POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

TLS-Monitor: An Intrusion Detection-based Monitoring Tool for countering TLS Attacks

Relatori
prof. Antonio Lioy
Dr. Ing. Diana Berbecaru

Giuseppe Petraglia

Anno accademico 2022-2023

Summary

Nowadays, one of the most used protocols is the Transport Layer Security (TLS) protocol. It is a cryptographic protocol designed to provide secure communication between 2 parties by exchanging parameters such as cipher suite, key, and certificate to encrypt the application layer. The first version of this protocol was published in 1995 with the name of SSL 2.0. Over the years the protocol has changed its structure to oppose the discovered attacks against it. After SSL 2.0, SSL 3.0 was published in 1996 to oppose some significant flaws of the previous version. In 1999, the protocol have minor upgrades but change also its name becoming TLS 1.0. This protocol changed other 2 versions TLS 1.1 in 2006 and TLS 1.2 in 2008 before becoming TLS 1.3 which is the version used nowadays since 2018. The TLS attacks discovered over the years harm the protocol in a specific implementation or in its configuration. Nowadays to test the TLS protocol on a specific machine, there are many solutions, like Qualys' SSL Server Test[6], TLSAssistant Tool[7], TLSAttacker, and many other tools that test a host against a specific TLS vulnerability. Unfortunately, these tools give a snapshot of the TLS protocol on the tested machine at that specific moment in time. Nowadays does not exist tool that informs about the machine's vulnerability status over time. A security administrator updates, malicious code, or internal attacks could set the machine prone to attacks at any time. In fact, investigating the default security of the 2 most popular web servers, Apache and Nginx, the researchers have found that the online recommendations are insecure: 89% recommend TLSv1.0, 55% deprecated ciphers, 28% insecure ciphers, and 8% are vulnerable to known attacks[1].

In this thesis, there will be a deep analysis of the TLS protocol, from its well-known and well-documented vulnerabilities to the different ways to mitigate them. With the help of some defensive tools, like Intrusion Detection System (IDS), and some offensive tools, like Metasploit framework, Nmap, and TestSSL, there will be the creation of a new tool: Monitor for TLS attacks. This tool uses IDS to monitor the network traffic with the aim of being able to detect TLS attacks in real-time. When a possible vulnerability is found in a TLS packet, the Monitor for TLS Attacks tool starts the offensive tools that are integrated into it to verify if the TLS vulnerability found is really present. The developed tool supports 2 of the most famous IDS: Suricata[49] and Zeek[50] and it can be used to verify 12 TLS attacks such as Heartbleed, POODLE, and Bleichenbacher. The tool is tested in a lab environment with 3 machines: the possible vulnerable server, the client, and the Monitor. It is tested against all the TLS attacks integrated, installing a vulnerable TLS library on the server and then, establishing a TLS connection between the client and server for each attack 20 times to show how many times the tools employees verify the single vulnerability. This test shows that the tool has good accuracy with integrated attacks. Then the tool has been tested for a stress test with many servers to show the tool's performance with many servers.

Acknowledgements

Contents

Chapter 1

Introduction

1.1 Introduction to TLS

Transport Layer Security (TLS) is a cryptographic protocol widely used to communicate in a secure way on the Internet. It is used also in differennt context, such as inside enterprise services [2], in authentication and authorization framework [3] and in web-based digital identity system [4]. Through different algorithms, it encrypts different messages sent on the Internet to avoid some attackers from easily sniffing and reading confidential data, like passwords, credit card data, or other useful information that can be used against the sender.

The TLS protocol occupies the 5th level of the ISO/OSI model, near the 4th level of Transport Protocol and the Application layer. It is not a single protocol but rather two layers of protocol, as shown in Figure 1.1. The Figure 1.1 shows the TLS protocol from version TLS 1.0 to version TLS 1.2. It consists of 3 higher-layer protocols and one lower-layer protocol. The 3 higher are used in the management of TLS exchanges and are Handshake Protocol; Cipher Change Protocol; Alert Protocol. The Record Protocol is the lower-layer protocol and it provides basic security services to various higher-layer protocols. The figure also shows some of the main protocols that use TLS

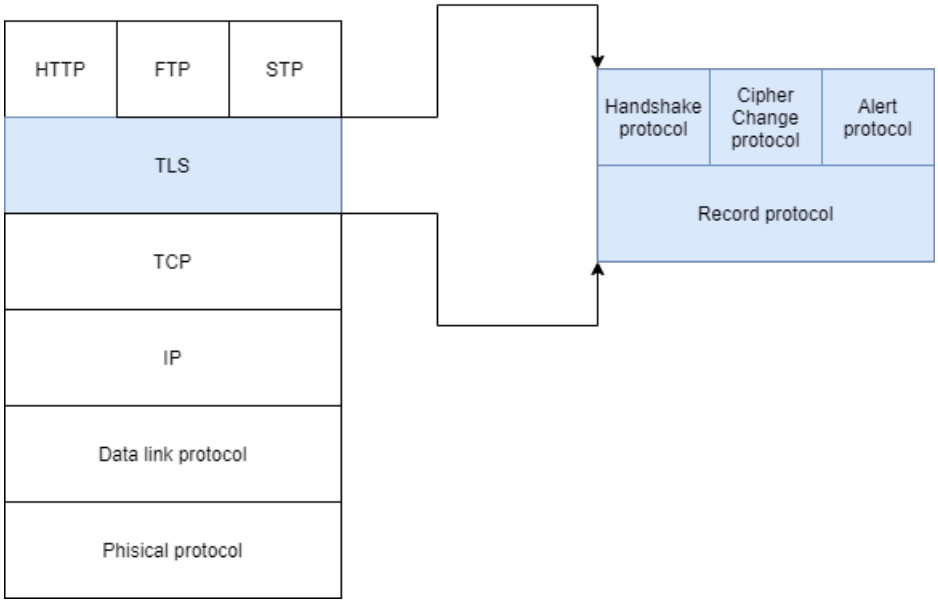


Figure 1.1. TLS protocol in ISO/OSI model from TLS version 1.0 to TLS version 1.2. When the packet is sent, the application layer is encapsulated into the TLS layer. Then the result of the TLS layer is encapsulated into the Transport Layer of the fourth level. This encapsulation is done until the packet reaches the first level

protocol, like File Transfer Protocol (FTP), HTTP, and SMTP, but many other protocols also use TLS to have secure communication.

The connection through TLS is secure because this protocol provides three security properties:

1. **Authentication** - The parties that use this protocol must authenticate themselves. In this way, many attacks that use the impersonation of legitimate users are avoided. For example, the shadow server attack can't be exploited with TLS because the server must authenticate itself and the attacker can't impersonate the server. The authentication of the parties will be explained more in detail in sub-section 1.1.2. This security property can be divided into
   (a) Peer Authentication - This kind of authentication is done in TLS protocol through the exchange of the certificate. The server is forced to send its certificate every time, instead of the client certificate is optional and can be requested by the server;
   (b) Message Authentication - This type of authentication is done in the Finished Message through the MAC.
2. **Integrity** - Verify if the data sent are equal to the data received. It's important to know if the packet was modified before arriving at the destination.
3. **Confidentiality** - Only the sender and the receiver can read the exchanged data. This is due to encrypting the data sent through one between different encrypting algorithms. This avoids a Sniffing attack through which an attacker can read the data if they are sent without encryption.

The TLS protocol provides also protection against Replay and Filtering attacks.

There are 2 important concepts in the TLS protocol:

- **TLS Connection** - a transient TLS channel between client and server. Every connection is associated with one session.
- **TLS Session** - a logical association between a client and a server. The sessions define a set of cryptographic security parameters, which can be shared among different connections. The Handshake protocol creates the sessions.

The following parameters define a session state:

- **Session identifier** - an arbitrary byte sequence used to identify an active o resumable session state. It is chosen by the server.
- **Peer certificate** - identifies the certificate of the peer. It can be null or an X509.v3 certificate.
- **Cipher spec** - specifies the encryption and hash algorithm used for MAC calculation.
- **Master secret** - a secret of 48-byte shared between the client and the server.
- **Is resumable** - a flag used to know if the session can initiate a new connection.
- **Compression method** - specifies the algorithm used to compress the data.

The following parameters define a connection state:

- **Server and client random** - Sequence of bytes chosen for each connection by the server and the client.
- **Server and client MAC secret** - The key used in MAC operations on data sent by the server and the client. For the server is a secret key, instead, for the client is a symmetric key.

- **Server and Client key** - the symmetric encryption key used. The server key is used to encrypt data sent by the server and decrypt by the client, instead, the client key is vice-versa.
- **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 is useful to avoid replay attacks.

1.1.1 History of TLS - from SSL to TLS

The first version of TLS was proposed in 1995 by Netscape Communication under the name of Secure Socket Layer (SSL). The first version of SSL exposed to the public was SSLv2, but unfortunately, this version had a short life because it had a lot of issues and for those reasons in 1996 was released SSLv3. These 2 versions of the SSL protocol were deprecated in 2011 and 2015 because they are evaluated as insecure implementations.

In 1996 was released the first version of the TLS protocol: TLS 1.0. With this version, the SSLv3 wasn't changed drastically but it changed significantly to not allow the interoperability between TLS 1.0 and SSLv3. (although TLS 1.0 does incorporate a mechanism by which a TLS implementation can back down to SSLv3 as written in RFC2246)[10]. The first version of TLS had a long life, in fact, it stayed for 6 years since when the second version of TLS was released in 2006: TLS 1.1. It didn't change a lot from the previous version. It fixes some issues of the CBC mode to avoid some attacks and enforce the utilization of the Initialization Vector (IV) [11]. After 2 years in 2008, TLS 1.2 was proposed to substitute the previous version to oppose some attacks that afflict the previous version, such as Sweet32. This new version changed significantly from the version before. The major changes from RFC5246[12] are the follows:

- The MD5/SHA-1 combination in the pseudorandom function (PRF) has been replaced with cipher-suite-specified PRFs.
- TLS_RSA_WITH_AES_128_CBC_SHA is now mandatory to implement cipher suite.
- IDEA and DES cipher suites are deprecated.
- Added HMAC-SHA256 cipher suites.
- Alerts MUST now be sent in many cases.
- Addition of support for authenticated encryption with additional data modes

Finally, in 2018 the last version of the protocol was released. TLS 1.3 is the safest version of the TLS protocol, due to the continuous fix done to it. In fact, many RFCs in these years update this version of TLS to obtain the version everyone uses on the Internet every day. TLS 1.3 is introduced by the RFC 8446[13] that also has introduced some updates affecting TLS 1.2:

- A protection mechanism against the version downgrade.
- RSASSA-PSS schemes are defined.
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

1.2 Description of TLS's layers

In this section, there is a description of the layer that compose the TLS protocol. This description exposes in deep all the phases of the TLS protocol and each message exchanged between the 2 parties involved. All the phases and the messages described in this section are not part of TLS 1.3 that will been describe in a specific section.

1.2.1 TLS Handshake Protocol

The handshake protocol is an important part of the TLS protocol and it is used before any application data is transmitted. This protocol allows the client and the server to authenticate each other and negotiate the encryption algorithm, MAC algorithm, and cryptographic keys. These negotiated parameters will use to encrypt the data sent in the TLS record. The Handshake Protocol is different for the TLSv1.3 and this version will be covered more in detail in Section 1.1.5, specifically dedicated to this version. For all the other versions this protocol is the same and it is explained in this section. The negotiation of the previously cited parameters is due to the exchange of different messages between the client and the server. These messages have 3 common fields [17]:

1. **Type (1 byte)** - Indicates one of the messages shown in Table 1.1[12][9][10][11][15].
2. **Lenght (3 bytes)** - Indicates the length of messages in bytes.
3. **Content (≥0 bytes)** - Indicates the parameters associated with the message.

Figure 1.2 shows an example of messages exchanged between a client and server in a network. It is a server hello message and the are the 3 common fields.

Table 1.1. List of Handshake Types with their byte value until TLS 1.2.

Description	Value
hello_requested_RESERVED	0
client_helo	1
server_hello	2
certificate	11
server_key_exchange_RESERVED	12
certificate_request	13
server_hello_done_RESERVED	14
certificate_verify	15
client_key_exchange_RESERVED	16
finished	20
certificate_url(21)	21
certificate_status(22)	22

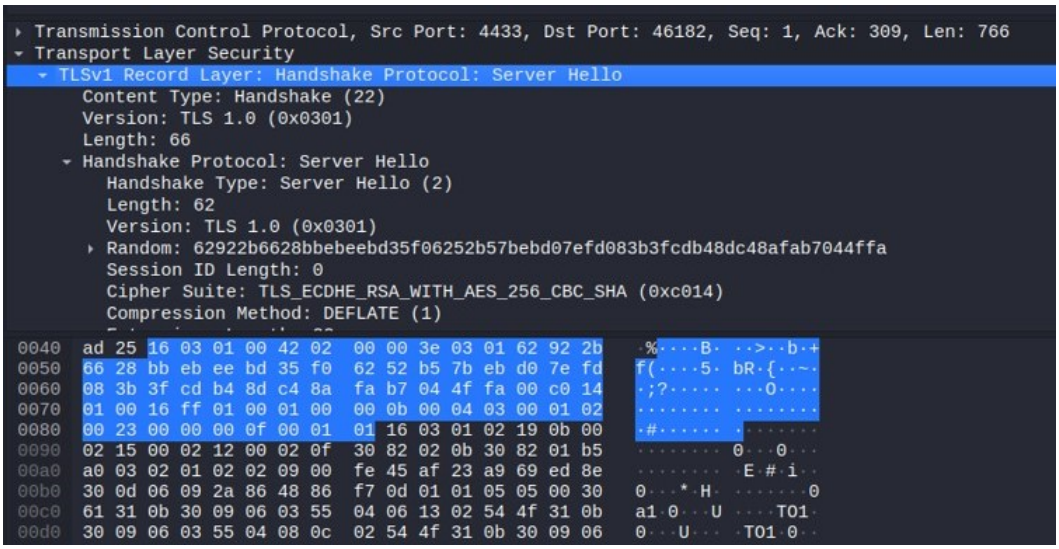


Figure 1.2. Packet sniffed on a network with Wireshark

The Handshake Protocol starts after establishing a TCP connection with the 3-way handshake. As shown in figure 1.3, this protocol can be divided into 4 logical phases[17]:

Phase 1 - initiates a logical connection and established the security capabilities that will be associated. The exchange is initiated by the client through the **client_hello** message with the following parameters:

- **Version** - The highest TLS version supported by the client.
- **Random** - 32 bytes generated by the client. These 32 bytes are divided into 32-bit (4 bytes) for the timestamp and 28 bytes generated by a secure random number generator. These values are used as nonce and they are useful during the key exchange to avoid replay attacks.
- **SessionID** - A byte sequence to identify the session. A non-zero value indicates that the client wants to update the parameters of an existing connection or to create a new connection on the session identified by the value of the sequence. A zero value means that a client wants to start a new connection on a new session.
- **CipherSuite** - This is a list with all the cryptographic algorithms supported by the client
- **Compression Method** - This is a list with all the compression methods supported by the client

After sent the **client_hello** message, the client waits for the **server_hello** message. The **server_hello** message has the same parameters of the **client_hello** message. The Version field is set with the highest version of TLS supported by the server and the client. If the server doesn't support any version available on the client the TLS connection is stopped, by sending an alert message from the server to the client and all the connection is terminated. The Random field on the **server_hello** message is generated by the server and it is independent of the client's Random. If the SessionID was a non-zero value, the same value is used to the server, instead in case, the SessionID was zero, the field on the server-side contains a value for the new session. For the CipherSuite and Compression Method fields, the server selects one of the algorithms and methods sent before by the client.

Phase 2 - This can be seen as the Server Authentication phase. The server sends the **certificate** message if it needs to be authenticated. This message is required if the server selects a cipher suite that uses fixed Diffie-Hellman authentication because in this case the client needs the public key of the server and this key is in the X.509 server's certificate sent through this message. The **server_key_exchange** message is also an optional message and it is sent in the following cases:

1. **Anonymous Diffie-Hellman** - in this case in the message there are 2 parameters that represent the global Diffie-Helman values and the server's public Diffie-Helman key.
2. **Ephemeral Diffie-Hellman** - in this case in the message there are all the 3 parameters of Anonymous Diffie-Hellman, plus a signature of those parameters
3. **RSA key exchange** - This is the case where the server uses the RSA but has a signature-only RSA key. In this case, the server creates a temporary RSA key pair and uses this message to send the temporary RSA public key. The parameters of this message are the exponent and the modulus of the temporary RSA public key, plus a signature of those.

The **server_key_exchange** message is not used if the Server is sent a certificate message with the Diffie-Helman parameters and if the RSA key exchange is to be used. If the server not using Anonimous Diffie-Helman can request the client certificate with **certificate_request** message. This message has 2 parameters: the certificate_type and the certificate_authorities. The certificate_type indicates the public-key algorithm and its use:

1. RSA for signature only

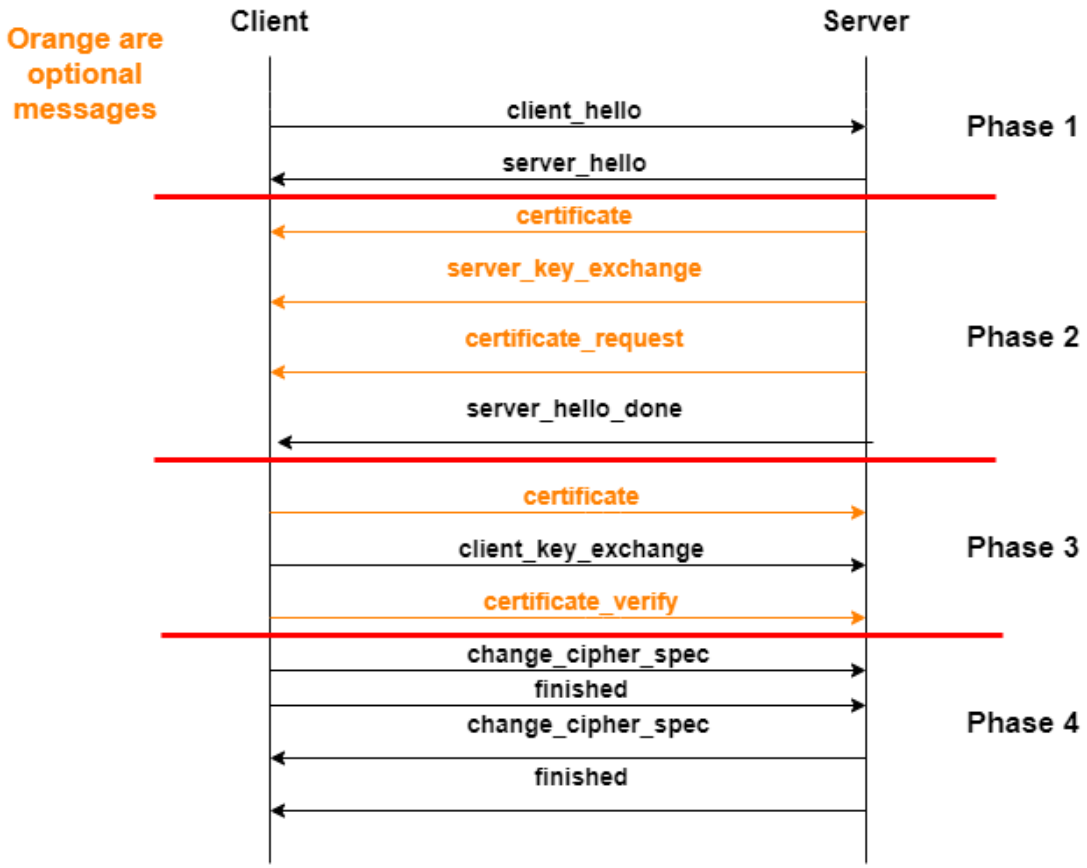


Figure 1.3. Handshake Protocol for version until TLSv1.2 (Inspired by Source [17])

2. RSA for fixed Diffie-Helman
3. DSS for signature only
4. DSS for fixed Diffie-Helman

The certificate_authorities is a list of the accepted Certificate Authority (CA). The second phase finishes with the **server_hello_done** message which indicates the end of the messages sent by the server. After this message, the server will wait for a client's response.

Phase 3 - In this phase, the client verifies the server's certificate if sent and verifies if the parameters sent in the **server_hello** are satisfactory. If requested by the server, the client sends its certificate. If a certificate is not available on the client side an alert message is sent to the server. In case of good results of the verifies the client sends the **client_key_exchange** message. The content of the message change based on the type of key exchanged:

- **Fixed Diffie-Helman** - The content of this message is null because the client's public Diffie-Helman parameters are sent in the **certificate** message
- **Anonymous or Ephemeral Diffie-Helman** - The content of this message is the client's public Diffie-Helman parameters
- **RSA** - The content of this message is a 48 bytes pre-master secret, generated by the client. This content is encrypted with the temporary RSA key sent by the client with **server_key_exchange** message or with the server's public key sent in the **certificate** message.

Phase 4 - It is the last phase of the Handshake Protocol. The client sends the **change_cipher_spec** message that triggers the change of the algorithms to be used for message protection. Theoretically, this message is not part of the Handshake Protocol but is part of the Change Cipher Spec Protocol. After this message, the client sends the **finished** message, which is the first message protected by the negotiated algorithms. Once the **finished** message arrives at the server, it sends also the **change_cipher_spec** and **finished** message. The content of the **finished** message is:

$$PRF(master\_secret, finished\_label, MD5(handshake\_messages)||SHA{-}1(handshake\_messages))$$

The finished_label is "client finished" for the client and "server finished" for the server. After this point, the client and the server may begin to exchange application-layer data.

1.2.2 TLS Record Protocol

The TLS Record Protocol provides:

1. **Confidentiality** - This protocol uses the shared secret key negotiated into the Handshake Protocol to encrypt the messages
2. **Message Integrity and Authentication** - This protocol is responsible to verify the integrity of the messages exchanged between the party, thanks to the shared secret key negotiated during the Handshake Protocol which is used to form a message authentication code (MAC)

As shown in the figure 1.4 it allows:

- The fragmentation of the sent packets and the defragmentation of the arrived packets. The maximum size of each block is 2^{14} bytes.
- The compression of the sent packets and the de-compression of the arrived packets. From TLSv1.2 no compression algorithms are specified, so the default compression algorithm is null. The compression can not increase the content length by more than 1024 bytes.
- The encryption is done on the compressed message plus the MAC using a symmetric key. The encryption can not increase the message length by more than 1024 bytes. This means that the total length of the message must not exceed 2^{14} + 2048 bytes (1024 bytes for the compression and 1024 bytes for the encryption). If the protocol uses block encryption before encrypting the block padding is added to the block.

The final step of the Record Protocol is to prepend a header to the block encrypted. If no extensions are added during the TLS Handshake Protocol, the content types can be: *handshake*, *alert*, *change_cipher_spec*, *application_data*. If the content type isn't matched with the extensions exchanged during the handshake an alert message is sent.

1.2.3 TLS Alert Protocol

The Alert Protocol is used to handle the alert and the error of the TLS Protocol. Like the other applications that use TLS, the alert messages are compressed and encrypted. Like The Handshake Protocol, the Alert Protocol has the 3 common fields: Content-Type, Version, and Length. As shown in figure 1.5 the first byte can be:

- **Warning (1)** - in this case, the browser shows the message related and the user chooses if continue the connection.
- **Alert (2)** - in this case, TLS terminates immediately the connection. Other connections in the same session may continue, but no new connection can start.

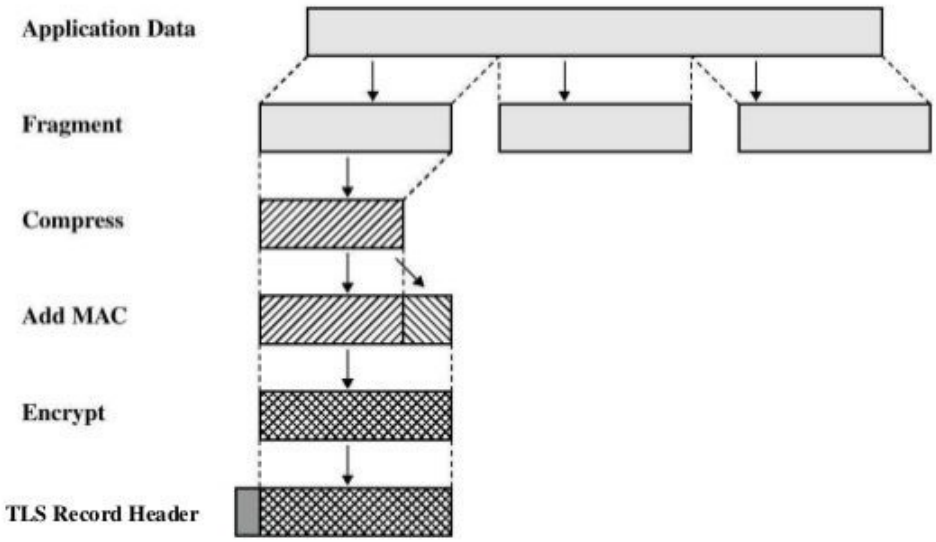


Figure 1.4. TLS Record Protocol Operations (Source[17]). In the newer version, the operations of the TLS Record protocol are less, to speed up the protocol. The compression phase is dismissed to oppose some TLS attacks that exploit it.

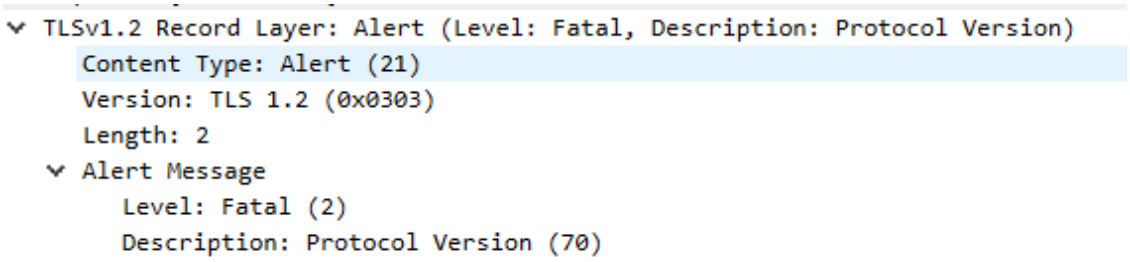


Figure 1.5. An example of an Alert message in Wireshark

The second byte is a code to identify the alert generated and it is set in the Description field. Following there are some alerts that terminate always the connection with the associated value:

- **unexpected_message (10)** - an inappropriate message was received.
- **bad_record_mac (20)** - An incorrect MAC was received.
- **decryption_failed (21)** - An error occurs during the decryption phase. The error can be occurred for the padding value or for the block length.
- **record_overflow (22)** - Occurs when the cyphertext decrypted exceed the $2^{14} + 1024$ bytes or for a ciphertext received greater than $2^{14} + 2048$.
- **decompression_failure (30)** - The decompression function received an improper input. It can be done to a value greater than the maximum allowable or because the function is unable to decompress the block.
- **handshake_failure (40)** - The sender of this message is unable to choose from the proposed option available.
- **illegal_parameter (47)** - A field in a handshake message is inconsistent or out of range.
- **unknown_ca (48)** - The certificate is not accepted because the CA is not in the set of the trusted CA or it can not be located.

- **access_denied (49)** - When access control was applied after the receiving of a valid certificate, the sender of this message decides to not proceed with negotiation.
- **decode_error (50)** - A message can not be decoded because the length of the message exceeds the maximum or a field of the message is out of its specific range.
- **protocol_version (70)** - The version that the client wants to use for the negotiation is not supported.
- **insuffcient_security (71)** - This alert is a specification of the handshake_failure alert, in fact, it is generated when the server wants ciphers more secure than the cipher offered by the client.
- **internal_error (80)** - It is sent when an error that is not related to the TLS protocol occurs.
- **missing_extension (109)** - sent by endpoints that are not received in the handshake messages a mandatory extension that must be sent to offer that specific TLS version or to offer the negotiated parameters.
- **unsupported_extension (110)** - sent by endpoints that have received any handshake messages that are prohibited. This alert is also generated if one endpoint does not include any extensions in a ServerHello or Certificate not first offered in the corresponding ClientHello or CertificateRequest.
- **bad_certificate_status_response (113)** - this alert comes when clients request the OCSP response and verify the response requested. If it isn't satisfactory this alert is generated.
- **bad_certificate_hash_value (114)** - alert generated to the server if any retrieved object from that URL doesn't match the hash.

Instead, the following alerts do not terminate always the connection:

- **close_notify (0)** - This alert is sent to inform the receiver that no more messages are sent on this connection. To close the connection both client and server must send this alert. If the message is not sent, a truncation attack can be exploited [5].
- **bad_certificate (42)** - The received certificate is corrupted, for example with a signature that did not verify.
- **unsupported_certificate (43)** - The certificate's type sent is not supported.
- **certificate_revoked (44)** - The signer revokes the certificate
- **certificate_expired (45)** - The certificate was expired.
- **certificate_unknown (46)** - There is an issue that makes the certificate unacceptable.
- **decrypt_error (51)** - This alert comes when a cryptographic operation failed, like decrypting a key exchange, verifying a signature or validating a finished message.
- **user_canceled (90)** - This alert message is sent when for a reason unrelated to the protocol the handshake is canceled.
- **no_renegotiation (100)** - This message is always a warning to mean that the sender is unable to renegotiate.
- **certificate_unobtainable (111)** - this alert is generated when the server is unable to obtain certificates in a given CertificateURL. If the certificate is required the fatal alert is generated, instead, if the certificate isn't required the alert is only logged.
- **unrecognized_name (112)** - this alert is generated when the server understood the ClientHello extension but does not recognize the server name.

