginx.conf` to include:

```
server {
    listen 443 ssl;
    server_name localhost;

    ssl_certificate /path/to/your/certificate.crt;
    ssl_certificate_key /path/to/your/private.key;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;

    location / {
        root html;
        index index.html index.htm;
    }
}
```

Restart Nginx to apply the changes:

```
sudo /usr/local/nginx/sbin/nginx -s reload
```

Appendix B

Developer's Reference Guide

B.0.1 Suricata

Configuration file:

```
/etc/suricata/suricata.yaml
```

Log file:

```
/var/log/suricata/fast.log
```

Custom rules file:

```
/etc/suricata/rules/tesi.rules
```

B.0.2 Zeek

Configuration file:

```
/usr/local/zeek/etc/zeekctl.cfg
```

Log file for SSL events:

```
/usr/local/zeek/logs/current/notice.log
```

Custom SSL scripts:

```
/usr/local/zeek/share/zeek/site/local.zeek
```

B.0.3 Nginx Settings

To configure Nginx and introduce vulnerabilities, edit the configuration file:

```
/usr/local/nginx/conf/nginx.conf
```

Example configuration for testing:

```
worker_processes 1;

events {
    worker_connections 1024;
}

http {
```

```
include mime.types;
default_type application/octet-stream;
sendfile on;
keepalive_timeout 65;

server {
    listen 80;
    server_name localhost;

    location / {
        root html;
        index index.html index.htm;
    }

    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
        root html;
    }
}

server {
    listen 443 ssl;
    server_name localhost;

    ssl_protocols SSLv2 SSLv3 TLSv1 TLSv1.1 TLSv1.2;
    ssl_certificate cert.pem;
    ssl_certificate_key cert.key;

    ssl_session_tickets on;
    ssl_session_cache shared:SSL:1m;
    ssl_session_timeout 5m;

    ssl_ciphers EXPORT:ALL;
    ssl_prefer_server_ciphers on;

    location / {
        root html;
        index index.html index.htm;
    }
}
}
```

After editing, restart Nginx to apply changes:

```
sudo /usr/local/nginx/sbin/nginx -s reload
```

Bibliography

- [1] S. Turner, T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC-6176, 1995, DOI [10.17487/RFC6176](https://doi.org/10.17487/RFC6176)
- [2] A. Freier, P. Karlton, P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", RFC-6101, 1996, DOI [10.17487/RFC6101](https://doi.org/10.17487/RFC6101)
- [3] T.Dierks, C. Allen, "The TLS Protocol Version 1.0", RFC-2246, January 1999, DOI [10.17487/RFC2246](https://doi.org/10.17487/RFC2246)
- [4] T.Dierks, E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC-4346, April 2006, DOI [10.17487/RFC4346](https://doi.org/10.17487/RFC4346)
- [5] T.Dierks, E.Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC-5246, August 2008, DOI [10.17487/RFC5246](https://doi.org/10.17487/RFC5246)
- [6] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3", RFC-8446, August 2018, DOI [10.17487/RFC8446](https://doi.org/10.17487/RFC8446)
- [7] R. Housley, T. Polk, W. S. Ford, D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", RFC-2459, January 1999, DOI [10.17487/RFC2459](https://doi.org/10.17487/RFC2459)
- [8] D. Eastlake 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC-6066, January 2011, DOI [10.17487/RFC6066](https://doi.org/10.17487/RFC6066)
- [9] D. Eastlake 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC-6066, January 2011, DOI [10.17487/RFC6066](https://doi.org/10.17487/RFC6066)
- [10] Summary on Public Key Infrastructure (PKI) - Scientific Figure on ResearchGate. https://www.researchgate.net/figure/Public-Key-Infrastructure-AppViewX-2020_fig1_351247755
- [11] Eidas Website <https://ec.europa.eu/digital-building-blocks/sites/display/DIGITAL/eIDAS%2BeID%2BProfile>
- [12] D. G. Berbecaru and G. Petraglia, "TLS-Monitor: A Monitor for TLS Attacks," 2023 IEEE 20th Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 2023, pp. 1-6 DOI [10.1109/CCNC51644.2023.10059989](https://doi.org/10.1109/CCNC51644.2023.10059989)
- [13] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1", In: Advances in Cryptology, CRYPTO 1998, LNCS, vol. 1462, Springer, Berlin, Heidelberg, DOI [10.1007/BFb0055716](https://doi.org/10.1007/BFb0055716)
- [14] Schinzel, Sebastian Somorovsky, Juraj meyer, christopher Schwenk, Jorg Tews, Erik weiss, eugen. (2014). Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. https://www.researchgate.net/publication/276418199_Revisiting_SSLTLS_Implementations_New_Bleichenbacher_Side_Channels_and_Attacks
- [15] H. Bock, J. Somorovsky, C. Young, "Return Of Bleichenbacher's Oracle Threat (ROBOT)", 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, Aug 2018, pp. 817-849, ISBN: 978-1-939133-04-5, <https://www.usenix.org/conference/usenixsecurity18/presentation/bock>
- [16] Vaudenay, S. (2002). Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS.... In: Knudsen, L.R. (eds) Advances in Cryptology - EUROCRYPT 2002. EUROCRYPT 2002. Lecture Notes in Computer Science, vol 2332. Springer, Berlin, Heidelberg. DOI [10.1007/3-540-46035-7_35](https://doi.org/10.1007/3-540-46035-7_35)
- [17] Eli Sohl,Cryptopals: Exploiting CBC Padding Oracles DOI <https://www.nccgroup.com/es/research-blog/cryptopals-exploiting-cbc-padding-oracles/>
- [18] Möller, Bodo; Duong, Thai; Kotowicz, Krzysztof (September 2014). <https://web.archive.org/web/20240101001947/https://www.openssl.org/~bodo/ssl-poodle.pdf>