- **unknown_psk_idenitity (109)** - sends by the server when the PSK is required and the client doesn't supply an acceptable PSK identity.
- **certificate_required (116)** - Sends by servers when the client's certificate is required but it is not provided.

1.2.4 TLS 1.3

In August 2018 the Internet Engineering Task Force (IETF) releases TLSv1.3 to improve the security of the TLS protocol. The IETF decided to develop this new version of the TLS protocol to reduce the handshake latency, encrypt more in the handshake, and avoid some features that can be exploited by some attackers to known attacks on the TLS protocol. As described for the TLS version until TLS 1.2, Table 1.2 describes the type of handshake messages for TLS 1.3[13][16][14].

Table 1.2. List of Handshake Types with their byte value for TLS 1.3.

Description	Value
hello_requested_RESERVED	0
client_helo	1
server_hello	2
hello_verify_request_RESERVED	3
new_session_ticket	4
end_of_early_data	5
hello_retry_request_RESERVED	6
Unassigned	7
encrypted_extensions	8
certificate	11
server_key_exchange_RESERVED	12
certificate_request	13
server_hello_done_RESERVED	14
certificate_verify	15
client_key_exchange_RESERVED	16
client_certificate_request	17
finished	20
certificate_url_RESERVED	21
certificate_status_RESERVED	22
supplemental_data_RESERVED	23
key_update	24
compressed_certificate	25
message_hash	254
Unassigned	255

To reduce the Handshake latency the TLSv1.3 allows for a 1-RTT (Round Trip Time) handshake. As shown in figure 1.6 the order of the messages is changed. This change is due to reducing the attack surface and speeding up the process. In the version before, as shown on the left-side of figure 1.6, without the renegotiation the RTT was a minimum of 2 before the peers could start sending messages. Now the client sends the **Key Exchange** message before the cipher suite is negotiated. In this way, the server that receives the cryptographic parameters of the client on the first message is able to calculate the keys for authentication and encryption earlier than before. The server sends the **Key share** message to the client to let know him the keys that will use and after the **Finished** server's message, the client starts sending messages in a secure way. The TLSv1.3 allows also a 0-RTT. This handshake mode is for connections where the peers have previously communicated, and in this case, the peers use keys from a previous connection as pre-shared keys. The pre-shared keys can be obtained also externally. In both ways, the client can send data in the first message ("early data"). The PSK is used by the client to authenticate the server and to encrypt the early data.

From the previous version TLSv1.3 has deleted also many items:

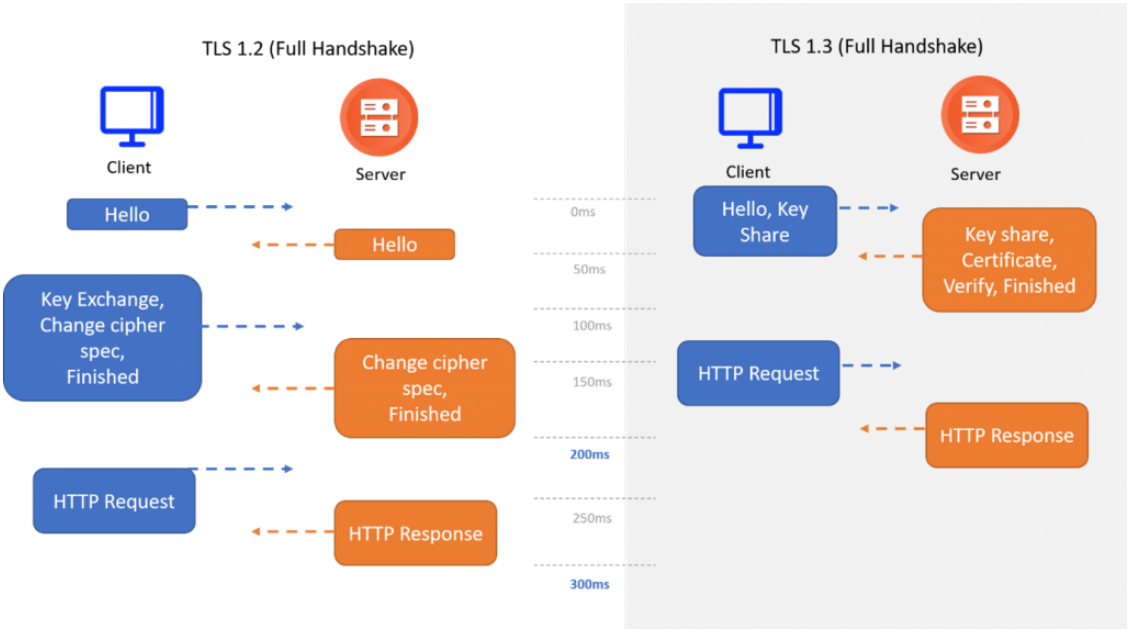


Figure 1.6. A comparison between the Handshake Protocol of TLSv1.2 and TLS1.3 (Source [18])

- Compression
- DH and RSA key exchange
- Ciphers that do not allow authenticated encryption
- Renegotiation
- RC4
- MD5 and SHA-224 hashes with signature messages
- The use of 32-bit timestamp in the **Client Hello** message as part of Client.Random

Most of these items deleted allows many vulnerabilities in the previous version. These vulnerabilities will be detailed in Chapter 2.

Many symmetric encryption algorithms have been removed in this version. All the algorithms that remain are Authenticated Encryption with Associated Data (AEAD) algorithms. This is due because in this version there is the will to separate the authentication and the key exchange mechanism from the record protection algorithms and a hash to be used with both the KDF (key derivation function) and MAC. For this change only these cipher suites are used in this version:

1. TLS_AES_128_GCM_SHA256
2. TLS_AES_256_GCM_SHA384
3. TLS_CHACHA20_POLY1305_SHA256
4. TLS_AES_128_CCM_SHA256

1.3 Public Key Certificate (PKC)

A certificate can be summarized as a block containing the public key and an identifier of the public key's owner. This block must be signed by a trusted third party. This Third-party, typically, is a certificate authority, like a financial institution or a government agency that is trusted by the

community. Anyone that needs a certificate can present his public key to a trusted third party and obtain his certificate. This certificate can be used by the user community without contacting the public-key authority, in fact, anyone that needs the public key can obtain it from the published certificate. Once the peer has obtained the certificate, it can verify its validity by verifying the attached signature of the trusted third party. There are many requirements for this scheme:

- Anyone can read the certificate to know the public key and the name of the certificate's owner;
- Only the Certificate Authority (CA) can read and update the certificate;
- Anyone can verify that the certificate is valid by verifying the signature of the CA on the certificate;
- Anyone can verify the time validity of the certificate;

1.3.1 Public Key Infrastructure (PKI)

The RFC4949[19] defines public-key infrastrcture (PKI) as the set of hardware, software, people, policies and procedures needed to create, manage, store, distribute and revoke digital certificates based on asymmetric cryptography.

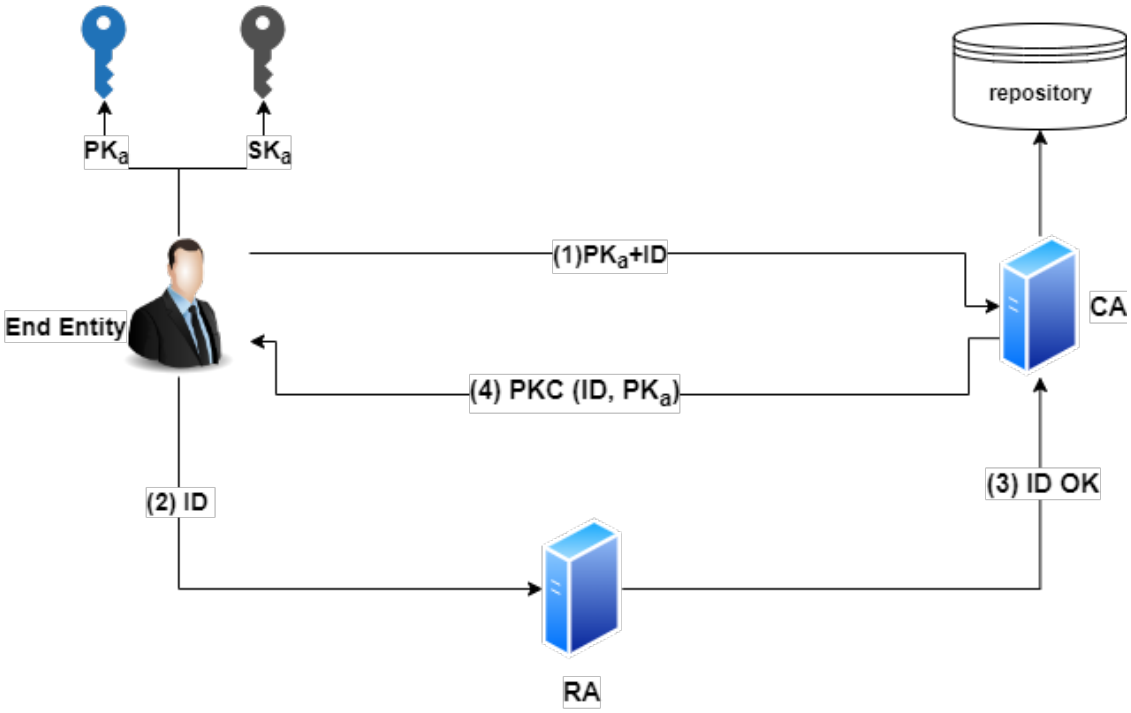


Figure 1.7. Steps for certificate generation

As shown in figure 1.7 there are many elements involved:

- **End entity** - It is a generic device, user, or other entity that is identified by the field in the certificate.
- **Certificate Authority (CA)** - The issuer of the certificate.
- **Registration Authority (RA)** - An optional component that checks the validity of the verifier. It authorizes PKC issuing or revoking based on the result of validation. It is an optional component because the validation process can be done also by the CA.

- **Repository** - A generic method used to store certificates.

The steps for the certificate generation are:

1. The end entity generates a key pair for himself. The key pair consists of a public key PK_a and a private key SK_a. After generating the key pair the end entity sends its public key with an identifier that attests the identity of the end entity. In some cases, the end entity must go physically to an RA that attests the identity of the end entity and gives the identifier to him.

2. RA must check the validity of the identifier and the identity of the requestor depending on the policy which is applied by the RA. The policy is what is needed to verify the identity.

3. After checking the validity of the end entity, the RA sends its result to the CA.

4. If the RA sends a positive result to the CA, the CA creates the certificate for the specific end-entity associating the identity and the public key of the end-entity. The CA sends the new certificate to the owner of the public key, signing the whole certificate with its private key. The CA stores the new certificate in the repository and if the certificate will be revoked, the CA will publish in the repository the CRL.

There are possible other architectures, for example, in which the RA generates the key pair, obtains the PKC, and distributes them on a secure device. This is the case of large companies, where the employees have a badge in which the public key is embedded. If the identity of the end-entity is approved, to speed up the architecture seen in figure 1.7, the RA can generate a code that will be used by the end-entity to authenticate his certificate request. The code generated by the RA has a key (K) that is shared between the CA and the RA to prove the validity of the request. The code can be the following:

$$code = MAC(K, ID)$$

The applications that perform the verification of the certificate do not know the Public Key Infrastructure that issued the certificate. The structure of a PKI defines the certification path that might be followed to validate a certificate. There are various PKI structures:

1. **Hierarchical Structure** - In this structure, the highest level of the hierarchy is the **Root CA**. This particular type of CA has a self-signed certificate and this means that the **Root CA** validates itself. This CA certifies the **Intermediate CA** or **Subordinate CA**. These CAs are on the second level of trust and they certify the **End Entity** certificate. Certification path-building in a hierarchical PKI is a straightforward process that simply requires the relying party to successively retrieve issuer certificates until a certificate that was issued by the Root CA is located.

2. **Mesh Structure** - In this structure, the Root CA does not have only one certificate. It has a self-signed certificate, like in the hierarchical structure, plus a certificate signed from another root. This allows that between two different Root CA will be a unilateral or bilateral trust by issuing a cross-certificate. This structure is not used a lot, because they could be some issues for the application that does not know which certificate trust between different Root CA certificates. There is also an issue with the number of certificates needed for complete trust among all hierarchies(N(N-1)/2).

3. **Bridge Structure** - In this structure, there is another authority: Bridge Certificate Authority (BCA). This BCA is a way to establish trust among multiple PKIs. The BCA cross-certifies with one CA of each participating PKI and each Root CA cross-certifies the BCA. In this way, the number of cross-certified relationships grows linearly.

1.3.2 X.509 Certificate

X.509 is a standard that was initially issued by ITU-T in 1988. Over the years this standard has been changed many times, since 2001 when was introduced X.509 version 3. The X.509 standard is part of the X.500 series that defines a directory service. The directory service is a way to maintain a database of information about the user. The X.509 defines a framework for the provision of authentication services by the X.500 directory to its users. The X.500 directory works like the repository in the figure 1.7, where the certificates containing the public key and the identifier of the user are stored. The X.509 is an important standard because the certificate structure and the authentication protocol defined in this standard are used in many contexts like SSL/TLS and IP Security. It is important to underline that the X.509 standard does not dictate which signature's algorithm must be used, but it is based only on the digital signature and the public key cryptography.

The figure 1.8 shows the evolution of the X.509 standard and the various elements of the certificate:

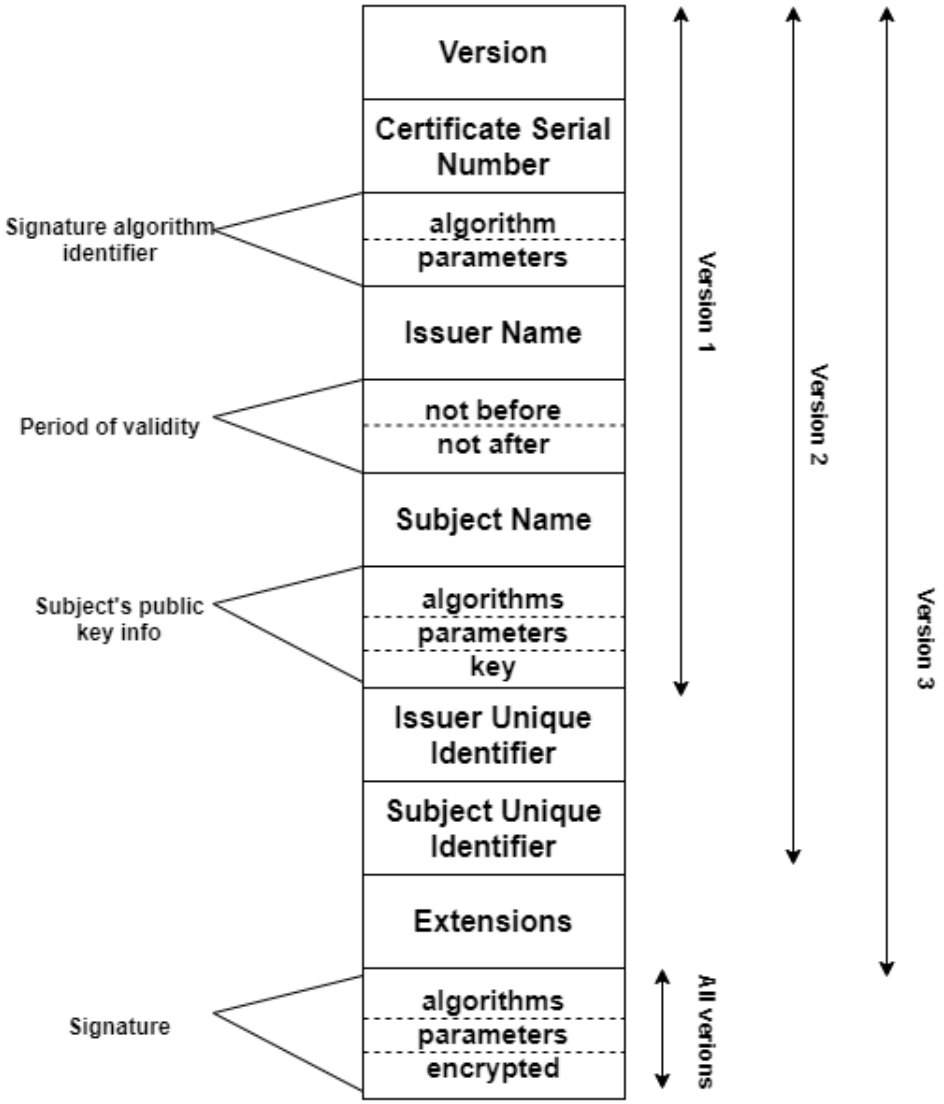


Figure 1.8. X.509 certificate (Source [17])

- **Version** - Represents the version of the standard. The default version is 1 but based on

the fields present on the certificate the version changes. As shown in the figure if the field Issuer Unique Identifier is present the version of the standard is 2. If in the certificate there are some extensions the version of the standard is 3. Until today the version of the X.509 standard is changed but the fields on the certificate do not change since version 3.

- **Serial number** - A unique integer number used to identify the certificate within the issuer CA.
- **Signature algorithm identifier** - In this field, there are the algorithm used to sign the certificate and the parameters used by the algorithm.
- **Issuer name** -The name of the CA that has created and signed the certificate. This is the X.500 name of the CA. This identifier can be considered a Distinguish Name (DN), where the main fields are:
  - **O** - Organization name;
  - **OU** - Organization Unit name;
  - **CN** - Common Name;
  - **L** - Locality name;
  - **ST** - State name;
  - **C** - Country name;
- **Period of validity** - In this field there are 2 dates. One date represents the start date of validity and the other represents the final date of validity.
- **Subject name** - The name of the end-entity to whom the certificate refers. The certificate certifies the end entity that has the private key corresponding to the public key in the certificate.
- **Subject's public-key information** - The public key of the end entity. There is also an identifier of the algorithm for which this key is to be used.
- **Issuer unique identifier** - This is an identifier to uniquely identify the issuing CA in the X.500 directory. This parameter is optional.
- **Subject unique identifier** - This is an identifier to uniquely identify the end entity in the X.500 directory. This parameter is optional.
- **Extensions** - A set of one or more extension fields.
- **Signature** -In this field there is the signature of the whole certificate.

The extensions since version 3 are added to have a more flexible standard. Each extension consists of an extension identifier, a critical indicator, and an extension value. The critical indicator is useful to know if the extension can be securely ignored. For example, if an extension has TRUE as a critical indicator and implementation does not recognize the extension, the certificate will be treated as invalid. The extensions of X.509v3 can be public or private if they are unique for a certain user community.

There are many public extensions and they can be categorized in 4 blocks:

1. **Key and Policy Information** - These extensions add information about the owner of the certificate. These extensions support alternative names, in alternative formats, for a certificate subject or certificate issuer. The additional information can be the address of the subject or a subject's picture. From RFC5280[21] The extensions of this category are:
   - **Authority key Identifier (AIK)** - This is a non-critical field. This extension is used where an issuer has multiple signing keys. This field identifies a specific public key used to sign a certificate by means of a key identifier and a serial number. In some applications this field is important to build the certificate chain, in fact, the keyIdentifier field of the AIK extension must be included in all certificates generated by conforming CAs to facilitate certification path construction.

- **Subject key identifier** - Provides a means of identifying certificates that contain a particular public key. This is a non-critical field.
- **Key Usage (KU)** - Describe for what the key contained in the certificate is used (for example to sign, to sign a certificate, etc.). This field is represented by a bit string, where each bit represents a restriction for usage of the key. For example, if the bit in position 0 is set, the subject public key is used to verify a digital signature. In order of the bit in the bit string, the restrictions that can be indicated are: digital signature, non-repudiation, key encryption, data encryption, key agreement, CA signature verification on certificates, CA signature verification on CRLs, if a key used only for encryption and key used only for decypher. The CA decides if this field must be considered critical or non-critical.
- **Private-key usage period** - Indicates the period of use of the private key corresponding to the public key. This is a non-critical extension.
- **Certificate Policies** - In this field, there is a list of policies, each of which consists of an Object Identifier (OID) and optional qualifiers. In an end-entity certificate, these policies represent the policies under which the certificate has been issued and the purpose under which the certificate may be used. In a CA certificate, these policies limit the set of policies for certification paths that include this certificate.
- **Policy Mappings** - Used for certificates for CAs issued by other CA.

2. **Certificate Subject and Issuer Attributes** -

   - **Subject alternative name (SAN)** - In this field can have one or more alternative names. This field is important for some applications, like electronic mail. EDI and IPSec. This field is always critical if the subject name is empty.
   - **Issuer alternative name (IAN)** - contains one or more different names to identify the CA that issued a certificate or a CRL.
   - **Subject directory attributes** - Coveys any desired X.500 directory attribute values for the subject of this certificate.

3. **Certification Path Constraint**

   - **Baisc Constraints** - Indicates if the subject of the certificate can act as a CA. If it is true, a certification path length may be specified. If it is false the subject of the certificate is an end-entity.
   - **Name Constraints** - This field is used only in a CA's certificate. Indicates a name space that can be certified by the CA.
   - **Policy Constraints** - used to prohibit policy mapping or require that each certificate in a path contain an acceptable policy identifier.

4. **CRL Distribution Point** - defines how the CRL information is obtained. This extension is non-critical but highly recommended.

It is also possible to define private extensions. These extensions are common to a specific group. There are 3 private extensions:

1. **Subject Information Access** - defines the methods to obtain information about the owner of the certificate where the extension appear.

2. **Authority Information Access (AIA)** - indicates how to access information and services for the issuer of the certificate in which the extension appears.

3. **CA Information Access (CAIA)** - indicates how to access information and services for CA that owns the certificate in which the extension appears.

1.3.3 CRL and OCSP

As seen in section 1.1.8 a certificate has a field that indicates the date of validity of that specific certificate. A certificate can become invalid if it is expired and in this case, the field Date Validity helps to know if the certificate is still valid during the exchange of the certificate. But a certificate can become invalid also if it is not expired yet, for example, the private key of the owner of the certificate or the certificate itself can be compromised and in this case, the certificate will become invalid. If for example the private key of the certificate's owner may be compromised an attacker can use the owner's certificate like he owns that and in this way, he can impersonate the certificate's owner. In the same way, if a CA will be compromised, the attacker can generate fake certificates. For these reasons, it was necessary to find a way to keep track of invalid certificates. There are 2 ways with which the end-entity can know if the certificate is still valid:

1. **Certificate Revocation List (CRL)** - a list of revoked certificate. This list is signed by a CA or a delegated authority. The CA issues CRLs for its issued certificate to inform the end entity about the status of that certificate. The CA can delegate this authority to another trusted authority: Revocation Authority. [21] [23] Each CRL has a scope. This scope identifies a set of CRL that could be found in that specific CRL. For example, a CRL's scope can be "all certificates issued by CA X" and in that set of CRL, an end entity can find all the certificates issued by that specific CA. If the scope of the CRL has one or more certificates issued by an entity different from the CRL issuer, in that case, the CRL is an **indirect CRL (iCRL)**. A problem with CRL is managing the list, because it is inefficient to download the whole list at all times. The list can become very big and consequently the time to download and verify it can become unfeasible. To solve this issue there are many solutions:

   (a) Following the first CRL issued, delete the revocation with the expiry date before

   (b) Publish a complete CRL and when the CRL is updated, publish only the difference between the past CRL and the current CRL. This difference between the CRL is called **delta CRL**.

   (c) Partitioning the CRL in groups with the help of the CRL Distribution Point extension.

   An example of the X.509 CRL format is shown in figure 1.9

2. **On-line Certificate Status Protocol (OCSP)** - It is defined in RFC6960[22]. It an online protocol to verify if the certificate is valid at the moment of the request. This is a client-server protocol. The client sends to the server a request about the certificate that he wants to verify and the server responds with the certificate's status. In the sender's request there are the following data:

   - Protocol version
   - Service Request
   - Target Certificate Identifier
   - optional extensions

   It is possible to add more than one certificate to an OCSP request. The OCSP server processes the OCSP request and verifies if:

   (a) the request is well-formatted

   (b) his configuration can provide the requested service

   (c) in the request there is the information needed

   If one of these conditions is not met, the OCSP server will respond with an error message. If all the conditions are met the server will respond with a message and for each certificate sent in the request, the response will have the following fields:

   - Target Certificate Identifier
   - Certificate Status Value

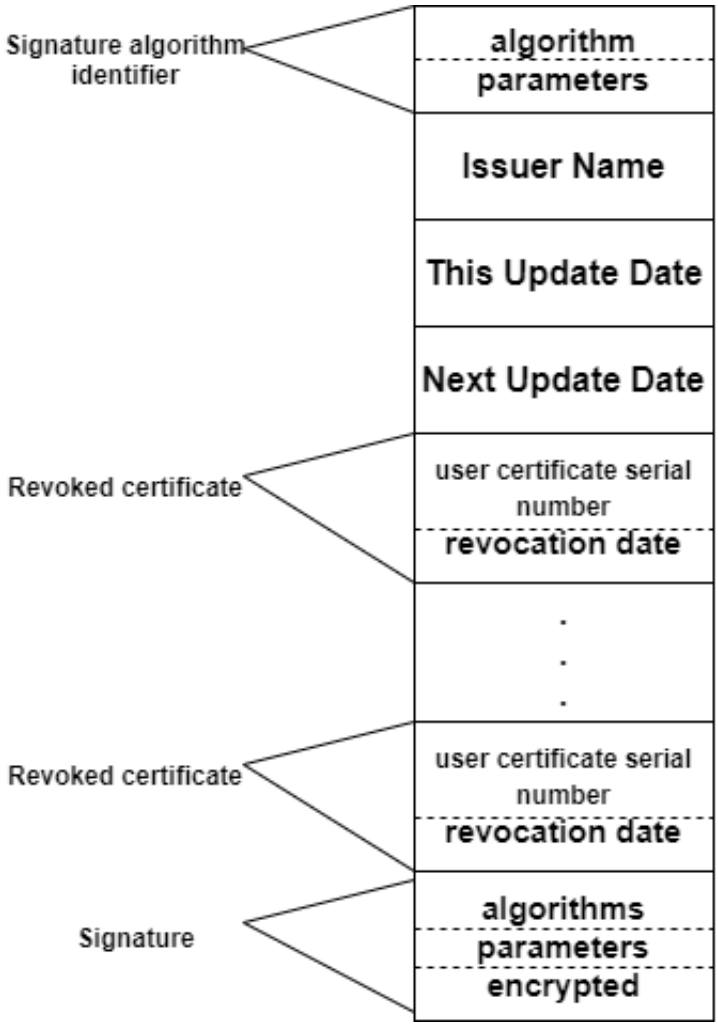


Figure 1.9. Certificate Revocation List (Source [17])

- Response Validity Interval
- Optional Extensions

The Certificate Status Value can be:

- **Good** - this status means that no certificate within its validity interval is revoked.
- **Revoked** - this status indicates that the certificate has been revoked temporarily or permanently. With this status the certificate should be rejected.
- **Unknown** - This status indicates that the responder does not know the requested certificate. This can happen when in the request there is an unrecognized issuer that is not served by the responder.

In case of errors, the OCSP may respond with an error message. These messages are not signed and they are:

- **Unauthorized** - This is an exceptional case and it is sent in cases where the client is not authorized to make this query to this server or the server is not capable of responding authoritatively;
- **malformRequest** - Error due to the request not conforming to the OCSP syntax;
- **internalError** - The OCSP responder reached an inconsistent internal state;

- **tryLater** - This response is sent when the service exists but is temporarily unable to respond;
- **sigRequired** - This exception indicates that the server requires the OCSP request signed by the client.

1.3.4 Certificate Transparency

Certificate Transparency (CT) [24] is an ecosystem that can be used in the PKI system. The CT is an ecosystem built to respond faster to a corruption of a CA or phishing attacks. If a CA was hacked, it can issue the certificates for any website. The communication with the server that owns this certificate is encrypted but the attacker that hacked the CA could intercept the private data and decrypt the communication. In order to respond quickly the CT provides 3 actors:

1. Logs - They are structures where append-only the certificates. They are independent and distributed, in this way, everyone can query them and verify if the certificate is included and when. Log actor uses a Merkle tree to append the certificate. The binary Merkle tree is a binary tree. In the beginning, the log server signs the root of the Merkle tree to obtain a Signed Tree Head. The log appends certificates to a separate Merkle tree hash. Then, the log combines the Merkel tree with the old ones to have a new Merkle tree. This newer Merkle tree hash is then signed to create a new STH.
2. User Agent - Typically they are represented by browsers like Chrome or Apple Safari. These user agents help to perform certificate auditing. They verify if the certificate has been added correctly to a log;
3. Monitors - This actor is responsible to watch for suspicious certificates in the log and it makes sure that all certificates present in the log are visible.

Anyone can submit a certificate to a log server. When the certificate is submitted to a log server, it provides each submitter a promise to log the certificate within a time period called Maximum Merge Delay (MMD). This MMD is usually 24 hours. The promise that the certificate will append to the log is represented by the Signed Certificate Timestamp (SCT). This SCT is returned immediately to the submitters and it accompanies the certificate through its lifetime. There are 3 ways to deliver the SCT for a certificate:

1. X.509v3 extension - As shown in figure 1.10, the domain owner requests the certificate to the Certificate Authority (CA). The CA checks if the domain owner has the right to request the certificate and if it does, the CA creates the precertificate. This precertificate ties the domain to a public key. Then the CA sends the precertificate to the Log server. When the precertificate arrives at the log server, it sends back immediately the SCT to the CA. The CA attaches SCTs to the certificate using x.509v3 extension. Then sends to the domain owner the finished certificate. The SCT can be read after the TLS handshake from client to server in the server's certificate;
2. OCSP - The first steps are equal to the x.509v3 extension, but this time the CA does not create the precertificate. This time the CA sends the certificate to the domain owner and to the Log Server. Then the domain owner requests the SCT through the OCSP query to the CA. The CA responds to the domain owner with the OCSP response where add the SCT. The domain owner sends the SCT to the client through the OCSP stapling during the TLS handshake. This workflow is shown in figure 1.11;
3. TLS extension - Differently from x.509v3 extension and OCSP, this time the domain owner submits its certificate to the Log Server. The domain owner first requests the certificate from the CA. When he receives the certificate from CA, he submits its certificate to the Log server. Then the Log server sends immediately the SCT to the domain user. This SCT is sent during the TLS handshake from the client to the server as TLS extension. The figure 1.12 shows this workflow.

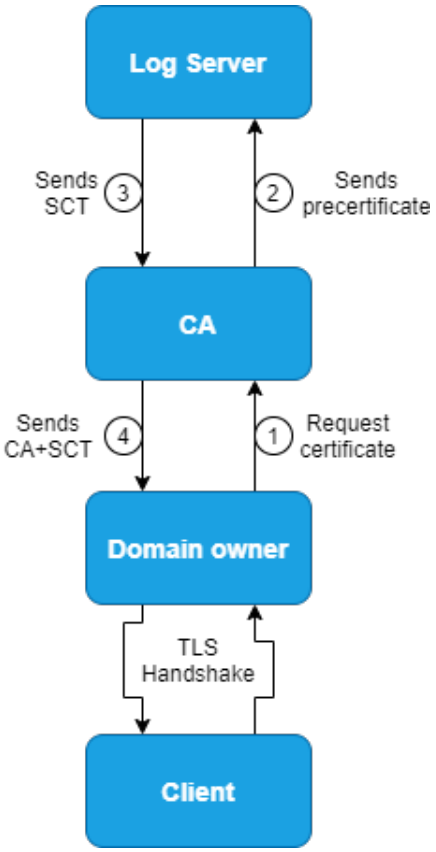


Figure 1.10. SCT in the certificate as X.509v3 extension

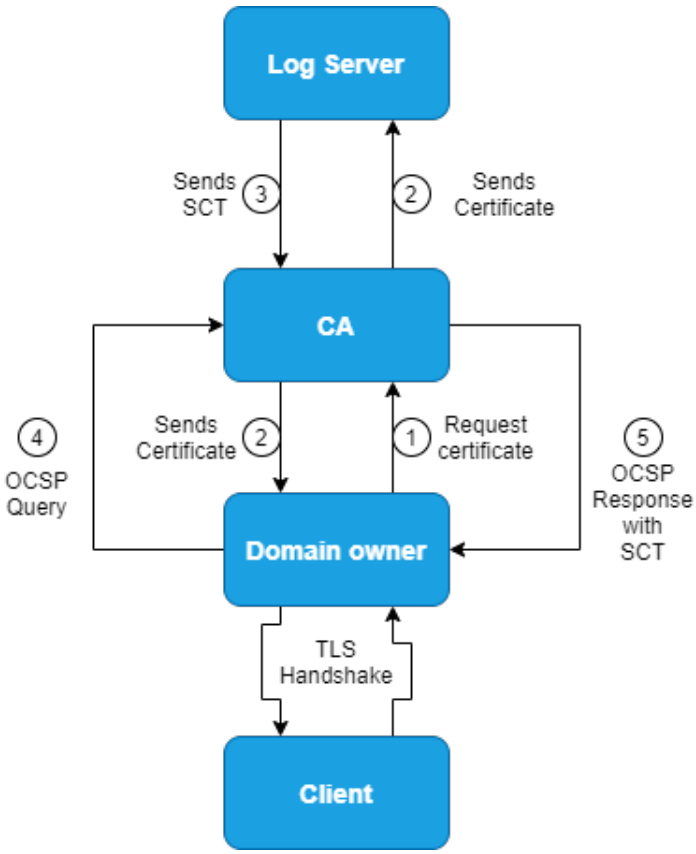


Figure 1.11. SCT through OCSP

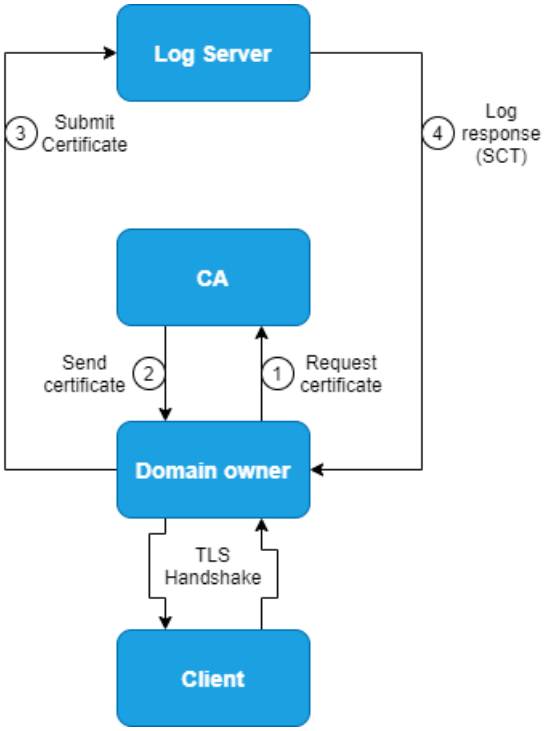


Figure 1.12. SCT through TLS extension

Chapter 2

TLS attacks: study and review

In this Chapter, there is a description of some TLS attacks tool that are discovered over the year. Some of these attacks are integrated into the Monitor for TLS attacks tool, such as Bleichenbacher, Heartbleed, and POODLE, others instead it was not possible to integrate into the tool for attack tools missing or not defined markers. For each described attack, there is also its countermeasure.

2.1 Attack Taxonomy

TLS is widely used worldwide for its interoperability with other protocols, like HTTP, SMTP, etc., and because it allows to exchange messages guaranteeing data's integrity, confidentiality, and peer authentication. The wide use of the protocol makes it possible that more attacks have been exploited against it. Adding to this wide use of the protocol, as shown in Section 1.2, this protocol does not have only 1 layer and this makes it possible to attack the protocol on a bigger surface. For these reasons year after year, researchers study the protocol to find new vulnerabilities to fix them before the attackers can exploit them. The TLS's attacks can be divided into 4 categories [25][26]:

1. **TLS Cryptography attacks** - This category of attacks can be divided into:
   (a) **Core Cryptography** - this category of attacks is focused on cryptographic algorithms' vulnerabilities. The targets of these attacks are algorithms like MD5, DES, RSA, 3DES, or RC4. These attacks can be exploited both during the TLS handshake on the public keys of the entities or on the Record Layer on the symmetric key generated with the parameters exchanged during the handshake.
   (b) **Crypto usage in ciphersuites** - these attacks can be exploited against the mode of the cipher suite rather than the algorithm used. In this category, there are attacks against the Cipher Block Chaining (CBC) mode or the HMAC-SHA1.
2. **TLS protocol functionality** - this category examines the attacks against some functionality of TLS protocol developed to make it more efficient. In this category, there are attacks against the Compression of the TLS Record Layer or against some extensions of the protocol.
3. **TLS library Implementations** - In the years many TLS libraries have been implemented. Nowadays, the most famous are OpenSSL, MatrixSSL, and GnuTLS. These libraries are different among them and there is possible that one library has a bug that the other does not have. In this category, there are attacks against the specific TLS library bug.
4. **TLS configuration attacks** - This kind of attack can be possible against the TLS configuration. For example, if the TLS protocol authenticates the pairs involved with the x.509 certificate, these must be properly configured, otherwise, possible MITM attacks can be possible. If for example, the client doesn't check in proper way the server's certificate, an attacker can impersonate a server and exploit MITM attack.

The countermeasures to these attacks are very different. For example, in the case of the TLS library implementations, it is possible to fix the library's bug with a library update, but in the case of a cryptographic algorithm's vulnerability, sometimes, the only way is to deprecate it.

2.2 Core Cryptography attacks

For this category of attacks, there will be discussed a famous attack against RSA with PKCS#1v1.5. This is the **Bleichenbacher attack**, one of the first attacks exploited against TLS.

2.2.1 Bleichenbacher attack

This attack takes its name from Daniel Bleichenbacher that is the researcher who found it twenty years ago. Daniel Bleichenbacher demonstrates how the PKCS#1 standard is vulnerable to a padding oracle attack in his "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1" [27]. A server, that uses this kind of standard and gives an indication of the plaintext's format associated with a ciphertext, can be attacked with a chosen-plaintext attack. In a chosen-plaintext attack, the attacker sends to the victim a selected cyphertext and the victim sends back to the attacker a plaintext or part of it. This type of attack is called adaptive if the attacker can choose the cyphertext to send depending on the previous result of the same attack. This attack as described by Daniel Bleichenbacher affects also the padding scheme described in the PKCS#1 v1.5 [27], which is one of the most used schemes in RSA to convert short messages into a full-length RSA plaintext. The PKCS#1 v1.5 is used to pad the message before its encryption in RSA algorithm. Given the use of the RSA algorithm will be needed a public key **(N,e)** and the corresponding secret key **N,d**. These 2 keys have a byte length **l** of **N** bytes. Let **k≤l-11** the bytes of a message **m**. The encryption is done as follows:

1. A random padding string is used **PS**. This padding string must not contain 0 values bytes and has a length of **l-3-k≥8**.
2. The message $\mathbf{m^*}$ is set equal to

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

   in figure 2.1 is possible to see an example of message m^*
3. Compute the ciphertext $\mathbf{c = m^{*\,e} \bmod N}$

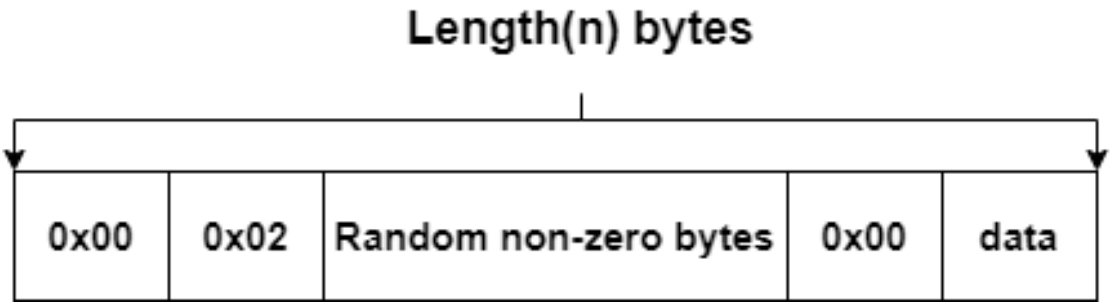


Figure 2.1. An example of message m^*

In the decryption phase the message m_d is computed by $\mathbf{m_d = c^d \bmod N}$. Then m_d will be checked if equal to:

$$0x00||0x02||PS"||0x00||m"$$

In this case, PS" is a string with at least 8 bytes and all of them are non-zero-value bytes. If the check end successfully, the message m" is returned, otherwise the decryption phase fails. The

Bleichenbacher attack or "Million Message" attack allows an attacker to compute an RSA key operation on a message of his choice without knowing the private exponent **d** of the secret key **(N,d)**. For this attack, an oracle is needed. Gives as input a ciphertext, this oracle responds if this ciphertext can be decrypted using the RSA PCKS#1 v1.5. The behavior of the oracle O can be resumed as follow:

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

An attacker can use the oracle O to compute an RSA key operation, such as decryption or a signature without knowing the secret exponent **d** of the secret key **(N,d)**. The attack as written in "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1"Bleichenbacher can be divided in 4 phases:

1. **Binding** - The attacker computes $\mathbf{c^A = c \cdot s_0^e \bmod N}$. Where s_0 is a random integer value, c is an integer representing the ciphertext. The attacker sends all the c^A value computed to the oracle O, until the oracle responds with 1. This means that a random integer value s_0 is found to obtain a c^A value with a valid PKCS#1 v1.5 padding. Now the attacker knows that the first 2 bytes of the string $m \cdot s_0$ are 00 and 02. This defines an upper bound and a lower bound:

   $$m \cdot s_0 \in [2B,3B)$$

   where $B=2^{8(l-2)}$, and this implies that there exist a value **k** such that:

   $$2B \leq m \cdot s_0 - kN < 3B$$

   This can be written as follows:

   $$\frac{2B + kN}{s^0} \leq m < \frac{3B + kN}{s^0}$$

   This phase can be skipped if with $s_0=1$ the ciphertext c is already a PKCS#1 v1.5 conforming.

2. **Range Reduction** - Once found the upper bound and lower bound, the attacker computes $c^A = c \cdot {s_i}^e \bmod N$ to reduce the range of the possible values. s_i is a set of random integer values($s_1, s_2, \ldots, s_n$) until m will be a single candidate. The reduction of the bounds will have only with the value of c^A with which the oracle $O(c^A)=1$.

For N equal to 1024bits Bleichenbacher's attack requires about one million calls to the oracle O. In a realistic scenario the attack can have success also with 3800 oracle queries [28]. A study called "The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations" [28] has shown that the majority of the implementations evaluated in the paper are still vulnerable to padding oracle attacks. Add the situation is worst than before because as shown in the paper the attacks can be made extremely more efficient via careful analysis and parallelization techniques.

In 2018 Hanno Bock et al. discovered a new attack based on Bleichenbachers called **ROBOT (Return Of Bleichenbacher's Oracle Threat)** [29]. Hanno Bock et al. demonstrate in their work how after years a known vulnerable may be not patched, for example, they found with ROBOT attack vulnerability on Facebook and Paypal. This attack suggests 5 test vectors to test if the target is vulnerable to the Bleichenbacher attack. This 5 vectors are:

1. Correctly formatted message - This message contains a correctly formatted PKCS#1 v1.5 padding and the correct position for the TLS version used. An example of the message is:

   $$M1 = 0x0002||pad()||0x00||TLSversion||rnd[46]$$

   where rnd[x] denotes a random no-zero string of length x and pad() indicates a function that generates a non-zeros padding string to have a message conforming to the RSA key length.

2. Incorrect PKCS#1 v1.5 padding - This message starts with incorrect PKCS#1 v1.5, like:

$$M2 = 0x4117||pad()$$

   This should trigger an invalid behavior of the server

3. 0x00 in a wrong position - This is due because the attacker would trigger a different invalid server's behavior, like buffer overflow or an internal error. The message can be:

$$M3 = 0x0002||pad()||0x0011$$

   Must be noted that the PKcs#1 v1.5 padding is in the correct position differently from M2.

4. Missing 0x00 - This is due because for the PKCS#1 v1.5 padding, the 0x00 must be added to correct unpad the message and if the attacker omits this byte the server goes in error. The message can be:

$$M4 = 0x0002||pad()$$

5. Wrong TLS version - The message can be:

$$M5 = 0x0002||pad()||0x00||0x00202||rnd[46]$$

A Server that responds to these messages with the same message is not vulnerable to a Bleichenbacher attack, instead, the Server can be exploited.

2.2.2 Countermeasures

For this kind of attack, there is no generic solution but a single case must be studied to fix it. For example, the Bleichenbacher attack can be easily fixed in many ways:

1. By using the PKCS#1 v2.0 instead of PKCS#1 v1.5. the PKCS#1 v2.0 introduces the **Optimal Asymmetric Encryption Padding (OAEP)** that is another padding scheme not affected by the Bleichenbacher attack;
2. The use of DH or ECDH for key exchange, instead of RSA;
3. With no information to the client about the failure of Pre-Master Secret decryption on the server. This can be done using a random Pre-Master Secret generated by the server after the ClientKeyExchange message that will be sent to the client instead of the Alert message.

Patching a server for the Bleichenbacher attack, this server will be patched also for the ROBOT attack.

2.3 Crypto usage in cipher suites attacks

In this section, there will be introduced first a common attack against the CBC mode called Padding Oracle Attacks [30] and then 2 variants of this attack called POODLE and Lucky13.

2.3.1 Padding Oracle Attack

For this kind of attack the attacker needs 2 things:

1. The Oracle Server must use the CBC mode to encrypt and decrypt the message;
2. An Oracle Server that informs the client about the validity or invalidity of the message's padding.

The CBC mode is used for example with AES encryption algorithm and it has the following steps:

1. It divides the plaintext into N blocks of the same size. The block's size depends on the algorithm used. For example, for the AES-128 each block will have 16-bytes (128-bit). If the bytes of the plaintext are multiple of the bytes used to divide, the plaintext is perfectly divided and another block of padding is added. If the plaintext can not be perfectly divided, some padding bytes are added to the last block to fill it.
2. As shown in figure 2.2 the next step is XOR each plaintext block with the previous plaintext block. The first plaintext block will be XORed with the Initialization Vector (IV).
3. After XORed each block, the result of the XOR operation will be encrypted with the encryption key.
4. Finally each ciphertext block is concatenated to the previous to from the whole ciphertext.

For the decryption phase the operations are equal to the encryption phase but in reverse order, as shown in figure 2.3.

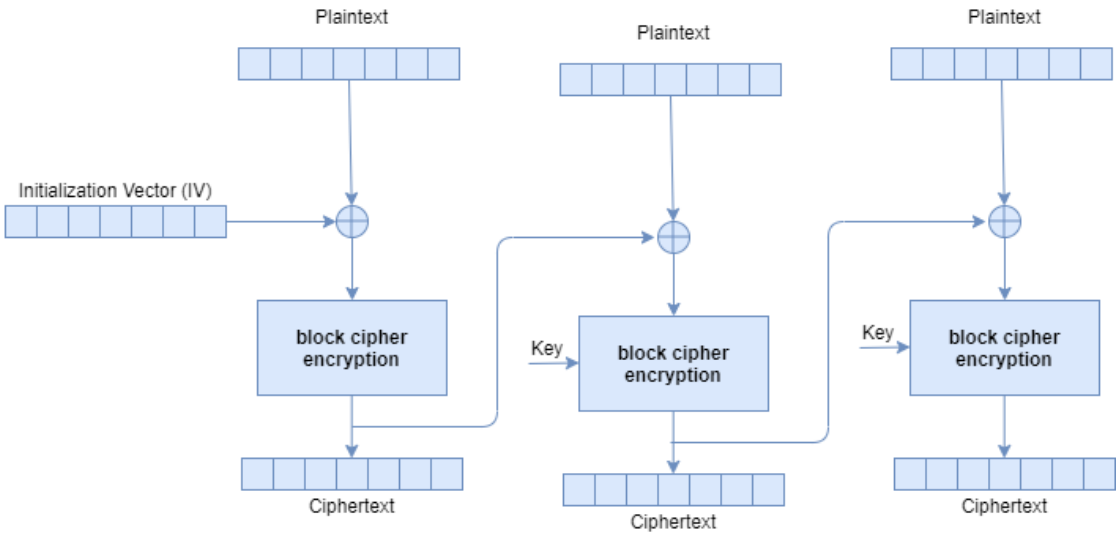


Figure 2.2. CBC mode encryption

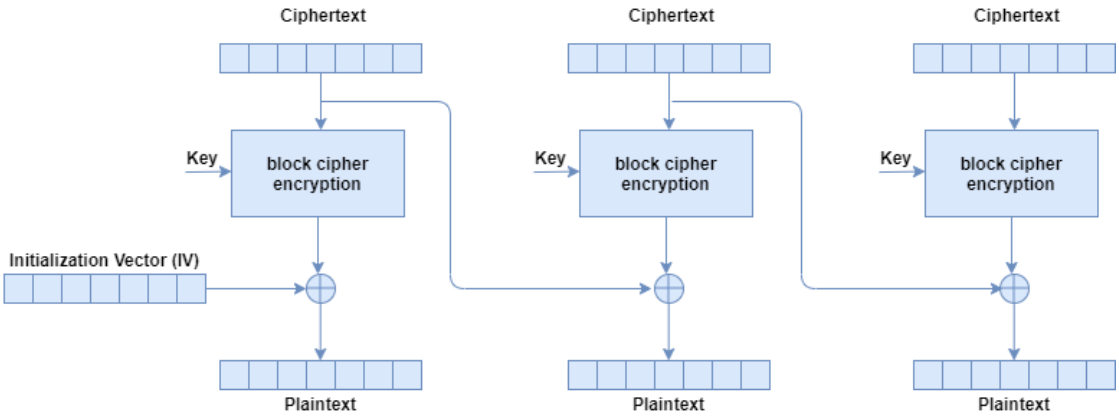


Figure 2.3. CBC mode decryption

To successfully exploit this attack, the attacker manipulates the N-1th block to guess the plaintext of the Nth block. The attacker starts manipulating the last byte of the N-1th block to

know the last byte of the Nth block. In the last block, there is the padding that has a default value depending on how many bytes of padding was been added. For example, if only one byte of padding is needed to fill the last block, the byte added will be 0x01. If 2 bytes of padding is needed, the bytes added will be 0x02, and so on. Knowing this the attacker starts manipulating the N-1th block to have as result the 0x01 byte. To do this he must try at most 256 values (2^8). He tries until the padding is valid or until all values have been tried. If the possibilities are exhausted, the attacker tries with 0x02 and so on, until the padding returned is valid. Once the value is found the attacker knows that the intermediate value (the value obtained after the decryption, see figure 2.3) XORed with the value found is equal to the padding value that he was trying. For example, if after many tries the value found is 0x3A and the attacker found this value for the padding value 0x03, the intermediate value will be:

$$IntermediateValue = 0x3A \oplus 0x03 = 0x39$$

Once the attacker has obtained the intermediate value, to know the plaintext he will XOR the intermediate value with the ciphertext. This operation will do byte after byte until the attacker obtained all the plaintext corresponding to the ciphertext. To obtain the plaintext the attacker needs (b*P*N)/2 tries, where b is the number of bytes per block, P is the number of possible tries and N is the number of blocks.

2.3.2 POODLE

POODLE (Padding Oracle On Downgraded Legacy Encryption) is an attack was discovered in 2014 [31]. To exploit this kind of attack there are 2 requirements to satisfy:

1. The attacker must have done with successful MITM attack to have visibility of the ciphertext exchanged between client and server and control messages sent by the client;
2. The client and the server must support SSL 3.0. This is possible because to have a smooth user experience most implementations of SSL/TLS have backward compatibility with SSL 3.0.

This attack can be divided into 2 phases:

1. **Downgrade phase** - During this phase, the attacker forces the TLS connection to SSL 3.0 version. As shown in figure 2.4, the normal handshake message to start a connection in TLSv1.2 is deleted. In this way, the client tries to establish a TLS connection with ever smaller versions until reaches the SSL 3.0 version. If the Server supports this version the SSL handshake goes on, and the SSL 3.0 connection is established and the cryptographic phase can start. This downgrade attack can be used for other attacks as described in **'What's in a Downgrade? A Taxonomy of Downgrade Attacks in the TLS Protocol and Application Protocols Using TLS'** in 2019 from Eman Salem Alashwali et al [32];
2. **Cryptographic phase** - In this phase, the attacker starts a Padding Oracle Attack to guess sensitive information such as a cookie or password.

These 2 phases must have been done once the MITM attack has been exploited.

On December 2014 a variant of the POODLE attack was discovered[35]. The discovered variant shows that SSL 3.0 doesn't want a particular padding format, so it is possible to decrypt with an SSL3 function with TLS for specific hardware such as F5 BIG-IP LTM. The function doesn't check the padding bytes and this means that was possible to exploit POODLE attack also in TLS connection.

2.3.3 Lucky 13

This attack was discovered in 2013 by Al Fardan, Nadhem J. and Paterson, Kenneth G.[36]. It takes its name from the TLS MAC calculation that includes 13 bytes of header information.

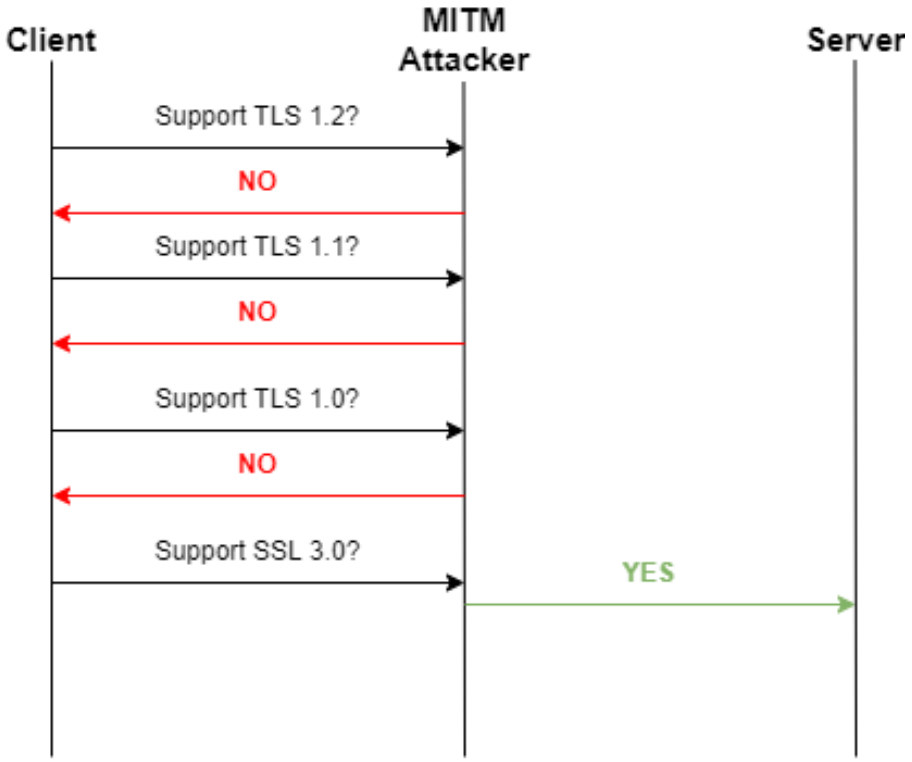


Figure 2.4. POODLE Downgrade phase

These 13 bytes are divided into 5 bytes for the TLS header and 8 bytes of the TLS sequence number. This attack can be exploited in a TLS connection that uses CBC as a mode cipher and the MAC-then-Encrypt scheme. This scheme calculates first the MAC of the plaintext, then appends the result to the plaintext, and finally adds the padding to become an integral multiple of the block length. This scheme is a requirement for this attack because this attack is based on the time taken from a server to respond. If the padding is wrong the server will take more time to respond because of the last bytes that represent the padding length and the padding bytes mismatch. In this case, it will use the entire data to calculate the MAC and this implies much more time. In the case of correct padding, the server will not compute the MAC on the whole message and will respond in less time.

2.3.4 Countermeasures

To mitigate the Padding Oracle Attack there are many ways:

1. Use another block cipher mode of operation instead of CBC that authenticate also the message. This Authenticated Encryption scheme can be GCM or CCM mode for the block cipher. With this solution, all the 3 attacks above are mitigated.

2. The server's response must take the same time if the request failed or successfully ended. In this way, the Lucky13 attack can not be exploited. There are other time attacks but fortunately, there are some countermeasures [37].

3. Do not have an oracle server that responds with a specific error message. If the server responds with a generic message the attacker doesn't know anything about the padding error and the Padding Oracle Attack can't be exploited.

4. For the POODLE attack, some researchers, that discovered the vulnerability, developed a fix. This fix consists of an extension protocol called **TLS_FALLBACK_SCSV** that prevents a protocol downgrade by a MITM attack. This fix is also implemented in the

OpenSSL latest versions. Both Client and Server need to support this extension to prevent the downgrade phase.

2.4 TLS Protocol Functionality

This typology will be explained 4 attacks. The first 3 attacks are similar and they are based on the compression used in TLS. The fourth attack is based on the utilization of PSK (Pre-Shared Key).

2.4.1 CRIME, BREACH, HEIST

The CRIME (Compression Ratio Info-leak Made Easy) technique was categorized as CVE-2012-4929 by MITRE [38]. This attack is a vulnerability in the compression of the SSL/TLS protocol and the SPDY protocol. The attack can leave cookie data vulnerable to session hijacking. A compression used in TLS is the DEFLATE compression that eliminates the duplicated string. As mentioned in the Record Protocol the TLS compression is before the encryption and MAC phase and it is used to delete the redundant data. The figure 2.5 shown represents the CRIME attack to guess a connection cookie. This attack has 3 steps:

1. **Step 1** - the victim visits a website that is owned by the attacker. Inside the website, there is malicious code.
2. **Step 2** - The malicious code established a connection with a third-party website while the attacker execute the attack.
3. **Step 3** - In this step the CRIME attack starts. The attacker enforces the victim to request many HTTP requests to the website forwarded and he inspects the size of the HTTP request after compression. If the length of the compressed content diminishes the content injected by the attacker matched with some part of the secret.

The form of the discovered exploit used JavaScript and needed six requests to obtain one byte. In 2012 SSL Labs tests across the SSL Pulse data set indicate that about 42% of the servers support TLS compression.