- [19] Savvy Security, 8 Reasons to Implement TLS Security on Your Website <https://cheapsslsecurity.com/blog/reasons-to-implement-tls-security-on-your-website/>
- [20] N. J. Al Fardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 2013, pp. 526-540 DOI [10.1109/SP.2013.42](https://doi.org/10.1109/SP.2013.42)
- [21] Y. Gluck, N. Harris and Angelo Prado. "BREACH: Reviving The CRIME Attack". Blackhat, USA (2013). <https://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>
- [22] S. Kyatam, A. Alhayajneh and T. Hayajneh, "Heartbleed attacks implementation and vulnerability," 2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT), Farmingdale, NY, USA, 2017, pp. 1-6 DOI [10.1109/LISAT.2017.8001980](https://doi.org/10.1109/LISAT.2017.8001980)
- [23] C. Williams, "Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug", Register, Apr 2014, https://www.theregister.com/2014/04/09/heartbleed_explained/
- [24] Adam Langley, "Early ChangeCipherSpec Attack", 2014 <https://www.imperialviolet.org/2014/06/05/earlyccs.html>
- [25] NIST, CVE-2014-0224, <https://nvd.nist.gov/vuln/detail/cve-2014-0224>
- [26] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, J. K. Zinzindohoue, Jean Karim, "A Messy State of the Union: Taming the Composite State Machines of TLS", 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 2015, pp. 535-552, DOI [10.1109/SP.2015.39](https://doi.org/10.1109/SP.2015.39)
- [27] Thái Dùng, "BEAST", 2011 <https://vnhacker.blogspot.com/2011/09/beast.html>
- [28] CVE MITRE, CVE-2011-3389 <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2011-3389>
- [29] Wireshark Web Page, <https://www.wireshark.org/docs/>,
- [30] Wireshark User Guide, https://www.wireshark.org/docs/wsug_html_chunked/,
- [31] Snort Web Page, <https://www.snort.org/>,
- [32] Suricata Web Page, <https://suricata.io/>
- [33] Zeek Web Page, <https://zeek.org/>
- [34] Suricata User Guide, <https://suricata.readthedocs.io/en/latest/index.html>
- [35] Suricata Web Page, <https://suricata.io/download/>
- [36] Zeek Download Page, <https://zeek.org/get-zeek/>
- [37] Nmap Web Page, <https://nmap.org/>
- [38] Metasploit Web Page, <https://www.metasploit.com/>
- [39] Ettercap Web Page, <https://www.ettercap-project.org/>
- [40] Juraj Somorovsky et al, "Systematic Fuzzing and Testing of TLS Libraries" In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, pp. 1492-1504. <https://www.nds.rub.de/research/publications/systematic-fuzzing-and-testing-tls-libraries/>
- [41] TestSSL Web Page, <https://testssl.sh/>
- [42] Qualys SSL Labs, <https://www.ssllabs.com/ssltest/>
- [43] Greenbone Web Page, <https://greenbone.github.io/docs/latest/background.html>
- [44] National Vulnerability Database page, <https://nvd.nist.gov/>
- [45] OpenSSL's vulnerabilities page, <https://www.openssl.org/news/vulnerabilities.html>
- [46] CVE-2012-0884, <https://www.openssl.org/news/secadv/20120312.txt>
- [47] CVE MITRE, CVE-2014-0224, <https://www.cve.org/CVERecord?id=CVE-2014-0224>
- [48] CVE MITRE, CVE-2013-0169, <https://www.cve.org/CVERecord?id=CVE-2013-0169>
- [49] CVE MITRE, CVE-2016-2183, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2183>
- [50] CVE MITRE, CVE-2016-2107, <https://www.cve.org/CVERecord?id=CVE-2016-2107>
- [51] CVE MITRE, CVE-2017-15361, <https://www.cve.org/CVERecord?id=CVE-2017-15361>
- [52] CVE MITRE, CVE-2016-9244, <https://www.cve.org/CVERecord?id=CVE-2016-9244>
- [53] CVE MITRE, CVE-2016-0800, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0800>