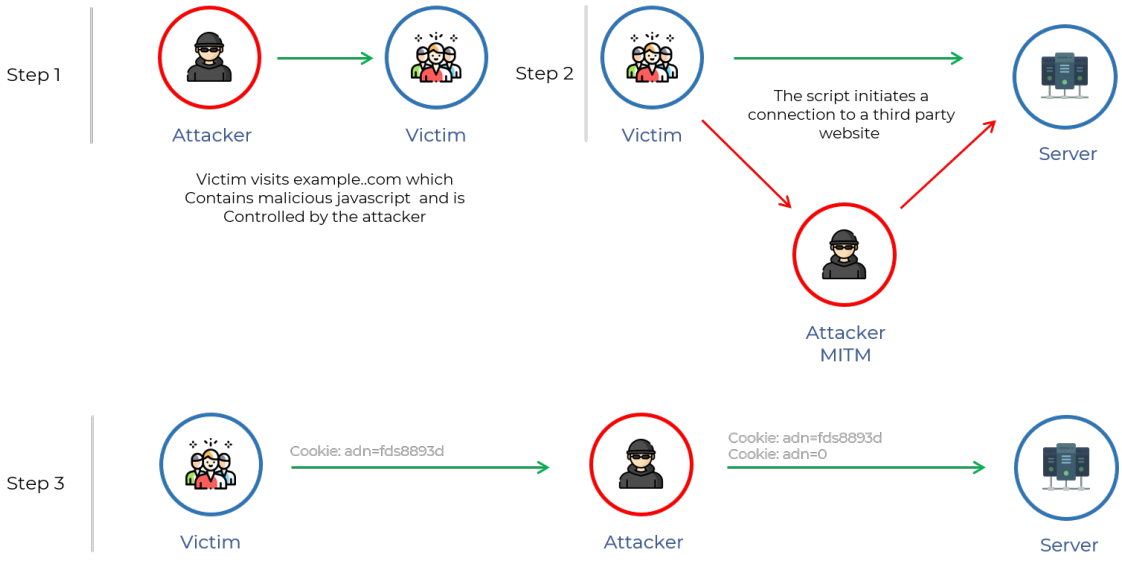


Figure 2.5. CRIME attack (Source[39])

The BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) attack is similar to the CRIME attack because it is also based on compression, but differently from the CRIME attack, the BREACH attack is based on HTTP compression. It can be exploited equally to CRIME attack.

The HEIST (HTTP Encrypted Information can be Stolen through TCP-windows) attack was discovered by Mathy Vanhoef and Tom Van Goethem in 2016 [40]. This more than an attack is a set of methods to increase the attack surface of the above attacks. In fact with HEIST is no more needed to intercept traffic to know the length of request to see if the compression produces more or fewer bytes. With HEIST the length of the message can be seen through the TCP windows.

2.4.2 SELFIE

The SELFIE attack is an important attack because it is the first attack that afflicts TLS 1.3. This attack was discovered by Nir Ducker and Shay Gueron in 2019 [41], the year after the TLS 1.3 release. TLSv1.3 like its predecessors allows two parties to establish a shared session key from an out-of-band agreed pre-shared key (PSK). The PSK allows the parties to skip the certificate verification step because the use of a PSK mutually authenticates the parties under the assumption that no one except the parties knows the PSK.

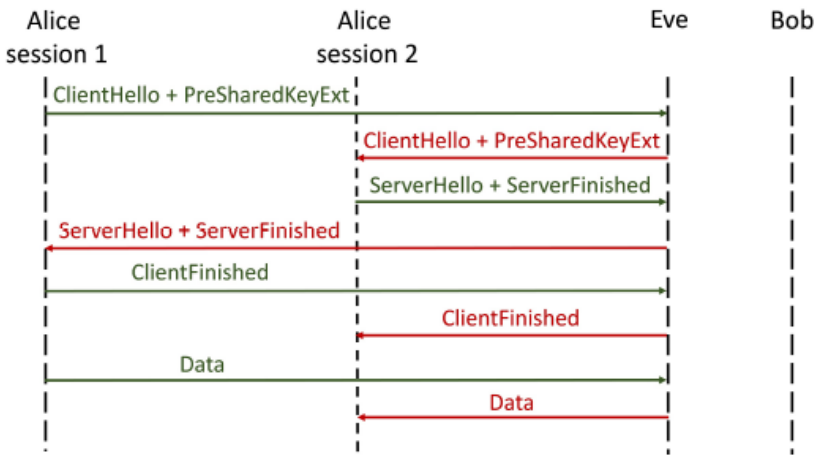


Figure 2.6. SELFIE attack (Source[41])

The figure 2.6 shows how the SELFIE attack works. In the following steps will be used the name Alice, Bob and Eve because there will be 2 sessions and Alice that is the client in session 1 will be the server in session 2. The same happens with Bob that is the server in session 1 and the client in session 2. Eve acts always as the attacker. The steps of this attack are:

1. Alice sends the Client Hello message to Bob with the **pre_shared_key** extension;
2. Eve intercepts the Client Hello message from a client to the server and sends back to the client the same and established with the client a new session (session 2). The client in session 2 receives what has been sent to Bob in session 1 like a Selfie image.
3. In session 2, Alice replies with a Server Hello and Server Finished.
4. The attacker captures the message containing the Server Hello and Server Finished and sends back the message to Alice in session 1.
5. Alice in session 1 as she have done in session 2, authenticates the message because only Bob has the PSK that allows him to send the correctly authenticated message.

6. After the authentication Alice, in session 1, replies to Bob with a Client Finished message.
7. Eve intercepts the message and sends back to Alice the Client Finished message. At this point, Alice has opened a TLS session with herself, a Selfie session.
8. Once the Selfie session is established, every message that Alice sends to Bob in session 1 is returned back to Alice in session 2.

This attack is based on 2 properties of TLS:

- A client can open parallel independent connections;
- The client does not explicitly check the identity of the server. It verifies only if the server is a legitimate owner of the PSK.

2.4.3 Countermeasures

To prevent users from the CRIME attack, SSL compression must be disabled. The latest version of Apache disables compression by default, but if you use Apache version 2.4.3 you must add the command 2.4.3 in the setting file. The default path for the file configuration with Apache2 is `/etc/apache2/mods-enabled/ssl.config`. Also Nginx is vulnerable to CRIME attack in the old version. The secure versions of Nginx are:

- 1.0.9 (if OpenSSL 1.0.0+ used)
- 1.1.6 (if OpenSSL 1.0.0+ used)
- 1.2.2
- 1.3.2

```
SSLCompression off
```

For the BREACH attack disabling the TLS compression is useless because the attack is based on HTTP compression. To mitigate this attack there many ways:

1. The easiest way is to disable the HTTP compression, but this impact a lot on the performance;
2. Protection of the web page through CSRF protection (Cross-site Request Forgery);
3. Hide the length of the response adding a random byte to the response;
4. Create random secrets for each request, don't use the same secrets.

To mitigate the SELFIE attack there are 2 ways:

1. External PSKs must be used together with the server certificate.
2. A PSK must not be shared between more than one client and one server.

The first way is not a good solution because a TLS certificate may cover a large number of servers and the attacker can redirect the connection from one server to the other. The first way is not a good option also because the PSKs are introduced to the protocol in order to avoid certificates. The second solution is the best because it is simple and completely mitigates the SELFIE attack.

2.5 Implementation - libraries attacks

To this typology of attacks will be presented 2 bugs in one of the most used libraries for the TLS protocol: OpenSSL. The attacks belonging to this library are HEARTBLEED and EARLY CCS.

2.5.1 HEARTBLEED

The Heartbleed attack was discovered in 2014 and now it is also known as CVE-2014-0160 [42]. This was an important attack because it affected 17.51% of the trusted HTTPS website. This attack was possible for a bug in the OpenSSL library, more precisely there was a bug in the Heartbeat extension of the OpenSSL library. In figure 2.7 is possible to see the structure of the HeartBeat extension. This extension is used to keep alive the connection, so the client sends to the server a payload of arbitrary data and the server sends back the exact copy of data to prove that everything is fine. The client's Heartbeat Message is sent via SSL3_RECORD structure that is possible to see in figure 2.8.

```
struct
{
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[HeartbeatMessage.payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;
```

Figure 2.7. The struct of the Heartbeat message written in C. (Source[43])

```
struct ssl3_record_st
{
  unsigned int length;    /* How many bytes available */
  [...]
  unsigned char *data;    /* pointer to the record data */
  [...]
} SSL3_RECORD;
```

Figure 2.8. SSL3_RECORS structure. (Source[43])

As shown in figure 2.8 the SSL3_RECORD has 2 main attributes: **length** and ***data**. The first is the length of the payload sent in the Heartbeat Message and the second is a pointer that points to the start of the received Heartbeat Message. Meanwhile, the attribute **payload_length** in the figure 2.7 is the number of bytes in the arbitrary payload that has to be sent back. The attack is possible for a bug in the code that does not verify if the bytes requested by the client are equal to the bytes that the server will send.

Due to this bug, an attacker can send to the server a Heartbeat Message with a single byte of payload but the attacker can modify the **payload_length** field to claim 65535 bytes. Once read the **payload_length** field, due to the fact that there is no control, the victim sends back 65535 bytes. The victim's Heartbeat Response will contain bytes coming from the victim's memory and this will bring to leakage of information. The attack is shown in figure 2.9

2.5.2 EARLY CCS

This attack was discovered by Masashi Kikuchi di Lepidum in 2014 [44]. This bug has existed since the very first release of OpenSSL. The problem is that OpenSSL accepts ChangeCipherSpec (CCS) inappropriately during a handshake. The ChangeCipherSpec must be sent before the FINISHED message in the TLS handshake. OpenSSL sends CCS in exact timing itself, but it

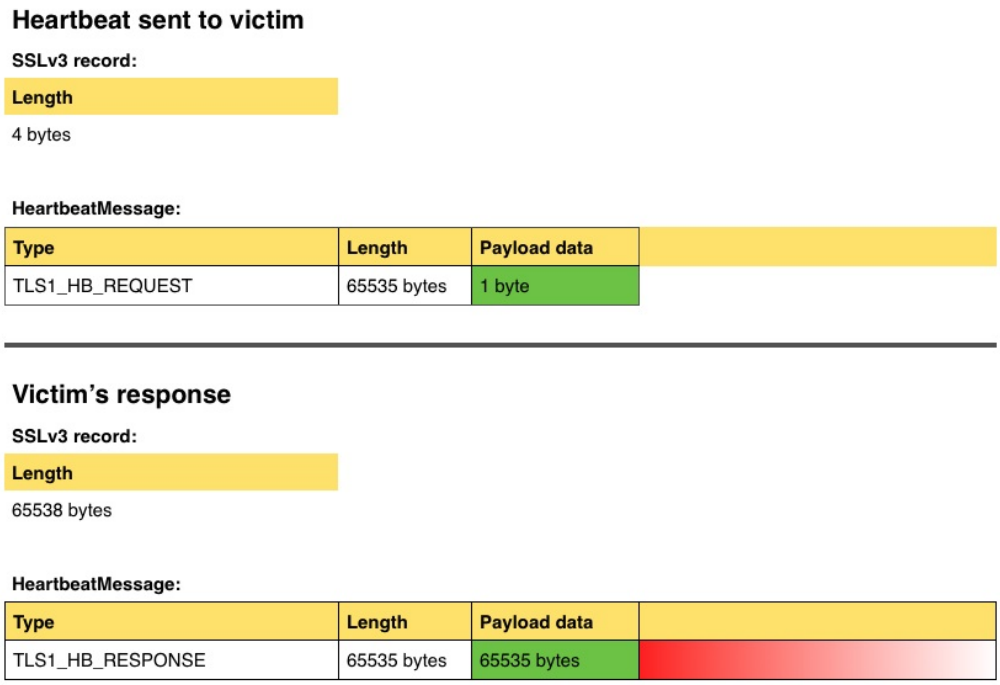


Figure 2.9. HEARTBLEED attack (Source[43])

accepts CCS at other timings when receiving. An attacker can exploit this bug by modifying or decrypting data. Since the first versions of OpenSSL if a CCS message is sent before the master secret generation and after the Server Hello message, the keys and the FINISHED hash will be calculated only on public information because the master secret is null. If the client will receive a second CCS message this belongs only to recalculating the FINISHED hash but not to the recalculation of the keys. The hash of the FINISHED message with the correct master secret impedes the attacker to generate a FINISHED hash acceptable. From version 1.0.1 of OpenSSL, a patch allows the server to calculate a FINISHED hash acceptable also in case it has processed the CCS mistakenly. The bug from the server side impedes receiving 2 CCS messages in the same handshake. The server will process only the first. If an attacker sends a CCS before the normal CCS message, the server can not receive a second CCS message that it can use to calculate the correct FINISHED hash. With OpenSSL 1.0.1 the server will use the anticipated CCS message of the attacker to calculate the FINISHED hash. From the client side, the code allows the client to receive 2 CCS messages in the same handshake. If an attacker is MITM, he can send a CCS message to the client and the server to fix the keys with public information without an error on the FINISHED hash. If the attack is successfully exploited the attacker can decrypt and modify the communication between client and server. To exploit this attack both the client and the server must be vulnerable.

2.5.3 Countermeasures

To mitigate the Heartbleed attack is sufficient to install the OpenSSL patch. Since 2014 the versions of OpenSSL has patched the bug that exploits the attack. Following there is the status of different versions of OpenSSL[42]:

- OpenSSL 1.0.1 through 1.0.1f (inclusive) are vulnerable
- OpenSSL 1.0.1g is NOT vulnerable
- OpenSSL 1.0.0 branch is NOT vulnerable
- OpenSSL 0.9.8 branch is NOT vulnerable

And the operating system distributions that could be vulnerable to Heartbleed attack:

- Debian Wheezy (stable), OpenSSL 1.0.1e-2+deb7u4
- Ubuntu 12.04.4 LTS, OpenSSL 1.0.1-4ubuntu5.11
- CentOS 6.5, OpenSSL 1.0.1e-15
- Fedora 18, OpenSSL 1.0.1e-4
- OpenBSD 5.3 (OpenSSL 1.0.1c 10 May 2012) and 5.4 (OpenSSL 1.0.1c 10 May 2012)
- FreeBSD 10.0 - OpenSSL 1.0.1e 11 Feb 2013
- NetBSD 5.0.2 (OpenSSL 1.0.1e)
- OpenSUSE 12.2 (OpenSSL 1.0.1c)

For the EARLY CCS attack, OpenSSL clients are vulnerable in all versions of OpenSSL. Servers are only known to be vulnerable in OpenSSL 1.0.1 and 1.0.2-beta1. Users of OpenSSL servers earlier than 1.0.1 are advised to upgrade as a precaution. to mitigate this attack:

- OpenSSL 0.9.8 SSL/TLS users (client and/or server) should upgrade to 0.9.8za.
- OpenSSL 1.0.0 SSL/TLS users (client and/or server) should upgrade to 1.0.0m.
- OpenSSL 1.0.1 SSL/TLS users (client and/or server) should upgrade to 1.0.1h.

2.6 TLS configuration attacks

As mentioned at the beginning of the Section, this kind of attack can expose to Man-In-The-Middle Attack. In this section are described 2 of the most famous TLS MITM attacks exploited in TLS history: FREAK and BEAST.

2.6.1 FREAK

FREAK ("Factoring RSA Export Keys") attack is a Man-In-The-Middle (MITM) attack discovered by Beurdouche et al. [33] in 2015. With a MITM attack, an attacker can intercept, modify or delete the messages in a conversation between a client and a server. The MITM attack is a typology of attacks that works if a client and a server don't authenticate themselves and this allows the attacker to impersonate a server or a client as shown in the figure 2.10. The Web application of the figure can be also a server that a user wants to connect to. The goal of the MITM in the FREAK attack is to change the cipher suite from a strongest one to a weak one, in this case, a special cipher suite called EXPORT grade cipher. This kind of cipher suite is special because it is an old cipher suite used in the 90' when the US government had a regulation that

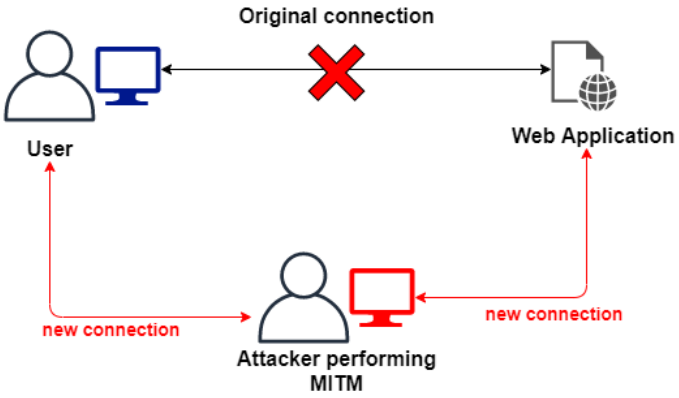


Figure 2.10. Man-In-The-Middle attack

any software exported outside of the US can have a maximum of 512 bits of RSA key and 40 bits of RC2/RC4. Nowadays cipher suite with these few bits can be broken in 1 hour. Since 2000 the EXPORT cipher suite has not been used by browsers.

The phases of FREAK attack can be shown in figure 2.11. A normal client starts the TLS handshake with the Client Hello message where inside there are some parameters such as Client Random, all supported client ciphers, and supported curves. The MITM intercepts the Client Hello message and modifies the client's supported ciphers, deleting all the ciphers except the EXPORT ciphers. After the modification, the attacker sends the message to the Server as a normal message. If also the Server supports the EXPORT ciphers, it sends the Server Hello to the client with the EXPORT cipher and in this way, when the message comes to the client, it generates a weak shared key that will be used to encrypt the future messages with the server. In fact, as shown in phase 3 the attacker can easily crack the weak shared key and modifies the Finished message.

2.6.2 BEAST

BEAST stands for Browser Exploit Against SSL/TLS[45]. This attack was discovered by Phillip Rogawayin in 2002 and then it was performed in 2011 by Thai Duong and Juliano Rizzo. This attack exploits network vulnerabilities in TLS v1.0 and older versions. The goal of this attack is to force the client and the server through a MITM attack to use the weakly cipher of the TLSv1.0 or older protocols. This attack is like POODLE because also in this attack there is a downgrade phase where the attacker forces the end-entity to use a weak version of TLS. This attack is possible only if the client and server support TLSv1.0.

2.6.3 Countermeasures

The FREAK attack nowadays is patched by removing the EXPORT cipher from all servers and browsers, but the typology of MITM attacks is nowadays a problem. To mitigate the BEAST attack there are different ways:

1. Turn off the TLSv1.0 and the older protocols. This can be done in apache by editing the *ssl.conf* file with:

   ```
   SSLProtocol all -SSLv3 -TLSv1.0
   ```

   End then restart httpd. In NGINX to turn off the TLSv1.0 and older versions is possible by editing the *nginx.conf* file:

   ```
   ssl_protocols TLSv1.2;
   ```

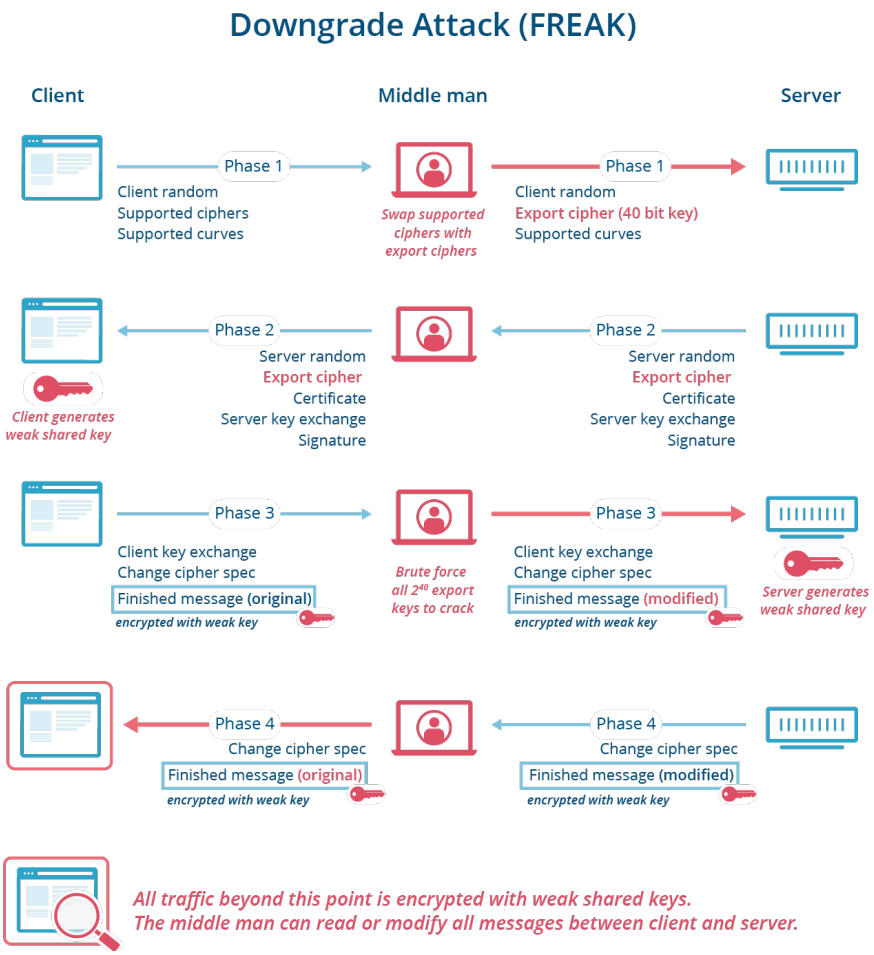


Figure 2.11. FREAK attack's phases (Source[34])

2. Do not use CBC mode and RC4 cipher. Both are unsafe and can be used to exploit this attack.

Chapter 3

TLS scanning tools

This Chapter presents an overview of the main TLS scanning tools used nowadays to verify the TLS configuration of the servers: SSLLab and TLSAssistant. This tool as mentioned before, captures a picture of the TLS server's configuration at that specific moment.

3.1 SSLLabs

Qualys Incorporate provides vulnerability management solutions using a SaaS model (Software as a Service). It was founded in 1999. Qualys starts the SSL Labs which is a collection of documents, tools, and thoughts related to SSL. SSL Labs is a non-commercial research effort and it is an attempt to better understand how SSL is deployed and an attempt to make it better. There are many projects of Qualys SSL Labs like SSL CLient Test which test the browser of the user that visits the link, or SSL Pulse which is a continuous and global dashboard for monitoring the quality of SSL / TLS support, based on Alexa's list of the most popular sites in the world. There is also a project called SSL Server Test [6] which is a free online service that performs a deep analysis of the configuration of any SSL web Server on the public internet. The user has to insert only the URL of the tested web server and the SSL Server Test returns a deep analysis of the Certificate used, Protocol Support, Cipher strength, and key exchange. In the figure 3.1 is shown only the summary of the *www.google.com* URL but this free tool gives a lot of information.

3.2 TLSAssistant

TLSAssistant is a tool written in Bash to assist average system administrators and app developers to deploy resilient instances of the TLS protocol[7]. The tool takes in input the IP address of the target that will be evaluated and outputs a single report file. The tool has been designed to be modular and easily upgradable. The TLSAssistant v2 will be released soon [51], but in this thesis, the TLSAssistant v1 will be described. Figure 3.2 shows the tool's architecture, which is divided into:

1. **Analyzer** - Takes as input a series of parameters depending on which analysis the user wants to run. In this version, the TLSAssistant integrates 3 tools:
    - testssl.sh - scans the server to find many vulnerabilities. There are a lot of tools used to scan targets for TLS vulnerabilities but this tool covers a wide range of vulnerabilities, as shown in the figure 3.3.
    - 3SHAKE checker - tool added to make the Analyzer able to test a target against the Triple Handshake Attack.
    - Malloroid - this is a tool written in Python that performs static analysis on the code of an Android application. It takes in input an *.apk* file and then decompiles the

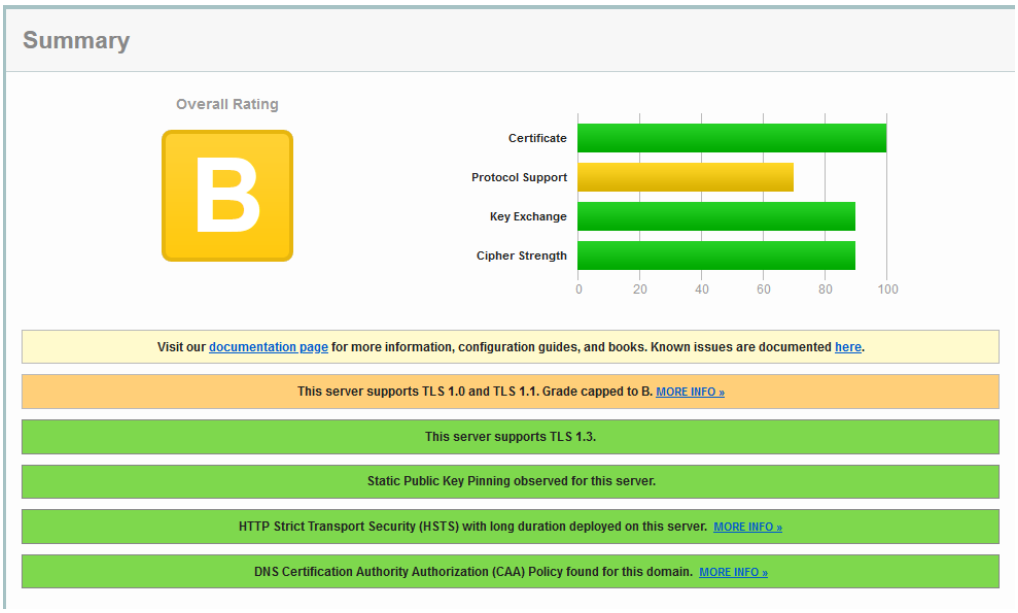


Figure 3.1. Result of TLS protocol test for www.google.com visited on 23-06-2023

application. Once decompiled, it extracts the set of URLs that the app uses and checks the validity of their certificates.

2. **Evaluator** - Once the Analyzer finishes starting all the tools, it collects all the reports and transmits theirs to the Evaluator. This component is responsible for the enumeration of the detected vulnerabilities and the generation of the report. It's possible to split this component into 2 subcomponents:

   - Vulnerability enumerator - collects and analyze all the report generated by the Analyzer. This subcomponent makes a list of all the vulnerabilities discovered;
   - Report handler - Takes the list of the previous subcomponent and renders the final report based on the system administrator's choices.

3.3 Advantages and Disadvantages

Both the previously described tools are very efficient and test the target in a very deep way against a lot of TLS vulnerabilities. The SSL Test of Qualys is simple to use because it can be used without an installation and he has only to insert the URL of the target to test and it returns all the vulnerabilities found, a lot of information on the target tested, and a grade for the security of the target. Some disadvantages in using the SSL Test of Qualys are:

- The tool can't be used to test a server in a local network detached from the public internet. For example, a system administrator can't test his developed server before attaching it to the public internet.
- The tool does not scan an IP address. The target must have a URL.
- The scan is not in real-time, but it takes a snapshot of the server's configuration at that moment.

In conclusion, the SSL Test Qualys is a very good tool for labs or educational purposes but can't be used for development. The TLSAssistant differently from the SSL Test can be used for development because it can test a target against a lot of TLS vulnerabilities also if it is detached

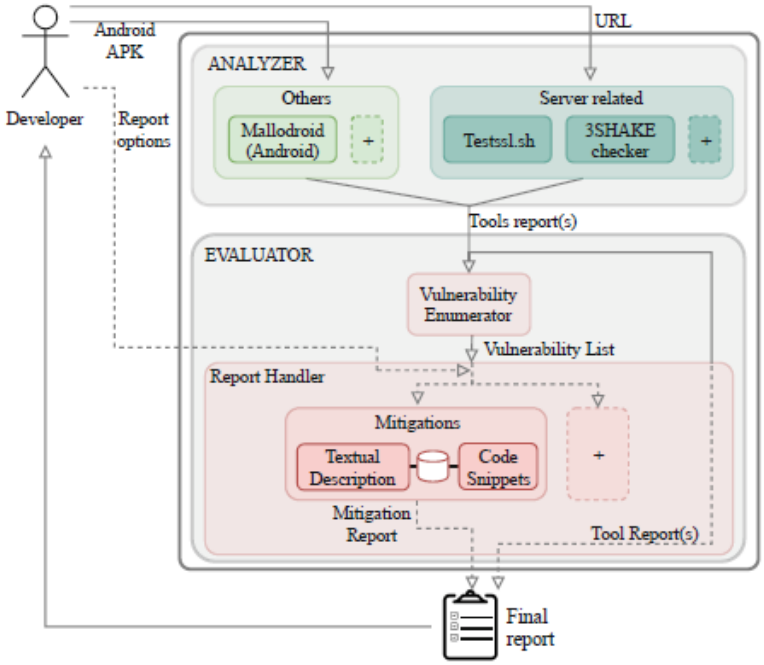


Figure 3.2. TLSAssistant Architecture (Source[7])

Checks	sslscan	sslenum	TLSSLed	TLS-atk	3Shake_chk	testssl
SSLv3, TLS 1.0, 1.1 and 1.2, RC4	●	●	●	●	○	●
AES ciphers	◗	◗	●	●	○	●
Weak ciphers	◗	◗	●	◗	○	●
SSLv2, Secure renegotiation	●	○	●	●	○	●
POODLE, CBC-mode cipher, 3DES	●	●	○	●	○	●
MD5/SHA1 signature alert	●	●	●	○	○	●
Sweet32	●	◗	○	◗	○	●
Certificate expiration	●	○	●	○	○	●
Weak DH parameters	●	●	○	○	○	●
Heartbleed, TLS compression	●	○	○	●	○	●
BEAST	◗	○	●	○	○	●
TLS 1.3, DROWN	○	○	○	●	○	●
Qualys scoring	○	●	○	○	○	●
More analysis[a]	○	○	○	○	○	●
3SHAKE	○	○	○	○	●	○

Figure 3.3. TLS tool comparison (Source[7])

from the public internet. It gives a lot of TLS information on the target and in the final report, there is not only the vulnerabilities found but also mitigation for these found vulnerabilities in order to help the system administrator. It is less simple to use than SSL Test but it can be used in more situations. It has also some disadvantages, like:

- It uses the testssl.sh that "takes a picture" of the target in that specific moment. If an attacker changes the target settings after the TLSAssistant has collected the results of the scan, the tool does not alert the system administrator about the newest vulnerabilities until the next scan.

- TLSAssistant tests only the expiration of the target's certificate. It does not verify if a certificate that is not expired yet is valid or not. An attacker can use a compromised certificate to exploit a MITM attack.

Chapter 4

Network Traffic analyzers

In this Chapter, the Network traffic analyzer will be described. At the beginning there is a brief introduction to distinguish all the different typologies of the Network Traffic Analyzer, then will be described the specific tools integrated into the Monitor for TLS Attacks.

4.1 Introduction

There are 4 main typologies of tools to analyze network traffic:

- **IDS (Intrusion Detection System)** - is a system that monitors network traffic for suspicious activity and it alerts when such activity is discovered.
- **IPS (Intrusion Prevention System)** - is a security tool or service that helps an organization identify malicious traffic and proactively blocks it from entering their network. The IPS can be deployed in-line to monitor incoming traffic and inspect that traffic for vulnerabilities and exploits. If the IPS detects a vulnerability, it can take the appropriate action written in the security policy, such as blocking access or quarantining hosts.
- **Packet Sniffer** - allows sniffing all the traffic on the network. This type of tool can save the traffic sniffed for analysis later. Wireshark is a famous packet sniffer tool.
- **Firewall** - is a software or hardware component that allows filtering the incoming and outcoming network traffic. For example, it can be used to deny all the packets coming from a specific IP address.

4.2 IDS/IPS tools

IDS can be classified into 5 types [46]:

1. **Host Intrusion Detection System (HIDS)** - run on independent hosts or devices on the network. A HIDS monitors the incoming and outgoing traffic from the device only and will alert the administrator if suspicious or malicious activity is detected. The suspicious traffic is found by comparing an old snapshot of the existing system files with the new snapshot taken.
2. **Network Intrusion Detection System (NIDS)** - is set up at a key point within the network to examine traffic from all devices on the network. It compares the passing traffic on the whole subnet with some rules written before that represent a collection of known attacks.

As described in " Internet Attacks and Intrusion Detection System: a review of the literature " from Raman et al. [47], the IDS has 3 detection methods:

1. **Signature-based** - detects the attacks based on a specific pattern of bytes in the packet. It also detects on the basis of the already known malicious instruction sequence that is used by the malware. These detected patterns in the IDS are known as signatures and they are used to write the rules for the IDS;
2. **Anomaly-based Method** - this method is used to easily detect unknown malware attacks. This method is used with machine learning to build a trust model and all the packets coming are compared with that model. If they are not found in the model are declared suspicious;
3. **Hybrid** - this method combines the 2 above IDS methods. In this way it is possible to have better results.

IPS detection methods are equal to IDS detection methods but IPS can be classified in 4 types:

1. **Network-based intrusion prevention system (NIPS)** - like IDS, it is placed at key network locations, where it monitors traffic and scans for cyber threats.
2. **Wireless intrusion prevention systems (WIPS)** - monitor Wi-Fi networks, acting as a gatekeeper and removing unauthorized devices
3. **Network behavior analysis (NBA)** - is focused on network traffic to detect flows that might be associated with distributed denial of service (DDoS) attacks.
4. **Host-based intrusion prevention system (HIPS)** - like IDS, it monitors inbound and outbound traffic from that device only

4.2.1 Snort

Snort is an open-source tool that can be utilized as a signature-based IDS or IPS [48]. It was created in 1998 by Martin Roesch. It uses a series of rules that define malicious traffic. As IPS, Snort can be deployed to stop these packets. It can be utilized as a packet sniffer, packet logger, or as a full-blown network IPS. Snort has a pre-defined set of rules against a well-known attack but allows the users to create their own rules. This tool is written in C and is similar to Suricata. In fact Suricata is considered the successor of Snort, but Snort is still widely used and in the years is changed a lot. At the start of 2021 Snort has officially released the new version of this tool: Snort 3.0. This version is different from the previous versions in terms of resources utilized and because the newer version supports multiple packet-processing threads, which frees up more memory for packet processing. The rules have the same keywords and syntax as Suricata's rules.

4.2.2 Suricata

Suricata is an open-source project developed by Victor Julien, Matt Jonkman e William Metcalf [49]. The first version of Suricata was released in 2009 and its copyright is owned by the Open Information Security Foundation (OISF), which is a nonprofit organization. Suricata can be used as IDS or IPS and like Snort it is written in C language and is signed-based. Differently from Snort Suricata offered multi-threaded support from the first released. The rules of Suricata are Snort compliant and this means that the rules are the same syntax and keywords of Snort. Suricata has its pre-defined rules but users can create their own rules. The Suricata's rules can be divided into 3 main parts as shown in figure 4.1:

1. **action** - This is the red part. This determines what happens when the rules mathes. The action in suricata are:
   - alert - generate an alert
   - pass - stop further inspection of the packet
   - drop - drop packet and generate alert
   - reject - send RST/ICMP unreach error to the sender of the matching packet.
   - rejectsrc - same as just reject
   - rejectdst - send RST/ICMP error packet to the receiver of the matching packet.
   - rejectboth - send RST/ICMP error packets to both sides of the conversation.
2. **header** - This is the green part. This part defines the protocol of the rule, the sender IP address, the sender port and the receiver IP address and the receiver port. In the header is also specified the direction of the rule. In figure 4.1 the direction indicates that the rule match only the packet of TLS protocol coming from *10.0.2.15:1518* to *10.0.2.7:443*. The direction can also be bi-directional with <>.
3. **options** - This is the blue part. This part defines the specifics of the rules.

```
alert tls 10.0.2.15 1518 -> 10.0.2.7 443 (msg:"|VULNERABILITY| #HEARTBEAT EXTENSION#
SSLv3";flow:from_server;content:"|16 03 00|";content:"|02|"; distance:2;within:4; content:"|00
0f|";content:"|01|"; distance:2;within:4; sid:10;)
```

Figure 4.1. An example of rule in Suricata/Snort

4.2.3 Zeek

Zeek is an Open Source Network Security Monitoring Tool that was developed in '90 by Vern Paxson with the name of *Bro* [50]. In 2018 the name was changed with *Zeek*. Zeek is a network security monitor (NSM) but can also be used as a network intrusion detection system (NIDS). When Zeek starts to sniff the network traffic it creates a directory called *current* where it creates a log file for each protocol used. When the user stops the tool, Zeek stores all the log files in a zip with the timestamp of the start date and time. All the zip files of a day are stored in a directory with the date of the day when the logs are created. Zeek uses a JSON format to store the log of the captured network traffic or a pre-defined table with all the information about the session of the protocol used. Zeek provides full customization of the output of the log file but does not provide add new customization signature-based rules. Zeek's rules are completely different from Snort/Suricata rules. These rules are written in C++like language and they are based on some pre-defined event that Zeek allows to use to customize the traffic interception. There are base concepts to clarify to write a script in Zeek:

1. **Stream** - Each stream correspond to a log file. For example, there is the SSL stream used to intercept SSL/TLS traffic. Each stream has some fields with a name and a type. These fields can be different for each stream file and these fields are the information written in the log file at the end. To create a stream it is necessary:
   - A *record* must be defined in which the user must declare all the fields that will be used.
   - A *log ID stream* must be defined to identify uniquely the Stream.
   - To write the fields into a log file the *Log::write* must be used.
   - To create a *log stream* the function *Log::create_stream* must be used.

2. **Filter** - For each stream exists a different set of filter to specify the information that must be stored. It is possible to add or remove a custom filter.

3. **Writer** - For each filter exists a writer that defines the output format for the information stored. The default writer is ASCII but it is possible to use other writers.

In figure 4.2 is shown the declared fields for the ssl stream and in figure 4.3 is shown the *ssl_server_hello* event that is called when in TLS protocol the SSL Server Hello message is intercepted.

```
export {
        redef enum Log::ID += { LOG };

        global log_policy: Log::PolicyHook;

        ## The record type which contains the fields of the SSL log.
        type Info: record {
                ## Time when the SSL connection was first detected.
                ts:               time            &log;
                ## Unique ID for the connection.
                uid:              string          &log;
                ## The connection's 4-tuple of endpoint addresses/ports.
                id:               conn_id         &log;
                ## Numeric SSL/TLS version that the server chose.
                version_num:      count           &optional;
                ## SSL/TLS version that the server chose.
                version:          string          &log &optional;
                ## SSL/TLS cipher suite that the server chose.
                cipher:           string          &log &optional;
                ## Elliptic curve the server chose when using ECDH/ECDHE.
                curve:            string          &log &optional;
                ## Value of the Server Name Indicator SSL/TLS extension.  It
                ## indicates the server name that the client was requesting.
                server_name:      string          &log &optional;
                ## Session ID offered by the client for session resumption.
                ## Not used for logging.
                session_id:       string          &optional;
                ## Flag to indicate if the session was resumed reusing
                ## the key material exchanged in an earlier connection.
                resumed:          bool            &log &default=F;
```

Figure 4.2. An example of log stream and its fields in TLS configuration file

```
event ssl_server_hello(c: connection, version: count, record_version: count, possible_ts: time, server_random:
string, session_id: string, cipher: count, comp_method: count) &priority=5
        {
        set_session(c);

        # If it is already filled, we saw a supported_versions extensions which overrides this.
        if ( ! c$ssl?$version_num )
                {
                c$ssl$version_num = version;
                c$ssl$version = version_strings[version];
                }
        c$ssl$cipher = cipher_desc[cipher];

        if( comp_method≠0){
                c$ssl$compression="COMPRESSION";
        }
```

Figure 4.3. Zeek pre-defined event

It is possible to create a file with the custom script into `/usr/local/zeek/share/site/` directory. After create the file, the script must be loaded into the *local.zeek* file in the same directory with the command:

```
@load namefile
```

4.2.4 Comparing Snort/Suricata/Zeek

From the above IDS, Snort and Suricata are quite similar but as described in "Performance Comparison and Detection Analysis in Snort and Suricata Environment" [52] and shown in Figure 4.5 and Figure 4.4 the 2 tools process differently the incoming packets. Figure 4.5 shows how many packets are intercepted by the 2 IDS using a single thread. As shown, Suricata succeeds to intercept more packets than Snort. The same succeeds with more than one threads as shown in Figure 4.4. In this situation Snort drop a lot of packet related to Suricata.

Figure 4.6 and Figure 4.7 show the CPU utilization for both the IDS. From a point of view of simplicity, both the IDS are easy to use and install. They use the same rules and both allow to use of LUA scripting for more complex rules.

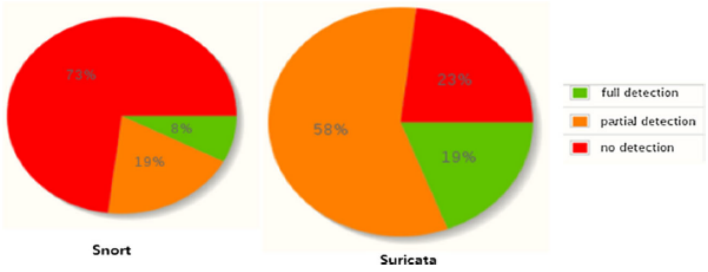


Figure 4.4. Packet intercepted in Suricata and Snort with one thread (Source[52])

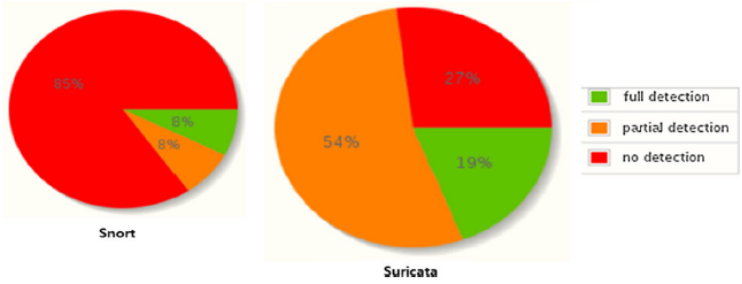


Figure 4.5. Packet intercepted in Suricata and Snort with more threads (Source[52])

Zeek is different from the above-cited IDS. It monitors the traffic flows and produces log file with all the information about network traffic. This tool is useful to understand network behavior. Zeek allows the user to better interpret this data with a C++like language. This allows the user to create a statistical and custom interpretation of the captured data. This tool is more difficult to use and its utilization is different from the above-cited IDS. From the start of March 2022, an article called "Effectiveness Evaluation of Different IDSs Using Integrated Fuzzy MCDM Model"[53] was released. This article evaluates the impact of five popular IDSs on their efficiency and effectiveness in information security. The authors of this research have used fuzzy Analytical Hierarchy Process and fuzzy technique for order performance by similarity to ideal solution (TOPSIS)-based integrated multi-criteria decision-making (MCDM) methodology. The 5 IDS analyzed in this research are:

1. **Zeek (S1)**
2. **Suricata (S2)**
3. **Security Onion (S3)**
4. **OSSEC (S4)**

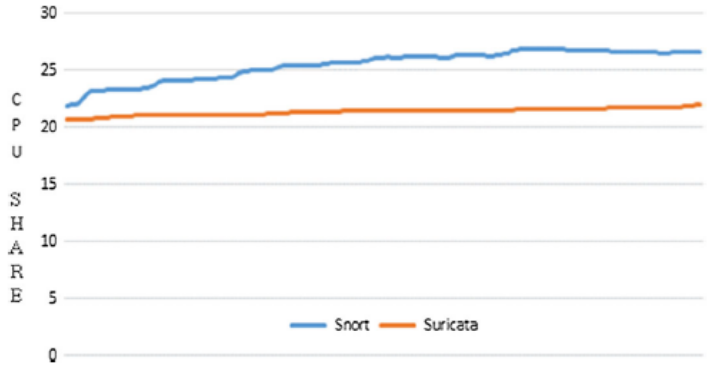


Figure 4.6. The figure shows the percentage of CPU comparing Snort and Suricata using single thread (Source[52])

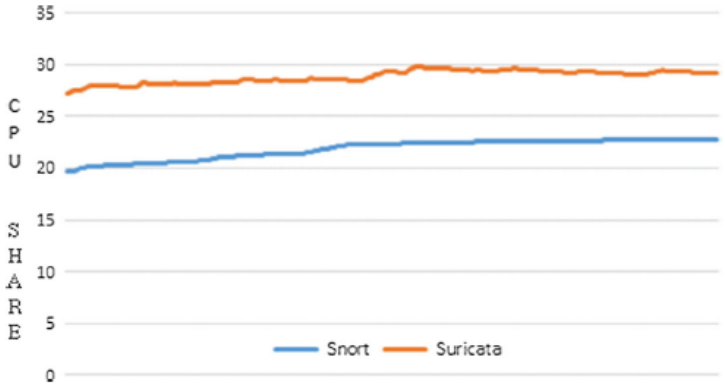


Figure 4.7. The figure shows the percentage of CPU comparing Snort and Suricata using multi-threads (Source[52])

5. **Snort (S5)**

The researchers first used the MCDM, that is is a discipline of combinatorial optimization in which the alternatives are evaluated, to identify the best alternative. This method utilizes a set of strategies and techniques to integrate various parameters into a decision-making process. Based on literature and with the help of experts the researchers have found 4 macro-areas that then are split into 13 sub-areas to test. After having found these 13 sub-areas the researchers used a fuzzy AHP-TOPSIS methodology which is divided into:

1. **Fuzzy AHP** - a popular methodology used to resolve difficult selection challenges. This methodology differiazites the problem into a tree form to describe better the problem. Then a Triangular Fuzzy Number (TFN) is created from the hierarchical structure. Then the researchers transformed the linguistic number into crisp numbers with advanced processes.
2. **Fuzzy TOPSIS** - the TOPSIS methodology face-up multi-standard decision-making problems as a problem with m options as a geometrical configuration with m points inside a n-dimensional problem area.

For this research, 30 cybersecurity specialists with more than 12 years of experience and 47 researchers with 10 years of IDS research experience were consulted for each parameter. The results of this research are shown in figure 4.8. On the left, there is the fuzzy result of the fuzzy AHP-TOPSIS methodology that represents the satisfaction degree. The researchers have then defuzzed the result to compare it with the traditional AHP-TOPSIS methodology that is possible to see on the right.

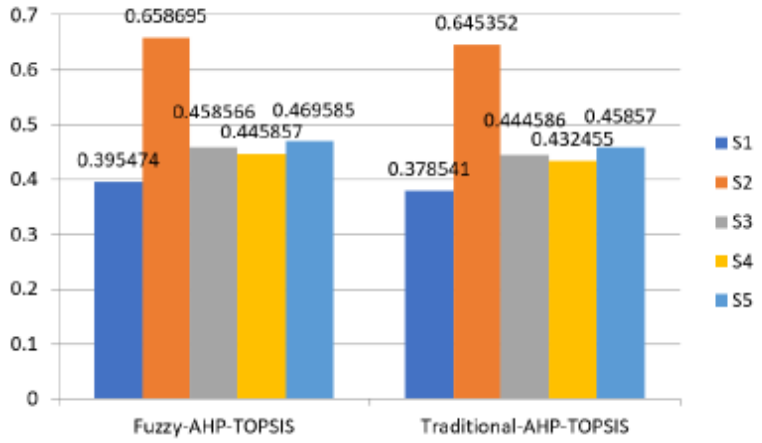


Figure 4.8. Comparison of the fuzzy results with traditional approach (Source [53])

To finish the researchers have done a sensitivity analysis where for each of the 13 variables picked they have altered a variable each time to test the validity of the gathered data. In figure 4.9 is possible to see the results of the 13 experiments done.

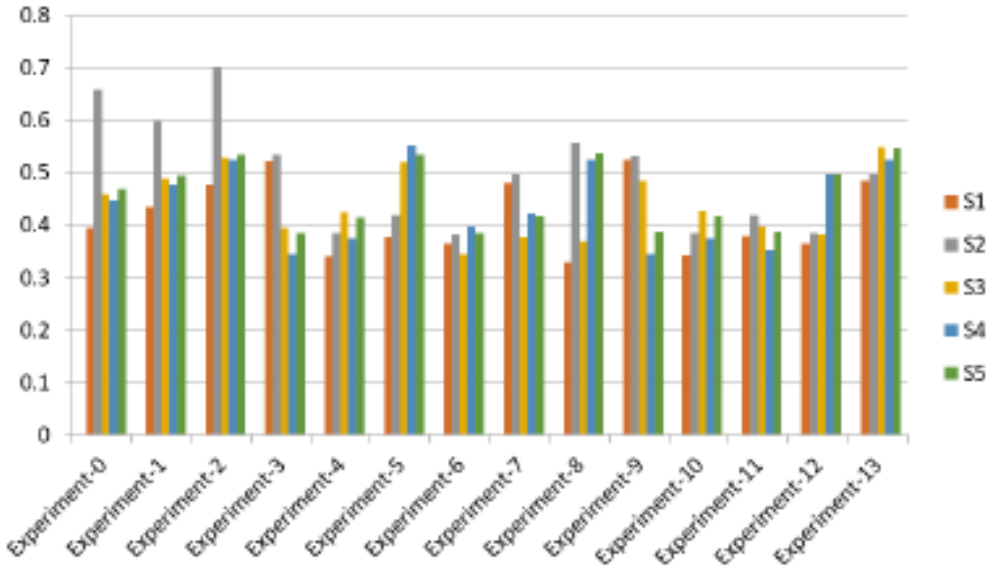


Figure 4.9. Experiment Result (Source [53])

Is possible to see that Suricata (S2) is the most satisfiable IDS in figure 4.8 and in more experiments than the others in figure 4.9. In Table 4.1 are shown some information about the above-cited IDS.

Table 4.1. Compare different IDS

IDS name	Provider	OS	Type	License	Network Traffic
Snort	Cisco System	Windows/Unix/Mac	NIDS	GNU General Public License	IPv4/IPv6
Suricata	OISF	Windows/Unix/Mac	NIDS	GNU General Public License	IPv4/IPv6
Zeek	Vern Paxton	Windows/Unix/Mac	NIDS	BSD License	IPv4/IPv6

4.3 Suricata Configuration

4.3.1 Requirements

To install Suricata the following libraries are required[54]:

- libjansson
- libpcap
- libpcre
- libmagic
- zlib
- libyaml

and the following tools are required:

- make
- gcc (or clang)
- pkg-config

To check the dependencies are present and to install them if they are not present the user can run these commands depending on the OS:

- **Ubuntu/Debian**

```
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**
  To install Dependencies in Windows the user can download the installer on Suricata download page [55].

4.3.2 Installation

To install Suricata there are many ways, like cloning the GitHub repository, adding the PPA binary repository in Ubuntu, or through the installer file in Windows, but the easiest way is to download the *.tar.gz* file on Suricata download page[55] and then extract and install it through the following commands:

```
tar xzvf suricata-6.0.6.tar.gz
cd suricata-6.0.6
./configure
make
make install
```

These commands are for the Suricata Stable Version 6.0.6 which is the newest stable version during the writing of this thesis but changing the version on these commands, they will continue to work.

The tool will be installed on */usr/local/bin/* and the default configuration will be in */usr/local/etc/suricata/*. If the user does not specify a different path with *–prefix* when he runs *./configure* command. If Suricata is installed with super user privileges with *sudo* command the location will be in */etc/suricata/*. In the location that the user have chosen there will be the configuration file of suricata *suricata.yaml* and a directory *rules/*, where will be set the Suricata'rules for the threat detection.

4.3.3 The configuration of YAML

The suricata's configuration file is a yaml file in version 1.1, where the user or the system network administrator can set custom settings. It is a very long file but can be divided into 5 parts:

1. **Network Settings** - In this part, the user can set his home network through the variable *HOME_NET* and the ports of the protocol used through some variable, like *HTTP_PORTS, TLS_PORTS,* etc.

2. **Output Settings** - In this section, the output's path can set on the variable *default-log-dir*. The default path if the user does not set differently is */var/log/suricata/*. It is also possible to set the name and other options like the format and how to create it (append or write only) for the log file generated by sniffing the network traffic. It is possible to generate a file for each protocol seen on the network as shown in figure 4.10 for the TLS protocol. In this section is possible to have different types of files for different utilization, for example, Suricata allows having fast alerts where it writes alerts that matched the rules as soon as possible, and the output file can have a JSON format with custom information for the network traffic, or can generate a pcap file that can be read from Wireshark to understand better the incoming and outcoming traffic. Suricata also allows a parameter called *community_id* that is used to give a predictable flow ID to records that can be used to match records to the output of other tools such as Zeek (Bro).

3. **Common Capture Settings** - There are some options to speed up Suricata, like the number of threads that will be used.

4. **App Layer Protocol Settings** - In this part, it is possible to configure the app-layer parser. For each protocol, is possible to set some options, for example for the TLS option that is possible to set the ports of that protocol and how to react when encrypted communication starts.

5. **Advanced Settings** - This is the section where the user can completely customize the tool for his needs. It is possible to customize Suricata to run it in a different group or with a different user, to customize the flow timeouts, or customize many other settings. Some important settings in this section are the path for the directory of the rules in the variable *default-rule-path* and the files' rules name to use in variable *rule-files*. It is possible to specify more than one rules file.

```
- tls-log:
    enabled: yes  # Log TLS connections.
    filename: tls.log # File to store TLS logs.
    append: no
    extended: yes     # Log extended information like fingerprint
    custom: yes       # enabled the custom logging format (defined by customformat)
    customformat: "%{%D-%H:%M:%S}t.%z | %a:%p → %A:%P | %v %n %d %D"
    filetype: regular # 'regular', 'unix_stream' or 'unix_dgram'
    # output TLS transaction where the session is resumed using a
    # session id
    session-resumption: no
```

Figure 4.10. Options to generate a file for TLS traffic

4.4 Zeek Configuration

4.4.1 Requirements

The following libraries are required to install Zeek:

- BIND8 library or greater (if not covered by system's libresolv)
- Bison 3.3 or greater
- C/C++ compiler with C++17 support (GCC 8+ or Clang 9+)
- CMake 3.15 or greater
- Flex (lexical analyzer generator) 2.6 or greater
- Libpcap
- Make
- OpenSSL
- Python 3.5 or greater
- SWIG
- Zlib

To verify and eventually to install these dependencies the user can use these commands depending on the OS used:

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

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

- **DEB/Debian-based Linux:**

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

- **FreeBSD:**

```
sudo pkg install -y bash git cmake swig bison python3 base64
pyver=python3 -c 'import sys;
    print(f"pysys.version_info[0]sys.version_info[1]")'
sudo pkg install -y \$pyver-sqlite3
```

- **macOS**

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

This command installs X-Code that is required for macOS OS. Then will be possible to install CMake, SWIG, Bison, and OpenSSL through Homebrew or MacPorts.

4.4.2 Installation

It is possible to install Zeek from binary packages or to download the source code of the version preferred from the Download page of Zeek[56]. If the source code is downloaded, it is possible to find a *.tar.gz* file in the Download directory. As Suricata the user can extract this zip file with the command:

```
tar -xvzf zeek-5.0.2.tar.gz
```

Once the file is extracted, the user must go into the directory extracted and run the following commands:

```
./configure
make
make install
```

As Suricata, Zeek has a default path that is */usr/local/zeek/*. A user can change the default path by using the option *–prefix* followed by the custom path with the *./configure* command. It is possible to use Zeek with Docker if the user knows Cloud Computing. There is the Zeek docker image on Docker Hub.

4.4.3 The configuration of high-level rule in Zeek

There are more configuration files for Zeek. On `/usr/local/zeek/etc/` directory is possible find the network configuration file *networks.cfg* and the Zeek configuration file *zeekctl.cfg*. The first is shown in figure 4.11 and it is used to set the private IP addresses. The second is used to set the Zeek settings as the path for a specific log file or to set an email to inform the user when some anomalies are found.

```
# List of local networks in CIDR notation, optionally followed by a
# descriptive tag.
# For example, "10.0.0.0/8" or "fe80::/64" are valid prefixes.

10.0.0.0/8          Private IP space
172.16.0.0/12       Private IP space
192.168.0.0/16      Private IP space
```

Figure 4.11. Zeek network configuration file

Chapter 5

TLS Attack tools

In this Chpater, the tools integrated into the Monitor for TLS attacks are described. These tools are used in the Monitor for TLS attacks tool to verify the possible vulnerability found by the IDS. Once the tool has finished, it writes the tool's result into a log file.

5.1 Nmap

Nmap [57] is a famous free and open-source tool used for network discovery and security auditing. This tool is used a lot by Ethical and non-Ethical hackers to know how many hosts there are on the network and what kind of services run behind the hosts. With this tool, the attacker can know if there is some open port on a host and start to study the victim. So this tool is principally used in the initial phase of an attack to gather information on the victim like the OS version running on a host, but it can be used also in other ways. As it is used in this work, Nmap can be used also in the attack phase. The Nmap's developers have a library with a lot of attack scripts that the user can launch. This tool allows the user also to write and launch his own custom script. In this thesis, Nmap is used for its attack script against the TLS protocol. With the following command, Nmap executes a script to verify the POODLE attack against a host:

```
nmap -A -p [port] --script=ssl-poodle --script-args=vulns.showall [ipAddress]
```

In the Nmap command, there are the following options:

1. -A - this options enables the OS detection for a better result.
2. -p - this option represents the port of the tested host.

ipAddress - this parameter is mandatory and represents the ip address of the host tested. This parameter must be inserted always at the end of the command.

3. –script-args - with this option, it is possible to pass some arguments to the script launched. If the argument is **vulns.showall** Nmap reports also the result **NOT VULNERABLE**. Without this option, Nmap reports only **VULNERABLE, LIKELY VULNERABLE, VULNERABLE (DoS) and VULNERABLE (Exploitable)**.
4. –script - with this option it is possible to launch a specific script in the Nmap library. With this option, other script are launched into the Monitor for TLS attacks tools, such as ROCA, Heartbleed, LogJam, and CCS Injection.

5.2 Measploit Framework

Metasploit Framework[58] is one of the most used tools for penetration testing. It has a huge library with many exploit written in a different language. It is an open-source modular program based on Ruby. In most cases, Metasploit is used in combination with Nmap. First, a penetration tester uses Nmap to gather information on a specific host, and then he uses Metasploit to test the vulnerabilities of network security. There are different modules that can be used with Metasploit:

- Exploit - This module is used for taking advantage of the weak points of a target. The exploit module performs a series of specific commands against specific vulnerabilities found in a system. An example of an exploit can be buffer overflow or code injection.
- Post-Exploitation Code - This module helps in deeper penetration tests to collect more information regarding the target. Hash dumps is an example of this kind of module.
- Auxiliary Function - This more than module is a set of supplementary tools and commands which do not require payloads to succeed. An example of these supplementary tools are sniffers, SQL injection tools, or scanners.
- Payloads - This module has a set of malicious codes which can be used after the exploitation phase. There are different features available in this module, like small codes or even a small application. The Payloads module can be used to open a command shell or meterpreter, a payload that allows writing DLL files, to the target system.
- Listeners - This module is called this way because collects malicious software called Listener that hides itself on the target machine to get access for the attacker.
- Encoders - This module collects a set of tools that are used for converting information or codes.
- NOPs - This module collects some kind of operation called NOP which stands for No OPeration. These operations prevent payloads from crashing. In some cases, the NOP module generates some arbitrary bytes to bypass standard IPS/IDS NOP signatures.

It is possible to use the Metasploit framework also in a single command line, in the following way:

```
msfconsole -x use auxiliary/scanner/ssl/openssl_heartbleed;set RHOST
    [ipAddress];set RPORT [port];set TLS_VERSION [tls_version];check;exit
```

The option **-x** allows using all the options that are required for the attack script in a single command line. All the single options required are separated by **;**. The above command executes the Heartbleed script with Metasploit Framework. For this script is possible also to retrieve the private key of the target if it is vulnerable to the Heartbleed attack. This is possible with the following command:

```
msfconsole -x use auxiliary/scanner/ssl/openssl_heartbleed;set RHOST
    [ipAddress];set RPORT [port];set TLS_VERSION [tls_version];keys;exit
```

The difference from the other command is the **keys** action. Almost all commands have the **check** action to verify if the target is vulnerable, but only few script have also the **exploit** or **keys** options. This attack tool is integrated into the Monitor for TLS attacks tool for other attacks such as CCS Injection and ROBOT.

5.3 Ettercap

Ettercap [59] is a famous tool used to implement MITM attacks. This tool can sniff lives connection, and filter the sniffed connection on the fly. It can sniff in four modes: IP Based, MAC Based, ARP Based (full-duplex) and PublicARP Based (half-duplex). With Ettercap there are available different method for a MITM attack, to change method is necessary to run ettercap with the following command:

```
sudo ettercap -T -M <option> /IP_CLIENT// /IP_SERVER//
```

This tool requires superuser privileges to run because it requires the utilization of the Network Card. The options methods that can be used with Ettercap are:

- **arp** - This option is used to implement arp poisoning MITM attack. For the ARP Poisoning Attacks the user sends ARP requests/replies to the victims to poison their ARP cache. Once the cache is poisoned the victim will send packets to the attacker, which can modify and then forward them to the real destination. This option can be used with 2 additional parameters: remote and oneway. The remote parameter is used if the attacker wants to implement an arp poisoning attack against a default gateway. Oneway is used if the attacker wants to implement the arp poisoning attack against only one direction, so if he wants to poison only the cache of the sender or receiver. If these parameters are omitted the arp poison attack is implemented in the 2 directions but only for targets in the same network.
- **port** - This option is used for Port Stealing. This technique is useful to sniff traffic when in a network is used a switch. This methodology can be used when static arp are used and the arp poisoning attack can not be implemented.
- **dhcp** - This option is used for DHCP spoofing. With this kind of MITM attack, the attacker impersonates a DHCP server and can hijack all the outgoing traffic generated by the client.
- **icmp** - This option is used to implement ICMP redirection. With ICMP redirection MITM attack, the attacker impersonates a better route for the internet for the victim. In this way, the victim will not send packets to the default gateway but to the attacker. The attacker can redirect the messages to a different target or modify and then forward them to the default gateway.
- **ndp** - This option can be used only if IPV6 support is eabled. It implements the NDP poisoning attacks that is used for MITM attack against the IPV6 connection. This option works like the arp option but only for IPV6 connection.

Ettercap offers also a GUI interface with the following command:

```
ettercap -G
```

5.4 TLS-Attacker

TLS-Attacker[60] is a java-based framework for analyzing TLS libraries. It is developed by the Ruhr University Bochum, the Paderborn University, and the Hackmanit GmbH. Has shown in figure 5.1, this tool has many modules. In this work, the Attacks module has been used for the Bleichenbachers and POODLE attacks.

The newer version of TLS-Attacker does not use more this module, because the newest version has split the Attack module from the rest. This module has been used in a different tool. For this work, the TLS-Attakcer version 3.7.2 is used. This tool depends on Maven and Java. Is possible to install Maven on ubuntu with the following command:

```
sudo apt-get install maven
```

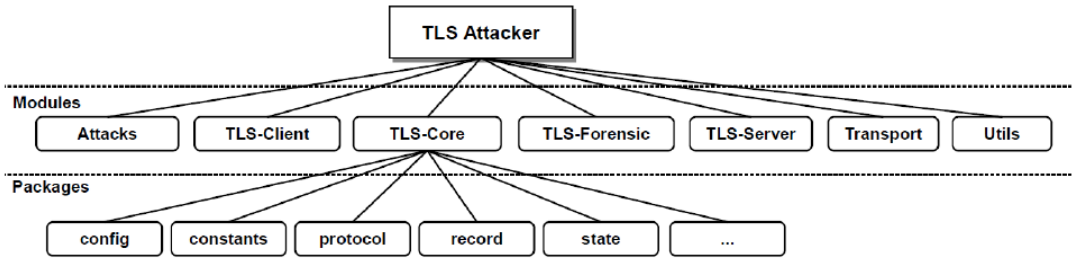


Figure 5.1. TLS-Attacker modules (Source[60])

This version can be downloaded from GitHub with the following command:

```
git clone https://github.com/tls-attacker/TLS-Attacker.git
cd TLS-Attacker
mvn clean install
```

In order to use the Attack module the user has to run the following command:

```
cd apps
java -jar Attacks.jar [Attack] -connect [host:port]
```

where Attack can be:

- POODLE
- Bleichenbacher

5.5 TestSSL

Testssl[61] is a free command line tool that checks on any port of the server to find a service that supports TLS/SSL ciphers and protocols. This tool can be used to have a general test of the server or can implement a specific test on the server. The figure 5.2 shows a general test on a server with the following command:

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

As shown from the figure, this tool uses different colors to better explain if a configuration adopted is good (green), medium(yellow), or bad (red). This tool can be used also for a specific attack. This is useful because the full version takes a lot of time. It is possible to use this tool for a specific attack with the following comand:

```
testssl [host]:[port] --[attack_name}
```

To have more informations for this tool the following command can be used:

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

```
## Scan started as: "testssl -U --full --severity LOW --json --parallel -n none --logfile testssl_10.0.2.7:443.log https://10.0.2.7:443"
## at kali:/usr/bin/openssl
## version testssl: 3.0.7  from
## version openssl: "1.1.1o" from "May  6 20:20:36 2022")

 Start 2022-05-24 06:42:33        -->> 10.0.2.7:443 (10.0.2.7) <<--

 rDNS (10.0.2.7):        (instructed to minimize DNS queries)
 Service detected:       HTTP


 Testing protocols via sockets except NPN+ALPN

 SSLv2      supported but couldn't detect a cipher and vulnerable to CVE-2015-3197
 SSLv3      offered (NOT ok)
 TLS 1      offered (deprecated)
 TLS 1.1    offered (deprecated)
 TLS 1.2    offered (OK)
 TLS 1.3    not offered and downgraded to a weaker protocol
 NPN/SPDY   not offered
 ALPN/HTTP2 not offered

 Testing for server implementation bugs

 No bugs found.

 Testing cipher categories

 NULL ciphers (no encryption)                not offered (OK)
 Anonymous NULL Ciphers (no authentication)  not offered (OK)
 Export ciphers (w/o ADH+NULL)               not offered (OK)
 LOW: 64 Bit + DES, RC[2,4] (w/o export)     offered (NOT ok)
 Triple DES Ciphers / IDEA                   offered
 Obsolete CBC ciphers (AES, ARIA etc.)       offered
 Strong encryption (AEAD ciphers)            offered (OK)


 Testing robust (perfect) forward secrecy, (P)FS -- omitting Null Authentication/Encryption, 3DES, RC4

 PFS is offered (OK)          ECDHE-RSA-AES256-GCM-SHA384 ECDHE-RSA-AES256-SHA384 ECDHE-RSA-AES256-SHA
                              DHE-RSA-AES256-GCM-SHA384 DHE-RSA-AES256-SHA256 DHE-RSA-AES256-SHA DHE-RSA-CAMELLIA256-SHA
                              ECDHE-RSA-AES128-GCM-SHA256 ECDHE-RSA-AES128-SHA256 ECDHE-RSA-AES128-SHA
                              DHE-RSA-AES128-GCM-SHA256 DHE-RSA-AES128-SHA256 DHE-RSA-AES128-SHA DHE-RSA-SEED-SHA
                              DHE-RSA-CAMELLIA128-SHA
 Elliptic curves offered:     prime256v1
 DH group offered:            RFC2409/Oakley Group 2 (1024 bits)

 Testing server preferences

 Has server cipher order?     no (NOT ok)
 Negotiated protocol          TLSv1.2
 Negotiated cipher            DHE-RSA-AES256-SHA256, 1024 bit DH -- inconclusive test, matching cipher in list missing, better see below
 Negotiated cipher per proto  (matching cipher in list missing)
```

Figure 5.2. An example of a general test with testssl

Chapter 6

Monitor for TLS attacks tool

In this Chapter, there is the presentation of my proposal to oppose the TLS attacks. At the beginning, there is a description of the architecture and the tool's structure. Then, the Python code's tool is described in Section 6.2.1.

6.1 Components of Monitor for TLS attacks tool

This tool can be divided into 4 components:

1. **IDS** - This component is represented by the IDS used by the tool to intercept the traffic from the Network Card. In this tool version, there are 2 IDS: Zeek and Suricata. Snort can be added easily because it uses the same rule of Suricata.
2. **IDS Log file** - This file is different for each IDS used. The tool uses the ssl.log file generated by Zeek and a mio_fast.log for Suricata. The Suricata's log file depends on the Suricata settings, instead, Zeek produces always ssl.log.
3. **Core** - This component is represented by 3 Python files:
   - **Monitor_for_TLS_Attacks.py** - This is the main script that reads the log file generated by the IDS and through a string's manipulation starts the attacks.
   - **certificate.py** - This file is used by Monitor_for_TLS_Attacks.py to verify the server certificate. In this file can be found all the functions used to verify the CRL, OCSP, and the SCT of the certificate.
   - **ciphers.py** - This file contains 4 arrays each one containing a specific kind of TLS cipher. The four arrays are: **export_ciphers**, **des_ciphers**, **cbc_ciphers** and **rsa_ciphers**.
4. **Attack Module** - This module represents all the TLS attack tools found that are used to verify the vulnerabilities. This component launches the TLS attacks against the target and writes the attack's result in a log file. This log file will be present in a dedicated directory that will have the name of the ip address tested.

It is possible to pass a JSON configuration file to the tool as an argument to give a white list of ciphers, TLS versions, or server certificates. All the arguments present in this white list will not be verified. The tool architecture is shown in figure 6.1.

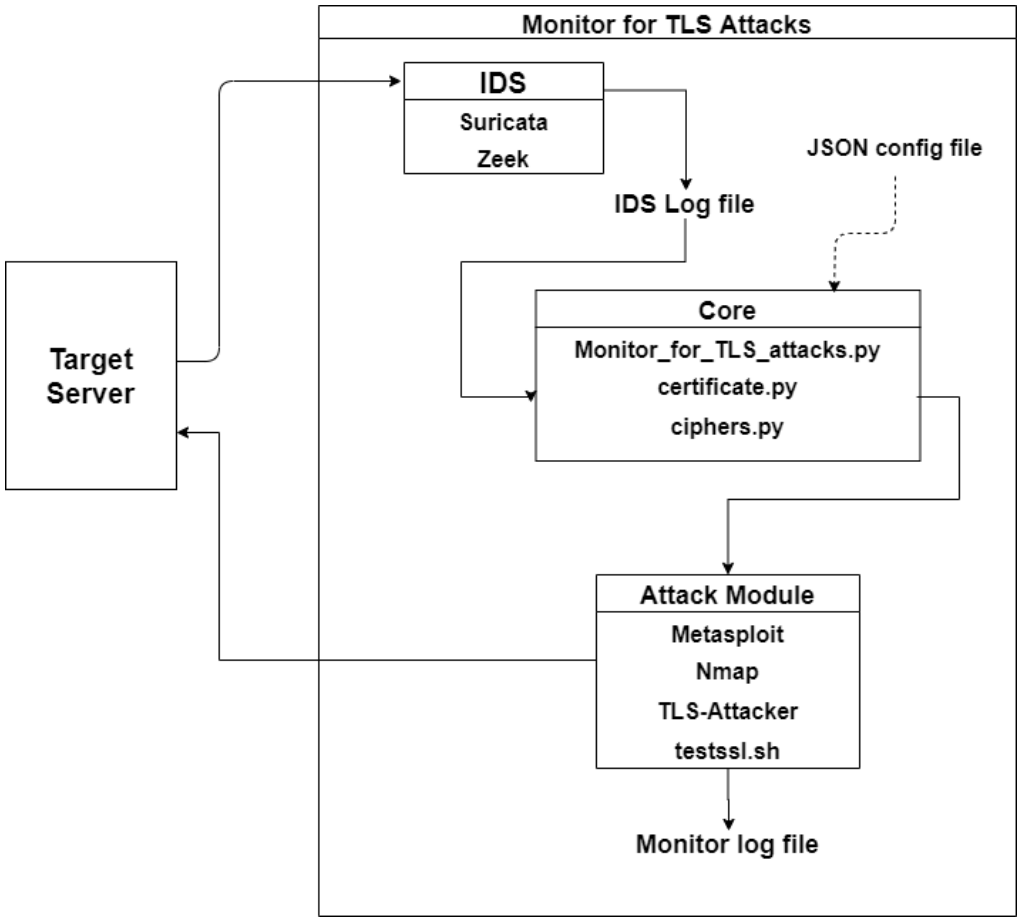


Figure 6.1. Monitor for TLS attacks tool Architecture

6.1.1 Multi-theaded tool

This tool uses a Producer-Consumer multi-threaded pattern. In this pattern, there is the need for a shared dictionary, where the producer can put the produced object and where the consumer can get the produced object. A dictionary in python is a collection that is ordered and changeable and it can not have duplicate members. In the Monitor_for_TLS_Attacks.py the producer thread is represented by a Watchdog object that monitors an object called **Handler**. When the **Handler** object is created, the logfile variable inside it is initialized with the path of the IDS log file used. When the IDS intercepts the network traffic and it modifies the log file, the Watchdog starts a function to read the new line in the log file. To read the log file the Watchdog calls 2 different functions:

1. file_reader_suricata
2. file_reader_zeek

The called function is different based on the IDS used because the 2 IDS use different kinds of log file. The Zeek log file is a table where for each field the IDS writes the corresponding value. The Suricata log file is written by a custom message for the **msg** keyword used in the Suricata rules. The difference for the 2 functions is only on how they extract the data from the logfile, but the goal for both functions is the same: produce in the dictionary an object.
In the python file the dictionary is called **vuln_conn** and it has kay-value structure, where:

- the key is the ip address sniffed
- the value is an array with all the vulnerabilities found for that connection

If **vuln_conn** reaches a max value stored in the **MAX_NUM** variable, the producer thread waits until **vuln_conn** has less value. The Consumer thread is represented in the python file by a function called **verify_vulnerability**. This thread waits until an item is inserted into **vuln_conn**. When this happens, the thread wakes up and pop an item from **vuln_conn**. Depending on the value of the item, the thread starts the corresponding attack tool against the ip address represented by the key of the item.

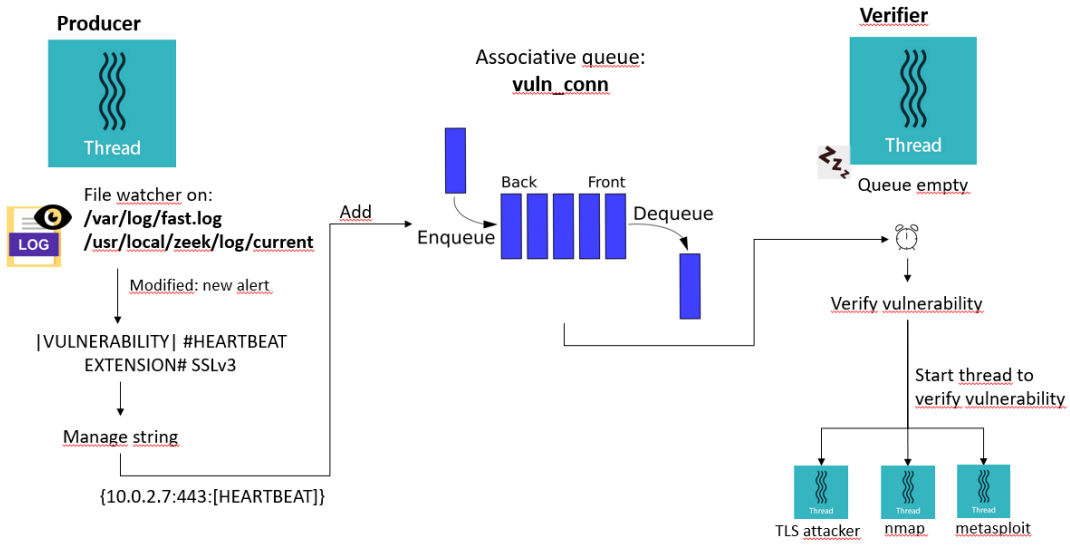


Figure 6.2. Monitor for TLS attacks Workflow

The Consumer thread put the item in the vulnerabilities' array in 2 different ways, depending on the IDS used by the tool. This different behavior is done by the different formats of the IDS log file.

1. **Suricata** - To obtain the main information from this IDS, there are some special characters in the message of the IDS's rules. The Consumer thread manipulates the strings of the message read in the Suricata's log file with the following special characters:
   - + - used to know the TLS version of the sniffed packet;
   - # - used to know the possible vulnerability sniffed;
   - % - if the possible vulnerability sniffed is based on a cipher suite used in TLS, this character is used to know what cipher suite is used in the sniffed packet;
   - $ - used in case of Heartbleed vulnerability, to know the packet's version and use this information for the Metasploit's Heartbleed attack.
2. **Zeek** - Different from Suricata, Zeek is not a rule-based IDS as mentioned earlier. In fact, Zeek with its function logs the main information in its log file in a table. The consumer thread reads Zeek's log file and manipulates the lines read thanks to the tabulation special character. Once split the line, saves the main information in the specific variables.

Independently by the IDS used, the tool verifies the server's certificate exchanged. The Monitor for TLS attacks verifies the validity of the certificate through OCSP and CRL. It verifies also the certificate's trustworthiness by the Certification Transparency. The consumer thread put into the item's array the value **|CERTIFICATE|**. The consumer thread takes this value in the array and starts the verification for the server's certificate.

6.2 How intercept TLS packets

As described in section 1.1 and shown in figure 1.2 the TLS protocol can be divided into phases and each phase has its own messages. This TLS message has a fixed structure that can be specified with a hexadecimal value. This fixed structure starts with the Record Layer, which wraps the other TLS parts. The structure of the TLS Record layer is very simple because it has 3 bytes to describe the handshake type and its version, 2 bytes to specify the length of the Record Layer, and the wrapped part, which has its own fixed structure. It is possible to see with Wireshark that the TLS protocol version is specified by the following values:

- SSLv2 - 02 00 [8]
- SSLv3 - 03 00 [9]
- TLSv1.0 - 03 01 [10]
- TLSv1.1 - 03 02 [11]
- TLSv1.2 - 03 03 [12]
- TLSv1.3 - 03 04 [13]

These values, in the case of the Handshake phase, are preceded by a byte that specify the phase (22 that in hexadecimal is 16). In case there is a handshake in the network with the TLS1.0 protocol it is possible to see with Wireshark that the TLS part of the packet starts with —16 03 01—. After these values, there is the byte that describes the Handshake types. Is possible to see the value with the described types in table 1.1. In the TLS packet there are some other parts with fixed lengths to describe:

- Compression method
- Cipher Suite
- Session ID
- Extensions

Depending on the TLS version, one part can have more or fewer bytes used to describe the protocol. TLSv1.0, TLSv1.1 and TLSv1.2 have similar structure, instead the other protocol are different. For example, the part where is written the cipher suite used is far 71 bytes in SSLv3 and SSLv2 versions but is far 41 bytes in TLSv1.0, TLSv1.1 and TLSv1.2 versions. Knowing the fixed structure and the difference in the various versions there is possible to write some Suricata rules to intercept specific packets in the network. For Zeek instead, there is the need to modify a bit the code for the TLS parts. To better modify the code and to better write the rules in Suricata, there is the need to know the TLS attacks very well to keep some marker in the TLS packet intercepted and give an alert. The Monitor for TLS Attacks tool is designed to be a preventing tool for TLS attacks, so for various attack, the marker isn't a specific marker, but the presence of this marker can mean that there is a probability to find a vulnerability. For example, the marker for the Heartbleed attack is the heartbeat message that can hide the vulnerability, but there isn't the heartbeat message sent for the memory leakage representing the attack. In the following table, there are reported the Marker for each attack that the tool tests and which attack tools is started when the marker is found.

These markers are been intercepted in different ways in the 2 IDS. For Suricata a file called tesi.rules is written and then enabled in the Suricata's yaml file. For Zeek the file `/usr/local/zeek/share/zeek/base/protocols/ssl` is modified to add some messages in the log file. In the following there will be explained how the marker is found by the different IDS.

- Heartbeat Extension - This kind of vulnerability is found by Suricata by the following rule:

Table 6.1. Attacks integrated in Monitor for TLS Attacks

Attack	Marker	Attack Tools
Heartbleed	Heartbeat Extension	Metasploit, Nmap, TestSSL, TLS-Attacker
CRIME	Compression enabled	TestSSL
DROWN	SSLv2 enabled	TestSSL, TLS-Attacker
Bleichenbachers/ROBOT	RSA Ciphersuites	TestSSL, TLS-Attacker, Metasploit
Sweet32	DES/3DES Ciphersuite	TestSSL
LogJam	Export Ciphersuite	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	TLSv1, TLSv1.1, TLS1.2	Nmap
Change Cipher Spec injection	SSLv3, TLSv1, TLSv1.1, TLSv1.2	Metasploit, Nmap
Roca Vulnerability	RSA CIphersuites	Nmap

```
alert tls any any <> any any (msg:"|VULNERABILITY| #HEARTBEAT EXTENSION#
    $1.0$";flow:from_server;content:"|16 03 01|";content:"|02|";
    distance:2;within:3; content:"|00 0f|";content:"|01|";
    distance:2;within:4; sid:11;)
```

This specific rule is written to intercept the Heartbeat extension for TLSv1.0 but by changing the content with the appropriate hexadecimal value is still possible to intercept the other TLS versions. This rule can be divided into 2 different Suricata rules joined with an AND logic operation. The first rule indicates the Server Hello message in the TLS handshake and it is used also for the others rules. This rule verifies the bytes indicating the Server Hello message (02) that must start after 2 bytes (distance rule) and end within 3 bytes (within rule). The first rule is shown in the figure 1.1 where the yellow indicates the Server Hello byte, the red arrow indicates the distance keyword and the orange arrow indicates the within keyword. The first rule is represented by:

```
alert tls any any <> any any (msg:"|VULNERABILITY| #HEARTBEAT EXTENSION#
    $1.0$";flow:from_server;content:"|16 03 01|";content:"|02|";
    distance:2;within:3;...)
```

The other rule uses the same Suricata keywords to check if the heartbeat extension in the Server Hello message is enabled. The Heartbeat extension is represented in the TLS message structure by 2 bytes **00 0f**. After these 2 bytes, there are 2 bytes representing the length of the extension and then the byte which represents the extension enabled **01**. The second rule is represent by:

```
(... content:"|00 0f|";content:"|01|"; distance:2;within:4; sid:11;)
```

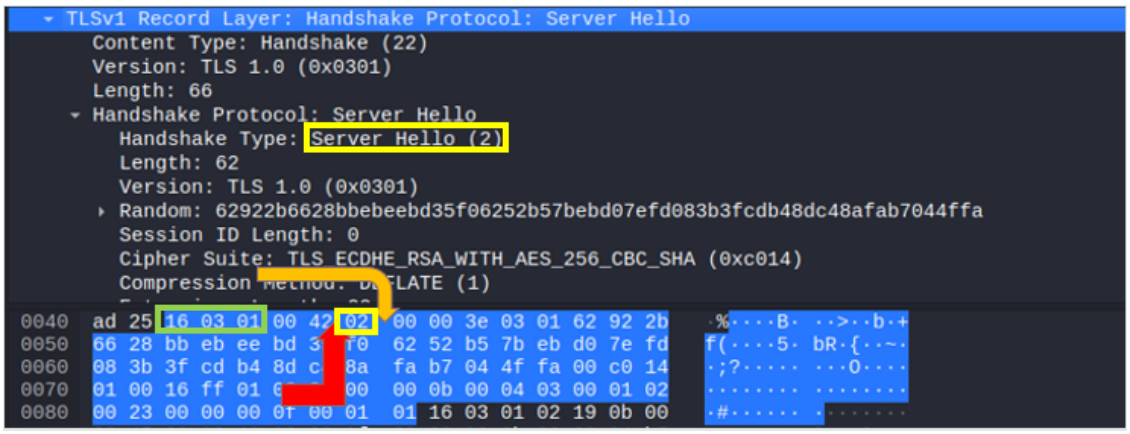


Figure 6.3. Server Hello Suricata rule

In Zeek is possible to add a new field in the log file by declaring in `/usr/local/zeek/share/zeek/base/protocols/ssl` the new field in the following way:

```
msg:     string &log &default="MESSAGE";
```

With this declaration, a string field with the default value MESSAGE is added to the log file. Now the value can be modified when the heartbeat extension is found in the following function:

```
event ssl_extension(c: connection, is_orig: bool, code: count, val:
    string) &priority=5
        {
        set_session(c);

        if (code == SSL_EXTENSION_HEARTBEAT )
                c$ssl$msg="HEARTBEAT";
        }
```

- CRIME - Suricata finds this kind of vulnerability thanks to the following rule:

```
alert tls any any -> any any (msg:"THIS IS COMPRESSED |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;)
```

As written in the content option of the rule, this specific rule is to find TLSv1.1 and as written in the first content option, which describes the type of the handshake message, is for the Server Hello message. The equivalent for Zeek is declaring the following variable:

```
compression:    string &log &default="MESSAGGIO";
```

This variable is modified in the event ssl_server_hello by the following code:

```
if( comp_method!=0){
                c$ssl$compression="COMPRESSION";
        }
```

- DROWN - The marker for this kind of attack is the SSLv2 version that is found in Suricata with the rule:

```
alert tls any any -> any any (msg:"RECORD LAYER +SSLv2+ |VULNERABILITY|
    #DROWN#";flow:from_server;content:"|16 00 02|";content:"|02|";
    distance:2;within:4; sid:221;)
```

And in Zeek, the version of the TLS version is written in the function ssl_server_hello by the following code:

```
if ( ! c$ssl?$version_num )
                {
                c$ssl$version_num = version;
                c$ssl$version = version_strings[version];
                }
```

This code is already written in the Zeek basic version.

- Bleichenbachers - The marker in this tool for this type of attack is the RSA cipher suite which is found n the traffic by the following rule in Suricata:

```
alert tls any any -> any any (msg:"|VULNERABILITY| #BLEICHENBACHER#
    %TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256%
    ";flow:from_server;content:"|16 03 00|";content:"|02|";
    distance:2;within:4; content:"|16 03 00|";content:"|C0 2f|";
    distance:73;within:2; sid:894;)
```

In Zeek the following code of the function ss_server_hello writes in the cipher variable the cipher suite used:

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

The Cipher Suite will be compared with an array of RSA cipher suites in the tool to check if present. This RSA cipher suite array is called **rsa_ciphers**

- Sweet32 - The marker is DES/3DES ciphersuite found by Suricata with the following rule:

```
alert tls any any -> any any (msg:"|VULNERABILITY| #SWEET32#
    %TLS_RSA_WITH_3DES_EDE_CBC_SHA% ";flow:from_server;content:"|16 03
    03|";content:"|02|"; distance:2;within:4; content:"|16 03
    03|";content:"|00 0a|"; distance:41;within:2; sid:330;)
```

In Zeek, the cipher suite is found in the same way used for the Bleichenbachers, but in this case, the cipher suite will be compared with another array with all the DES/3DES cipher suites in the tool **des_ciphers**.

- Logjam - In Suricata there are written all the rules with the EXPORT cipher suite to find this kind of vulnerability:

```
alert tls any any -> any any (msg:"|VULNERABILITY| #SWEET32#
    %TLS_RSA_WITH_3DES_EDE_CBC_SHA% ";flow:from_server;content:"|16 03
    03|";content:"|02|"; distance:2;within:4; content:"|16 03
    03|";content:"|00 0a|"; distance:41;within:2; sid:330;)
```

The cipher suite variable kept with Zeek will be compared with an Export cipher suite array called **export_ciphers** containing all the export cipher suites.

- Lucky13 - For each existing CBC cipher suite a rule like the following is written in Suricata:

```
alert tls any any -> any any (msg:"|VULNERABILITY| #LUCKY13#
    %TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256%
    ";flow:from_server;content:"|16 03 01|";content:"|02|";
    distance:2;within:4; content:"|16 03 01|";content:"|C0 23|";
    distance:41;within:2; sid:528;)
```

And like the above attacks, the cipher suite in Zeek will be compared with a CBC cipher suite array called **cbc_ciphers**

- ROBOT - This kind of vulnerability is checked also when a Bleichenbachers vulnerability is found.

- POODLE - The tool checks this kind of vulnerability with 2 markers. The first rule in Suricata is for an SSLv3 and the second is in the case of RSA_CBC Cipher Suite.

```
alert tls any any -> any any (msg:"RECORD LAYER +SSLv3+ |VULNERABILITY|
    #PADDING ORACLE ATTACK# -> POODLE";flow:from_server;content:"|16 03
    00|";content:"|02|"; distance:2;within:4; sid:883;)


alert tls any any -> any any (msg:"|VULNERABILITY| #PADDING ORACLE
    ATTACK# %TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5%
    ";flow:from_server;content:"|16 03 01|";content:"|02|";
    distance:2;within:4; content:"|16 03 01|";content:"|00 06|";
    distance:41;within:2; sid:35;)
```

Both rules are used also for the Padding Oracle Attack. Instead for Zeek, the found cipher suite in the log file must be present in both the array: **cbc_ciphers**, **rsa_ciphers**

- Ticketbleed - To find the TLS version Suricata uses the following rule with the different hexadecimal version described above:

```
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; sid:888;)
```

In Zeek, the version found as described above is then compared with the static string used by Zeek to describe the TLS version.

- Change Cipher Spec Injection - This marker is similar to the previous one but a rule is written also for the SSLv3.

```
alert tls any any -> any any (msg:"RECORD LAYER +TLSv10+ SERVER HELLO
    |VULNERABILITY| #CCSINJECTION#";flow:from_server;content:"|16 03
    01|";content:"|02|"; distance:2;within:4; sid:891;)
```

For Zeek the control is similar to the previous one.

- Roca Vulnerability - The Suricata's rules used for Bleichenbachers are also used for this kind of vulnerability. Zeek as for Bleichenbachers vulnerability controls the cipher suite found with the **rsa_ciphers** array.

6.2.1 Monitor for TLS attacks' code

Monitor_for_TLS_attacks.py

The tool starts like the majority of Python scripts with the main function. In this function, there is a check to enable the right features passed by the command line as parameters. First, the tool controls if more information is needed from the user with the parameter **-h** or –**help** or if the tool is used to run the single attack tool for a specific vulnerability. This is shown in figure 6.2.1. If no of both parameters is used, the tool is used in monitored mode and checks for the other parameters passed. For example, as shown in picture 6.4, here there is the verification of the IDS that the tool will be used.

```
if __name__ == "__main__":
    if sys.argv.__contains__("-h") or sys.argv.__contains__("--help"):
        help()
    else:
        if sys.argv.__contains__("--attack"):
            i = sys.argv.index("--attack")
            attack = sys.argv.pop(i + 1)
            j = sys.argv.index("--host")
            host = sys.argv.pop(j + 1)
            single_attack(attack, host)
```

```
if sys.argv.__contains__("--IDS=Zeek") or sys.argv.__contains__("--IDS=zeek") \
        or sys.argv.__contains__('--zeek') or sys.argv.__contains__('--Zeek'):
    print("ZEEK IDS")
    IDS = "ZEEK"
```

Figure 6.4. Control of IDS passed through command line

If no parameter –**IDS=[parameter]** or –**zeek** is passed, the default IDS used by the tool is Suricata. The other checks in the main function are used to:

- Full-mode - This mode is used in the monitored mode. If this mode is enabled by the command line, the flag **full_version** is set to true and this allows the use of the testssl tool to run generic tests against the TLS server like SSL labs. An example of the result of the full version is shown in picture 5.2;
- JSON file - If a JSON file is passed through the command line the tool initializes 3 arrays:
  1. versions_config - Contains all the versions allows in the network. If the tool sniffs a TLS packet with a version contained in the array, the tool does not start the TLS attacks against the server;
  2. ciphers_config - Contains all the ciphers allowed in the network. If the server uses a cipher suite contained in this array, the tool does not start an attack to verify the vulnerability;
  3. certificate_fingerprint_config - Contains the server's certificate fingerprint that the tool will not verify.

  and set to true the flag **config_input**. This flag is used in the tool to add more checks on the TLS versions, cipher suite, and certificate used in the TLS session.

The main function ends calling the **producer** and **verifica_vulnerabilita** functions. The producer function is shown in figure 6.5. In this function, a variable **src_path** is initialized depending

```
def producer():
    if IDS == "Suricata":
        src_path = r"/var/log/suricata/mio_fast.log"
    else:
        src_path = r"/usr/local/zeek/logs/current/ssl.log"
        file_exists = os.path.exists(src_path)

        if not file_exists:
            with open(src_path, 'x') as fp:
                fp.close()

    event_handler = Handler()
    observer = watchdog.observers.Observer()
    observer.schedule(event_handler, path=src_path, recursive=False)
    observer.start()
```

Figure 6.5. Producer Function

on the IDS used. In this variable is saved the log file's path of the IDS. As shown in figure 6.5, in case the tool uses Zeek IDS a control is added. This additional control on Zeek's log file is done because Zeek creates the ssl.log file only when the IDS sniffs TLS traffic. Instead, Suricata creates the log file from the beginning. Once the **src_path** is initialized the **event_handler** is created. This object is a custom item of **Handler** class. This class has 2 main functions:

1. __init__ - this function initializes the object. When the object is created the superclass **PatternMatchingEventHandler** is called to inherit all the superclass functions. This object is an event handler object that will be used later. After the initialized object, in this function, the IDS log file is opened, and the file pointer is saved in a local variable of the class Handler;

2. on_modified - this function is called when the IDS file opened is modified. When this function is called, a thread is started based on the IDS used. This thread will read the IDS changes and will perform string manipulation. The thread is started in daemon mode, this means that this thread does not block the main thread and continues to run in the background.

The Handler class is shown in figure 6.6. After creating the **event_handler** object, the **observer**

```
class Handler(watchdog.events.PatternMatchingEventHandler):
    def __init__(self):
        watchdog.events.PatternMatchingEventHandler.__init__(self,
                                                             ignore_directories=True, case_sensitive=False)
        if IDS == "Suricata":
            logfile = open(r"/var/log/suricata/mio_fast.log", "rb")
        else:
            logfile = open(r"/usr/local/zeek/logs/current/ssl.log")
        self.logfile = logfile


    def on_modified(self, event):

        if IDS == "Suricata":
            f = threading.Thread(target=file_reader_suricata, args=(self.logfile,), daemon=True)
        else:
            f = threading.Thread(target=file_reader_zeek, args=(self.logfile,), daemon=True)
        try:
            f.start()

        except (KeyboardInterrupt, SystemExit):
            print("---STOP - Keyboard Interrupt")
            sys.exit(2)
```

Figure 6.6. Handler Class

object is initialized. This object watches the path passed by the arguments. When the file is modified, the observer watches the modification and starts the **on_modified** function of the **PatternMatchingEventHandler** object. In this case the observer calls the **on_modified** function of **event_handler** object because it is a subclass of **PatternMatchingEventHandler** object. As shown in figure 6.6, when the IDS log file is modified 2 different functions are called based on the IDS used:

1. **file_reader_suricata**
2. **file_reader_zeek**

These 2 functions perform the string manipulation as described in subsection 6.1.1. The string manipulation is different based on the IDS used, but both functions use a condition variable to append in the **vuln_conn** dictionary the vulnerabilities and the host sniffed. The condition variable is used in the multi-threaded script to synchronize the different threads that want to use shared objects. The condition variable allows a thread to enter in a critical section where the shared object is used only if a condition is satisfied. In the tool,**cv** condition variable allows the verifier thread to start only if the dictionary is not empty. The same condition variable stops the producer thread if the dictionary is full. In the above producer functions the cv uses 4 main methods:

1. acquire - this method allows to acquire the underlying lock. A lock is a primitive object in Python that has 2 states: blocked or unblocked. When a lock is acquired, its status becomes blocked. When the status of a thread is blocked, the lock blocks all subsequent attempts to acquire the lock;
2. release - this method changes the status of the underlying lock from blocked to unblocked;

3. wait - this method releases the underlying lock and then blocks until it is awakened by a notify(). This method is used in the producer thread if the **vuln_coon** is full;

4. notify - this method wakes up one thread waiting on this condition, if any. When the tool starts this method is used to wake up the verifier thread.

In picture 6.7 is shown how the **cv** object is used in the **suricata_reader** function. After the **producer** function starts the observer object to watch the logfile, it returns to the main function and starts the **verify_vulnerability** function that represents the verifier thread.

```
if vuln[1] != '':
    cv.acquire()

    cipher_suite = vuln[2].split("%")
    if len(cipher_suite) > 1:
        cipher_suite_found = cipher_suite[1]

    vulnerabilities = vuln[2].split("#")
    new_vulnerability = vulnerabilities[1]
    flag = False
    if new_vulnerability == 'CERTIFICATE' or new_vulnerability == 'SELF SIGNED' or new_vulnerability == 'EXPIRED' \
            or new_vulnerability == 'CRIME' or new_vulnerability=='HEARTBEAT EXTENSION':
        flag = True
    if tls_version_found not in versions_config and tls_version_found != '':
        flag = True
    if cipher_suite_found not in ciphers_config and cipher_suite_found != '':
        flag = True

    if flag:
        if source_dest not in vuln_conn.keys():
            vuln_conn[source_dest] = []

        if new_vulnerability not in vuln_conn[source_dest]:...

        if len(vuln_conn) == MAX_NUM:
            log_print(f'The queue is full. Producer Thread waits.')
            cv.wait()
            log_print(f'The queue is no longer full. Producer Thread wakes up.')
        cv.notify()
    cv.release()
```

Figure 6.7. Use of cv in the reader_suricata function

This thread waits until an object is appended in the **vuln_conn** dictionary. When it wakes up, pop the first object in the dictionary and start with a string manipulation to retrieve the IP address of the client and the server, and their port, on which the TLS connection starts. After this manipulation, there is a switch constructor to start the specific attack with the possible vulnerability found. An example is shown in figure 6.8 where if the producer thread append the string **HEARTBEAT EXTENSION**, the consumer thread starts all the attack tools to verify the vulnerability.

The thread is created and started in the consumer thread. Each thread calls the specific function to verify with the specific tool the vulnerability. For example in figure 6.8, the thread **heartblee_metasploit** calls the function **metasploit_process** passing its the ip of the server, the port of the server and the tls version used in the session as arguments.

The same Producer-Consumer mechanism is used also for the **verify_vulnerability** and the function **scheduling**. This mechanism is used once more to manage the number of the attacks thread used. All the threads, after starting, are added into a thread array **jobs**. This array is used in function **scheduling** to inform the user if no attacks is still running and to verify if some thread is still alive. As shown in figure 6.9 the **scheduling** function is a self-called function that is called every 3 seconds to manage the **jobs** array.

```
for test_vuln in all_vuln_for_conn:

    if test_vuln == "HEARTBEAT EXTENSION":
        tls_version = tls_version_conn.pop(connessioni)
        heartbleed_metasploit = threading.Thread(target=metasploit_prosess,
                                                 args=(ip_source, port_source, tls_version))
        heartbleed_metasploit.start()
        log_print_attack(ip_source, port_source, 'HEARTBEED WITH METASPLOIT')
        job.append(heartbleed_metasploit)

        heartbleed_nmap = threading.Thread(target=nmap_process, args=(ip_source, port_source,))
        heartbleed_nmap.start()
        log_print_attack(ip_source,port_source,'HEARTBLEED WITH NMAP')
        job.append(heartbleed_nmap)

        #HeartBleed TestSSL
        heartbleed_testssl= threading.Thread(target=testssl_heartbleed_process, args=(ip_source, port_source,))
        heartbleed_testssl.start()
        log_print_attack(ip_source,port_source,'HEARTBLEED WITH TESTSSL')
        job.append(heartbleed_testssl)

        #Heartbleed TLS-Attacker
        heartbleed_tls_attacker_thread= threading.Thread(target=heartbleed_tls_attacker, args=(ip_source, port_source,))
        heartbleed_tls_attacker_thread.start()
        log_print_attack(ip_source,port_source,'HEARTBLEED WITH TLS-ATTACKER')
        job.append(heartbleed_tls_attacker_thread)
```

Figure 6.8. Verifier thread starts tool for Heartbleed vulnerability

```
def scheduling():
    global job
    global active_timer
    try:
        if active_timer:
            if job.__len__()==0:
                log_print(f'SCHEDULING: No attack is running')
            else:
                while job.__len__()>0:
                    cv_jobs.acquire()
                    test=job.pop()
                    if test.is_alive():
                        test.join()

                    cv_jobs.notify()
                    cv_jobs.release()

                t=threading.Timer(3, scheduling)
                t.start()
            else:
                raise KeyboardInterrupt

    except (KeyboardInterrupt, SystemExit,RuntimeError):

        for test in job:
            print("----STOP -> KEYBOARD INTERRUPT-----")
            test.join()
        print("END OF SCHEDULING")
        sys.exit(3)
```

Figure 6.9. Scheduling function

The specific attack function uses the **SubProcess** python library to launch the attack on the command line. An example is shown in figure 6.10.

The **SubProcess** allows to run the specific command through the function **Popen** that can capture also the output and the error of the command through the **stderr=subprocess.PIPE** and **stdout=subprocess.PIPE**. These values are stored in **stdout** and **stderr** variables that will be used to write the information retrieved on the command line and in the log file. The

```python
def metasploit_prosess(ip, port, tls_version):
    command = f"use auxiliary/scanner/ssl/openssl_heartbleed;set RHOST {ip};set RPORT {port};set TLS_VERSION {tls_version};check;exit"
    try:
        metasploit = subprocess.Popen(['msfconsole', '-x', command], stdout=subprocess.PIPE,stderr=subprocess.PIPE)

        stout,stderr = metasploit.communicate()
        stout=stout.decode()
        if stout[2].removeprefix(" ").__contains__('The target appears to be vulnerable.'):
            commandKey = f"use auxiliary/scanner/ssl/openssl_heartbleed;set RHOST {ip};set RPORT {port};set TLS_VERSION {tls_version};keys;exit"
            metasploitKey = subprocess.Popen(['msfconsole', '-x', commandKey], stdout=subprocess.PIPE)
            keyout = metasploitKey.communicate()[0].decode()
            privateKey = keyout.rpartition('-----BEGIN RSA PRIVATE KEY-----')
            if privateKey[2] != '':
                privateKey = privateKey[2].rpartition('-----END RSA PRIVATE KEY-----')[0]
                privateKey = f'-----BEGIN RSA PRIVATE KEY-----{privateKey}-----END RSA PRIVATE KEY-----'
                # print(privateKey)
                write_file(ip, port, keyout, 'PRIVATE KEY RETRIEVED BY METASPLOIT')
                log_print(
                    f'The {ip}:{port} is vulnerable to Heartbleed. The private key is stolen and it is in the log file')

        write_file(ip, port, stout[2].removeprefix(" "), 'HEARTBLEED BY METASPLOIT')
    except (SystemExit, KeyboardInterrupt, subprocess.CalledProcessError, ValueError, OSError,
            subprocess.SubprocessError,RuntimeError):
        print("Interrupt Metasploit Process for KeyInterrupt")
```

Figure 6.10. Specific function to run Metasploit for Heartbleed Vulnerability

informations are written in the log file through the **write_file** and are printed on the command line through the **log_print**. These functions are used for each attack function and are used also in the program to log the most important information. This schema is used for all the attacks.

certificate.py

The verification of the certificate is a little different. It runs also in a thread and this thread is added to the **jobs** array, but the verification does not use the subprocess library. It uses some specific function to retrieve the server certificate and verify its OCSP, CRL, and if it uses Certification Transparency. The certificate verification starts in the consumer thread, as shown in figure 6.11. The certificate is first retrieved and saved in the **cert** variable to verify if it is self-signed, not valid yet, or expired. After this verification the **test_certificate** thread starts calling the **get_cert_status_for_host**, passing the server IP and port, and the certificate retrieved as arguments.

```python
if test_vuln == 'CERTIFICATE':

    [cert, cert_string] = get_cert_for_hostname(ip_source, port_source)
    certificate_monitored_fingerprint = get_certificate_fingerprint(cert,ip_source,port_source)
    if certificate_monitored_fingerprint not in certificate_fingerprint_config:
        try:
            if cert.issuer == cert.subject:
                log_print(f"{COLOR['RED']} THE CERTIFICATE FOR CONNECTION: {connessioni} IS SELF SIGNED {COLOR['ENDC']}")
                write_file(ip_source, port_source, f"THE CERTIFICATE FOR CONNECTION: {connessioni} IS SELF SIGNED","Certificate Self Signed")

            if cert.not_valid_after < datetime.now():
                log_print(f"{COLOR['RED']} THE CERTIFICATE FOR CONNECTION: {connessioni} IS EXPIRED -> Date Valid Until: {cert.not_valid_after} {COLOR['ENDC']}")
                write_file(ip_source, port_source,
                           f"THE CERTIFICATE FOR CONNECTION: {connessioni} IS SELF EXPIRED","Certificate Expired")
            if cert.not_valid_before > datetime.now():
                log_print(
                    f"{COLOR['RED']} THE CERTIFICATE FOR CONNECTION: {connessioni} IS NOT VALID YET -> Date Valid after: {cert.not_valid_beforer} {COLOR['ENDC']}")
                write_file(ip_source, port_source,
                           f"THE CERTIFICATE FOR CONNECTION: {connessioni} IS NOT VALID YET",
                           "Certificate Expired")

            test_certificate = threading.Thread(target=get_cert_status_for_host, args=(ip_source, port_source,cert,cert_string),
                                                daemon=True)
            test_certificate.start()
            log_print(f'Start test for certificate in connection {connessioni}')
```

Figure 6.11. Start certificate verification

In this function first, the CRL and the OCSP are retrieved from the certificate through the extensions and then verified. The same is done for the SCT, first is retrieved and stored in the **sct_cert** variable and then it is verified through the **sct_web** function. First, the function tries to retrieve the SCT from the certificate. If there is no SCT in the certificate, the function tries to

retrieve the SCT through the OCSP and through the TLS extensions. If no SCT has been found an alarm is printed on the command line and the result is written in the log file. If the SCT has been found, the function uses the log list **ctlogs** downloaded when the tool started to verify the SCT. The **sct_web** function is shown in figure 6.12 and in figure 6.13

```
def downloadLogList():
    global ctlogs
    ctlogs = download_log_list()

def sct_web(hostname, port, sct_cert):
    global ctlogs
    try:
        handshake = do_handshake(hostname, int(port))
        if handshake.__getattribute__('err')!='':
            log_print(f"{COLOR['RED']} SCT Found for connection {hostname}:{port} but Octect is not correct.{COLOR['ENDC']}")
            write_file(hostname,port,f'SCT Found for connection {hostname}:{port} but Octect is not correct.','SCT Error')
        else:
            if sct_cert.__len__() > 0:
                log_print(f"{COLOR['YELLOW']}Start Test to verify Certificate Transparency by SCT extension into the certificate")
                verification_cert = verify_scts_by_cert(handshake, ctlogs)
                verify_sct = ''
                for ver in verification_cert:
                    if ver.verified:
                        description = ver.log["description"]
                        verify_sct = f"{COLOR['GREEN']}For {hostname}:{port} -> {ver.verified}: {description}{COLOR['ENDC']}"
                        verify_sct_file=f'For {hostname}:{port} -> {ver.verified}: {description}'
                    else:
                        verify_sct = f"{COLOR['RED']}SCT NOT VERIFIED for {hostname}:{port}{COLOR['ENDC']}"
                        verify_sct_file=f'SCT NOT VERIFIED for {hostname}:{port}'

                    log_print(verify_sct)
                    write_file(hostname, port, verify_sct_file, 'Certificate Transparency by SCT')
```

Figure 6.12. SCT function

```
    else:
        log_print(f"{COLOR['YELLOW']}Start Test to verify Certificate Transparency by OCSP{COLOR['ENDC']}")
        verification_ocsp = verify_scts_by_ocsp(handshake, ctlogs)
        if verification_ocsp.__len__() == 0:
            log_print(f"{COLOR['RED']}NO SCT FOUND BY OCSP for {hostname}:{port}{COLOR['ENDC']}")
            write_file(hostname, port, f"NO SCT FOUND BY OCSP for {hostname}:{port}",
                       'Certificate Transparency by OCSP')
        else:
            for ver in verification_ocsp:
                ocsp_verify = ''
                if ver.verified:
                    description = ver.log["description"]
                    ocsp_verify = f"For {hostname}:{port} -> {ver.verified}: {description} - {ver}"
                else:
                    ocsp_verify = f"{COLOR['RED']}For {hostname}:{port} -> SCT NOT VERIFIED{COLOR['ENDC']}"

                log_print(ocsp_verify)
                write_file(hostname, port, ocsp_verify, 'Certificate Transparency by OCSP')

        log_print(f'{COLOR["YELLOW"]} Verify Certificate Transparency through TLS Extension{COLOR["ENDC"]}')
        verification_tls = verify_scts_by_tls(handshake, ctlogs)
        final_tls_string = ''
        if verification_tls.__len__() == 0:
            action = f"For {hostname}:{port} -> NO SCT FOUND IN TLS EXTENSION"
            log_print(f'{COLOR["RED"]}{action}{COLOR["ENDC"]}')
            final_tls_string = action
        else:
            for ver in verification_tls:
                description = ver.log["description"]
                action = f"For connection {hostname}:{port} -> {ver.verified}: {description} - {ver}"
                if ver.verified:
                    log_print(f'For connection {hostname}:{port} TLS Extension is verified and writted in file')
                else:
                    # log_print(action)
                    log_print(
                        f"{COLOR['RED']}For connection {hostname}:{port} -> TLS Extension NOT VERIFIED. For more detail read the log file.{COLOR['ENDC']}")
                    # log_print(action)
                final_tls_string = f'{final_tls_string}\n{action}'

        write_file(hostname, port, final_tls_string, 'Certificate Transparency through TLS Extension')
```

Figure 6.13. SCT function

6.3 Testbed

To test the Monitor for TLS attacks more versions of OpenSSL and Apache2 have been installed on the machines. As described in the OpenSSL's vulnerability page[62] some library's versions are vulnerable to specific attacks. The following table summarizes the OpenSSL versions installed on the machines and the vulnerabilities associated with each version.

Table 6.2. Vulnerabilities associated with old Openssl version

Apache Version	Openssl Version	TLS Vulnerabilities	Referenceces
	OpenSSL 0.9.7	Bleichenbachers CCSInjection POODLE	CVE-2012-0884[63] CVE-2014-0224[64] CVE-2014-3566[31]
Apache2 v2.51.4	OpenSSL 1.0.1c	Heartbleed CCSInjection Lucky13 POODLE	CVE-2014-0160[42] CVE-2014-0224[64] CVE-2013-0169[65] CVE-2014-3566[31]
Apache2 v2.47.2	OpenSSL 1.0.2	Padding Oracle Attack Sweet32 DROWN	CVE-2016-2107[67] CVE-2016-2183[66] CVE-2016-0800 [70]

Some attacks integrated into Monitor for TLS attacks tool require specific hardware. In the following table, there is a summary of the TLS attacks that can't be tested and the associated references.

Table 6.3. 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[68]
Ticketbleed	F5 BIG-IP LTM, AAM, AFM, Analytics, APM, ASM, GTM, Link Controller, PEM, PSM	CVE-2016-9244[69]
ROBOT	BIG-IP (F5 vendor), Citrix NetScaler,etc..	Diffrent CVEs[29]

The commands used to install the specific version of OpenSSL and Apache2 are described into A.2

6.3.1 Network Configuration

In the laboratory to test the Monitor for TLS attacks tool, 3 machines are used:

- Server - This machine has an Apache service running linked with an openssl vulnerable library. ;
- Client - This machine is used to visit the server's web page It uses commands like wget or openssl to connect and download the server's web page;
- Monitor - This machine uses ettercap to intercept all the traffic between client and server. This sniffed traffic is analyzed by the IDS and passed to the Monitor for TLS attacks tool.

An example of the network configured in the laboratory is represented by picture 6.14.

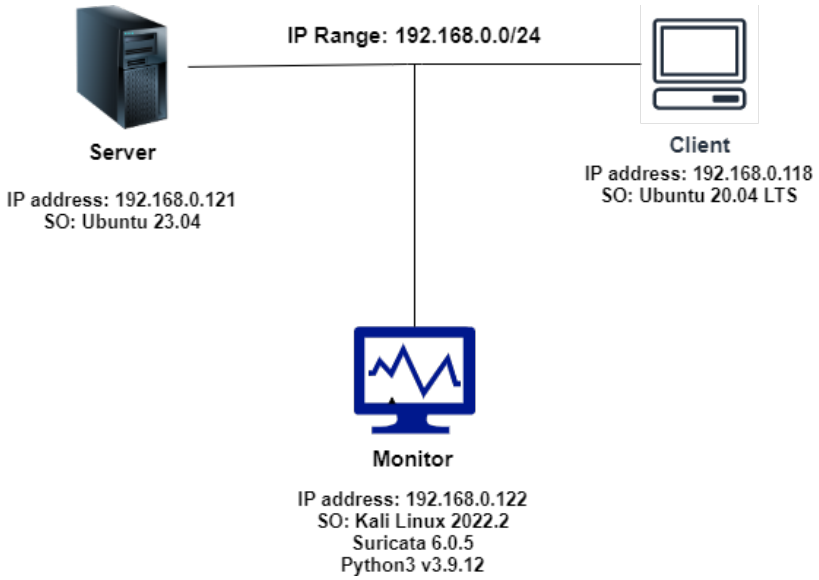


Figure 6.14. Network configured in laboratory

6.3.2 Single Vulnerability test

After installing the specific vulnerable library on the server machine, the test described into A.2.6 was performed. The test is done both with Suricata and Zeek. For each vulnerability, 10 packets are sent from the client to the server and sniffed by the Monitor. The test results are shown in figure 6.15.

This figure shows for each vulnerability, how long the Monitor for TLS attacks tool takes to run all the attacks for that specific vulnerability. The vulnerability that takes more time than the other is the Heartbleed vulnerability.

This vulnerability takes more time than the other because it runs 4 tools to verify the vulnerability (TestSSL, Nmap, TLS-Attacker, Metasploit) and Metasploit is run twice if a machine is exploitable. The first time the attack verifies if the machine is vulnerable and the second time it tries to retrieve the private key of the server through the Heartbleed vulnerability.

The graphic shows also the standard deviation of the result. This standard deviation is the vertical line present on each column. The heartbleed vulnerability has the highest standard deviation and this is done by the higher network dependency. This network dependency, as described for the speed measurement, is done because for this vulnerability 5 attacks are run.

The measurement for these tests starts from the moment that the IDS sniffs the packets on the network and finishes when all the attacks tool end. The figure 6.15 shows that the IDS takes about the same amount of time. The only difference is that Zeek can't sniff the SSLv2 traffic. In fact, as shown in figure 6.15 it can not sniff traffic for the DROWN vulnerability.

Once test how much time all the attacks take for a specific vulnerability, the Monitor for TLS attacks tool is tested by sending a packet in a network shown in figure 6.14. This time, the tool launches all the tools for each possible vulnerability found in the packet. The figure 6.16 shows how much time the tool takes for a packet sent with each library. The test with Openssl 1.0.1c is done twice because the second time the packet is sent with a vulnerable CBC cipher to start more attack tools.

For both the IDS, the Monitor for TLS attacks starts the same number of attack tools that are summarized in table 6.4 :

The measurement shown in table 6.4 represents a packet sent 10 times. Each time the tool starts the same attacks. The attacks are then analyzed more in detail to find how many False Positive (FP), True Positive (TP), False Negative (FN), and True Negative (TN) the Monitor for TLS attacks tool has been found. The result of this analysis is shown in tables 6.5 and 6.6 .

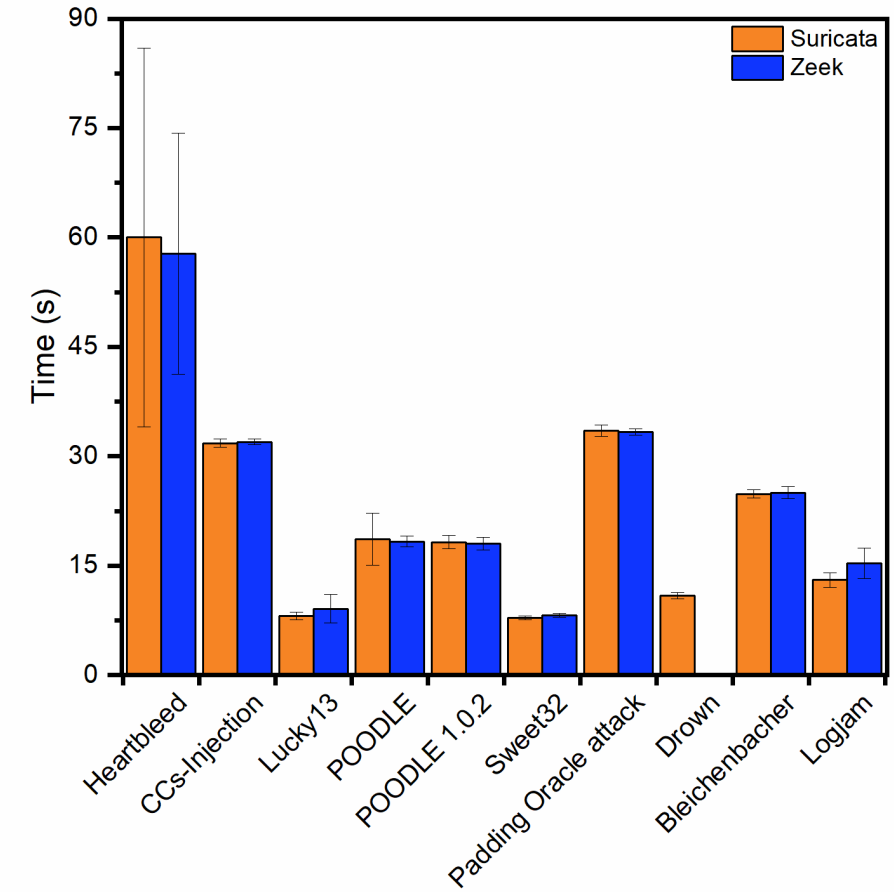


Figure 6.15. Test results for single vulnerability

Table 6.4. Attacks started for each Openssl version installed

IDS	Openssl Version	n. attacks started	Average time (s)	Standard Deviation (s)
Suricata	OpenSSL 1.0.1c	11	95,658	67,788
	OpenSSL 1.0.1c + cipher	16	111,211	30,754
	OpenSSL 1.0.2	12	148,954	6,492
	OpenSSL 0.9.7	12	103,455	2,763
Zeek	OpenSSL 1.0.1c	11	53,363	5,238
	OpenSSL 1.0.1c + cipher	16	123,288	18,397
	OpenSSL 1.0.2	12	141,024	2,551
	OpenSSL 0.9.7	12	129,023	4,909

For both IDS, with the library **OpenSSL 1.0.1c** another attack is started. This attack is not counted in Table 6.5 and 6.6 because this attack started only if the target is vulnerable to a Heartbleed attack. This additional attack tries to retrieve the target's private key through the Heartbleed vulnerability. This attack has retrieved the target's private key 6 times on 10 with Suricata and 7 times on 10 with Zeek. The False Negatives for the same library are the result of the Heartbleed vulnerability with the **TLS-Attacker** and **TestSSL**. In Table 6.5 and 6.6 is shown for all the POODLE attacks a TN for the 1.0.1c+cipher library. This is strange because the library is vulnerable to the POODLE attack, but a deep analysis of Apache's configuration shows that the server has disabled the SSL 3.0 version of TLS. For this reason, the results in the tables are set to TN.

Table 6.5. Results for TLS attack tools for Zeek

OpenSSL Version	Attack	TP	FP	FN	TN
OpenSSL 1.0.1c	Heartbleed with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c	Heartbleed with Nmap	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c	Heartbleed with TLS-Attacker	7/10	0/10	3/10	0/10
OpenSSL 1.0.1c	Heartbleed with testssl	0/10	0/10	10/10	0/10
OpenSSL 1.0.1c	Bleichenbacher with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c	Bleichenbacher with Metasploit	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c	CCS-Injection with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c	CCS-Injection with Nmap	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c	Robot with testssl	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c	ROCA with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c	Ticketbleed with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	Heartbleed with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c + cipher	Heartbleed with Nmap	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c + cipher	Heartbleed with TLS-Attacker	0/10	0/10	10/10	0/10
OpenSSL 1.0.1c + cipher	Heartbleed with testssl	0/10	0/10	10/10	0/10
OpenSSL 1.0.1c + cipher	Bleichenbacher with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	Bleichenbacher with Metasploit	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	CCS-Injection with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c + cipher	CCS-Injection with Nmap	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c + cipher	Robot with testssl	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	ROCA with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	Ticketbleed with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	Lucky13 with testssl	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c + cipher	POODLE with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	POODLE with testssl	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	POODLE with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	Padding Oracle Attack with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Bleichenbacher with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Bleichenbacher with Metasploit	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	CCS-Injection with Metasploit	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	CCS-Injection with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Robot with testssl	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	ROCA with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Ticketbleed with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Lucky13 with testssl	0/10	0/10	0/10	0/10
OpenSSL 1.0.2	POODLE with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	POODLE with testssl	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	POODLE with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Padding Oracle Attack with TLS-Attacker	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	Bleichenbacher with TLS-Attacker	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	Bleichenbacher with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	CCS-Injection with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	CCS-Injection with Nmap	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	Robot with testssl	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	ROCA with Nmap	0/10	0/10	0/10	10/10
OpenSSL 0.9.7	Ticketbleed with Nmap	0/10	0/10	0/10	10/10
OpenSSL 0.9.7	Lucky13 with testssl	0/10	0/10	0/10	0/10
OpenSSL 0.9.7	POODLE with Nmap	0/10	0/10	0/10	10/10
OpenSSL 0.9.7	POODLE with testssl	0/10	0/10	0/10	10/10
OpenSSL 0.9.7	POODLE with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 0.9.7	Padding Oracle Attack with TLS-Attacker	0/10	0/10	0/10	10/10

Table 6.6. Results for TLS attack tools for Zeek

Openssl Version	Attack	TP	FP	FN	TN
OpenSSL 1.0.1c	Heartbleed with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c	Heartbleed with Nmap	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c	Heartbleed with TLS-Attacker	3/10	0/10	7/10	0/10
OpenSSL 1.0.1c	Heartbleed with testssl	0/10	0/10	10/10	0/10
OpenSSL 1.0.1c	Bleichenbacher with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c	Bleichenbacher with Metasploit	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c	CCS-Injection with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c	CCS-Injection with Nmap	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c	Robot with testssl	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c	ROCA with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c	Ticketbleed with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	Heartbleed with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c + cipher	Heartbleed with Nmap	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c + cipher	Heartbleed with TLS-Attacker	0/10	0/10	10/10	0/10
OpenSSL 1.0.1c + cipher	Heartbleed with testssl	0/10	0/10	10/10	0/10
OpenSSL 1.0.1c + cipher	Bleichenbacher with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	Bleichenbacher with Metasploit	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	CCS-Injection with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c + cipher	CCS-Injection with Nmap	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c + cipher	Robot with testssl	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	ROCA with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	Ticketbleed with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	Lucky13 with testssl	10/10	0/10	0/10	0/10
OpenSSL 1.0.1c + cipher	POODLE with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	POODLE with testssl	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	POODLE with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.1c + cipher	Padding Oracle Attack with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Bleichenbacher with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Bleichenbacher with Metasploit	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	CCS-Injection with Metasploit	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	CCS-Injection with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Robot with testssl	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	ROCA with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Ticketbleed with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Lucky13 with testssl	0/10	0/10	0/10	0/10
OpenSSL 1.0.2	POODLE with Nmap	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	POODLE with testssl	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	POODLE with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 1.0.2	Padding Oracle Attack with TLS-Attacker	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	Bleichenbacher with TLS-Attacker	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	Bleichenbacher with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	CCS-Injection with Metasploit	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	CCS-Injection with Nmap	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	Robot with testssl	10/10	0/10	0/10	0/10
OpenSSL 0.9.7	ROCA with Nmap	0/10	0/10	0/10	10/10
OpenSSL 0.9.7	Ticketbleed with Nmap	0/10	0/10	0/10	10/10
OpenSSL 0.9.7	Lucky13 with testssl	0/10	0/10	0/10	0/10
OpenSSL 0.9.7	POODLE with Nmap	0/10	0/10	0/10	10/10
OpenSSL 0.9.7	POODLE with testssl	0/10	0/10	0/10	10/10
OpenSSL 0.9.7	POODLE with TLS-Attacker	0/10	0/10	0/10	10/10
OpenSSL 0.9.7	Padding Oracle Attack with TLS-Attacker	0/10	0/10	0/10	10/10

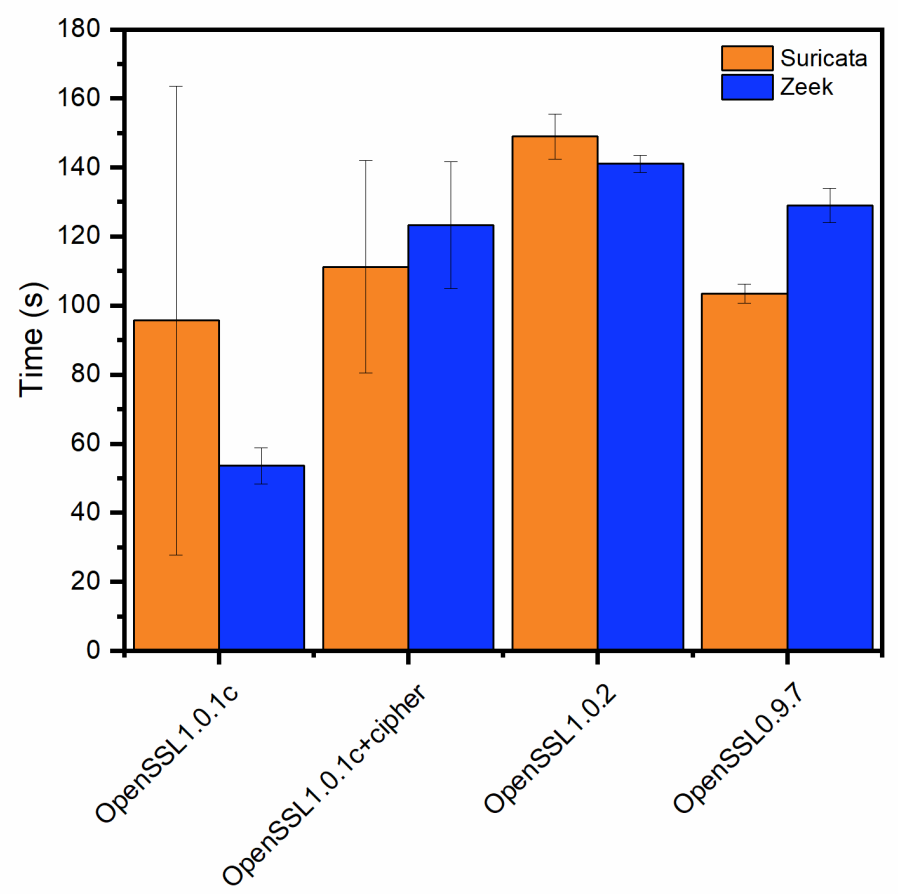


Figure 6.16. Time spent with all attacks for one packet. Packet sniffed for: **OpenSSL 1.0.1c**: Heartbeat:YES, TLS Version: 1.2, Cipher:ECDHE-RSA-AES256-GCM-SHA384. **OpenSSL 1.0.1c+cipher**: Heartbeat:YES, TLS Version: 1.2, Cipher:DES-CBC3-SHA. **OpenSSL 1.0.2**: Heartbeat:NO, TLS Version: 1.2, Cipher:DES-CBC3-SHA. **OpenSSL 0.9.7**: Heartbeat:NO, TLS Version: 1.2, Cipher:DES-CBC3-SHA.

6.3.3 Configuration file test

Another feature of the Monitor for TLS attacks tool is to import from the command line a JSON file. This JSON file is used as a white list to allow some specific traffic. In this file is possible to write versions of TLS, cipher suites, and specific the certificate fingerprint. All the values inserted in this file will not control by the tool. When the tool starts, in the main function there is a control to verify if the JSON file is passed as an argument. If it is true, the tool stores the value written in the JSON file. All the future sniffed traffic will be checked with this value and if they match, no attack tool will be launched. The JSON file must be present in the directory of the Monitor for TLS Attacks tool in its own directory `./JSON/config.json`.

The figure 6.17 shows 2 TLS sniffed connections. The 2 connections can be found by the row:

```
Found new connection: 10.0.2.15:443 --> 10.0.2.4
```

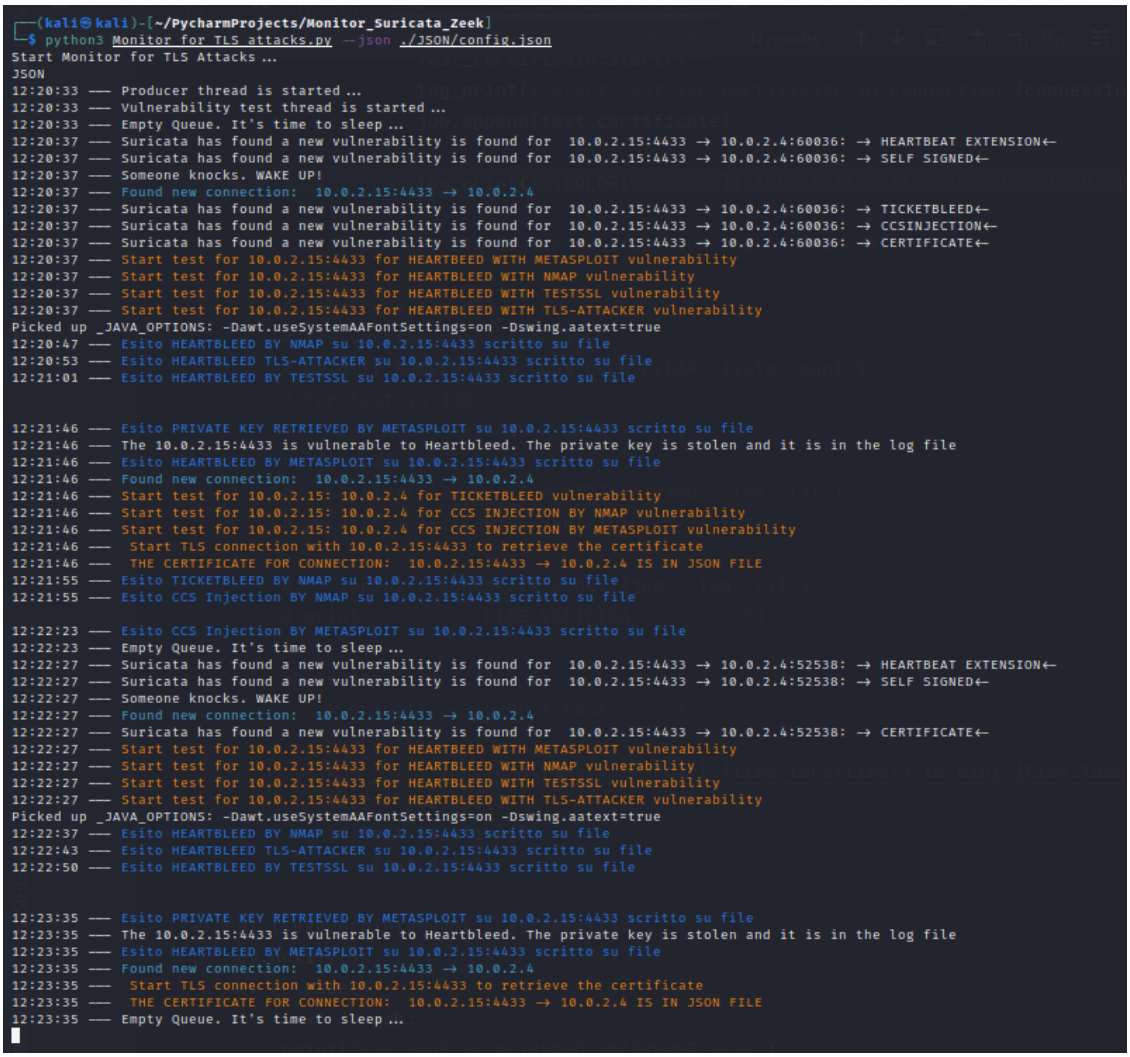


Figure 6.17. Tool in action with JSON config file

Both the sniffed packets are SERVER HELLO. This message is from the server 10.0.2.15 to the client 10.0.2.4. As shown in figure 6.17 the first connection found more possible vulnerability than the second. This is because the first connection is made through TLSv1.2 which is not present in the JSON file. The second connection does not find the same vulnerability because the second connection is made with TLSv1.1 which is present in the JSON file. Both the connection are made with **DES-CBC3-SHA** cipher suite that is allowed because it is present in the JSON

file. Finally the certificate is not checked because its fingerprint is written in the JSON file. As shown in the figure 6.17 the tool writes the following row to inform that the certificate is present:

```
THE CERTIFICATE FOR CONNECTION 10.0.2.15:4433 --> 10.0.2.4 IS IN JSON FILE
```

6.3.4 Certification Transparency test

As described in section 6.2.1 the Monitor for TLS attacks tool can verify the CRL, OCSP and the Certification Transparency of a certificate. When the tool runs, it shows to the command line a brief description of the action done. The figure 6.18 shows what the tool write on the command line. As shown in figure 6.18, the tool informs immediately if the CRL, OCSP are found. If it does not found the CRL or the OCSP, it write in red an alarm to inform the user. The tool verify if the tested server use the SCT in 3 ways:

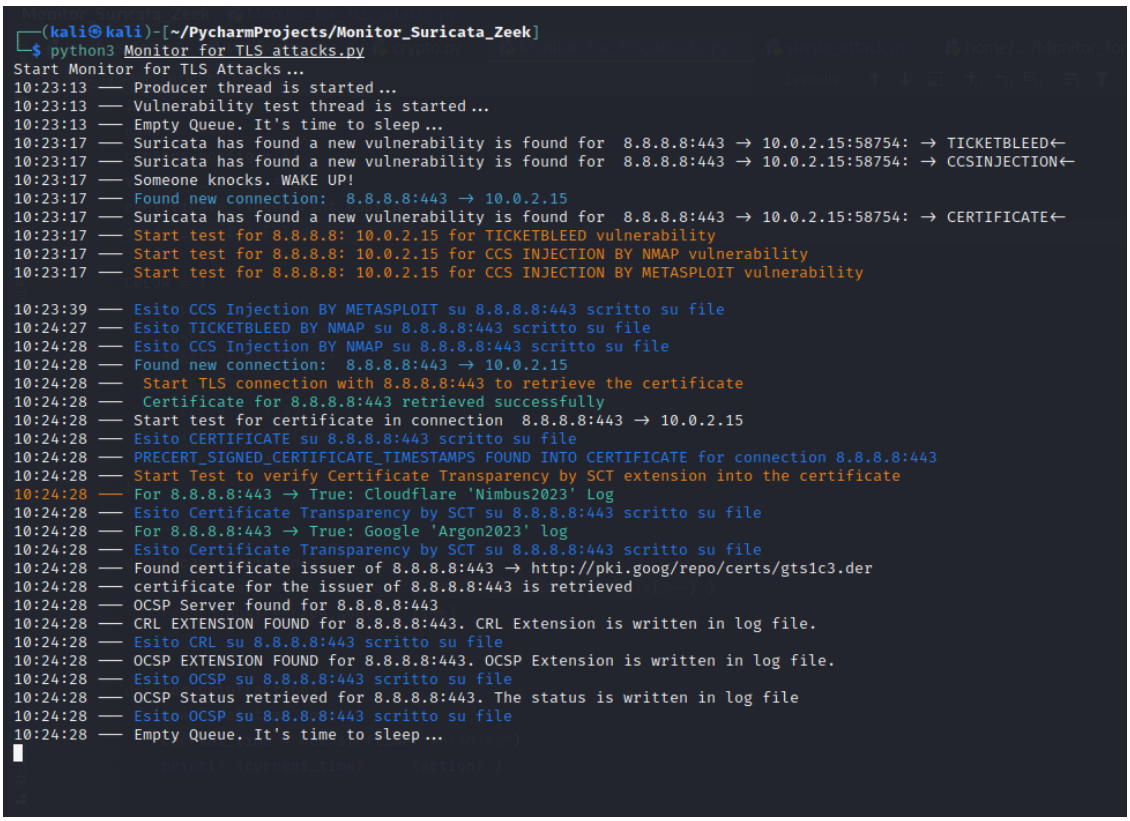


Figure 6.18. Certificate verification by Monitor for TLS attacks tool

1. SCT in the certificate - this is the most common way to use the SCT. In fact the tool first check if there is an SCT in the certificate. If it found the SCT, it does not use the other way to retrieve the SCT;
2. SCT in OCSP - This is the second method used in the tool;
3. SCT in TLS extension - this verification is made also if the 2 above does not find any SCT.

More information are written in the log file for the specific tested server in **OCSP.log** and **CRL.log**. Both file for the above test are shown in figure 6.19 and figure 6.20.

A verification of the information retrieved by the Monitor for TLS attacks tool can be the figure 6.21 that represents the certificate of the server tested by the tool. The figure 6.21 shows that the OCSP's URL is correct. It shows also that the certificate has 2 SCT. The first's name is

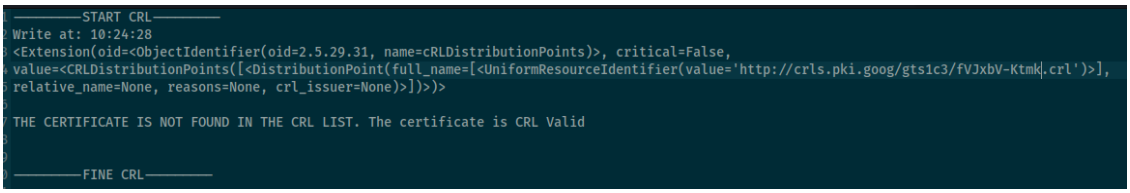
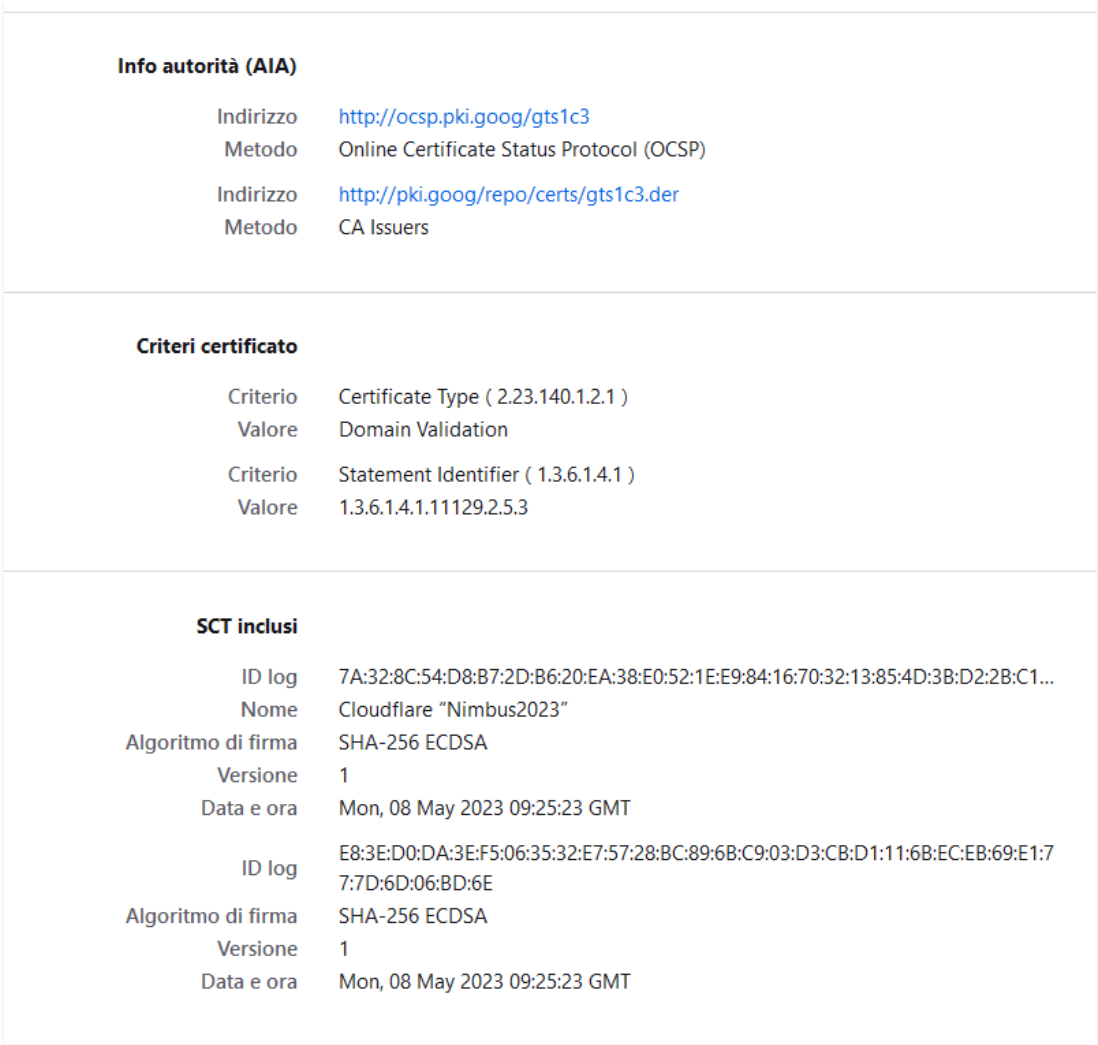


Figure 6.19. CRL.log file

Figure 6.20. OCSP.log file

Nimbus2023 which is the same that the tool verified. The second has no name in the certificate, but as shown in figure 6.18 the name of the second SCT is Argon2023. Both SCT are correct and verified.



Info autorità (AIA)

Indirizzo	http://ocsp.pki.goog/gts1c3
Metodo	Online Certificate Status Protocol (OCSP)
Indirizzo	http://pki.goog/repo/certs/gts1c3.der
Metodo	CA Issuers

Criteri certificato

Criterio	Certificate Type (2.23.140.1.2.1)
Valore	Domain Validation
Criterio	Statement Identifier (1.3.6.1.4.1)
Valore	1.3.6.1.4.1.11129.2.5.3

SCT inclusi

ID log	7A:32:8C:54:D8:B7:2D:B6:20:EA:38:E0:52:1E:E9:84:16:70:32:13:85:4D:3B:D2:2B:C1...
Nome	Cloudflare "Nimbus2023"
Algoritmo di firma	SHA-256 ECDSA
Versione	1
Data e ora	Mon, 08 May 2023 09:25:23 GMT
ID log	E8:3E:D0:DA:3E:F5:06:35:32:E7:57:28:BC:89:6B:C9:03:D3:CB:D1:11:6B:EC:EB:69:E1:77:7D:6D:06:BD:6E
Algoritmo di firma	SHA-256 ECDSA
Versione	1
Data e ora	Mon, 08 May 2023 09:25:23 GMT

Figure 6.21. Certificate tested server

6.3.5 MITM test

The tool is developed to verify the TLS traffic in a small network. The goal of the tool is to run on the same network as the server. In this way, the tool can control the TLS traffic for the server and if it sniffs some vulnerable TLS packets, it starts the attack against the server to verify if the possible vulnerable sniffed is True Positive. Another way to use the monitor is on the client network to know if a client is under a MITM attack. For this test, 3 machines are used:

1. Client - This machine is under a MITM attack and tries to connect to a server on the Internet;
2. MITM Attacker - This machine performs the MITM attack against the client to sniff the traffic in clear;
3. Monitor - On this machine runs the Monitor for TLS attacks tool.

The figure 6.22 shows an example of the network used for the test.

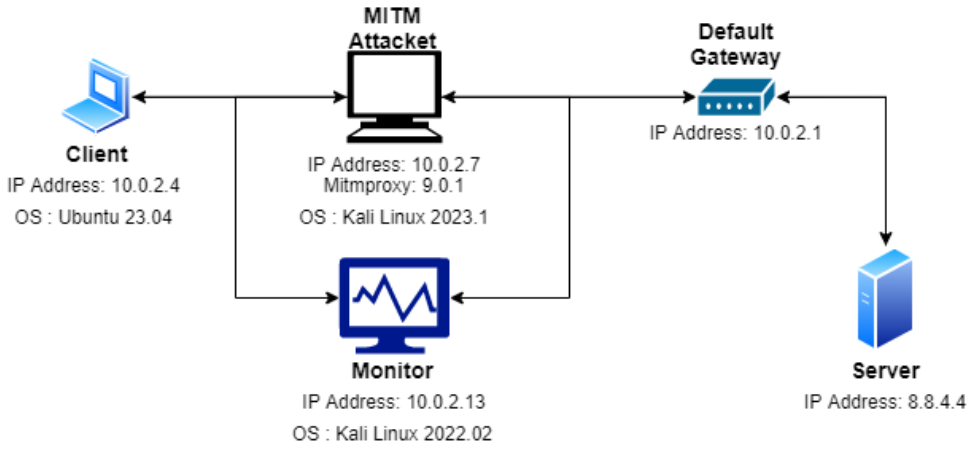


Figure 6.22. Network for MITM test

For this test, we assume that on the client there is the attacker certificate saved on the browser as a trusted certificate and the attacker machine is saved on the client machine as a proxy. In this way, when the client tries to visit the web page, the browser doesn't show any warning.
In this situation, the monitor sniffs the 2 different connection that is established to connect the client to the web page. The first connection is between the client and the attacker which acts like a proxy. The second is between the Attacker and the web page. The figure 6.23 shows how the Monitor for TLS attacks alerts the user for the untrusted certificate of the attacker. In fact, the first connection has no CRL and no OCSP. It is impossible to use also the Certificate Transparency because the certificate has no SCT. Instead, for the second connection, as shown in the figure 6.23 the tool has found the CRL and the OCSP in the certificate. The second certificate has also che SCT in the certificate which is verified.

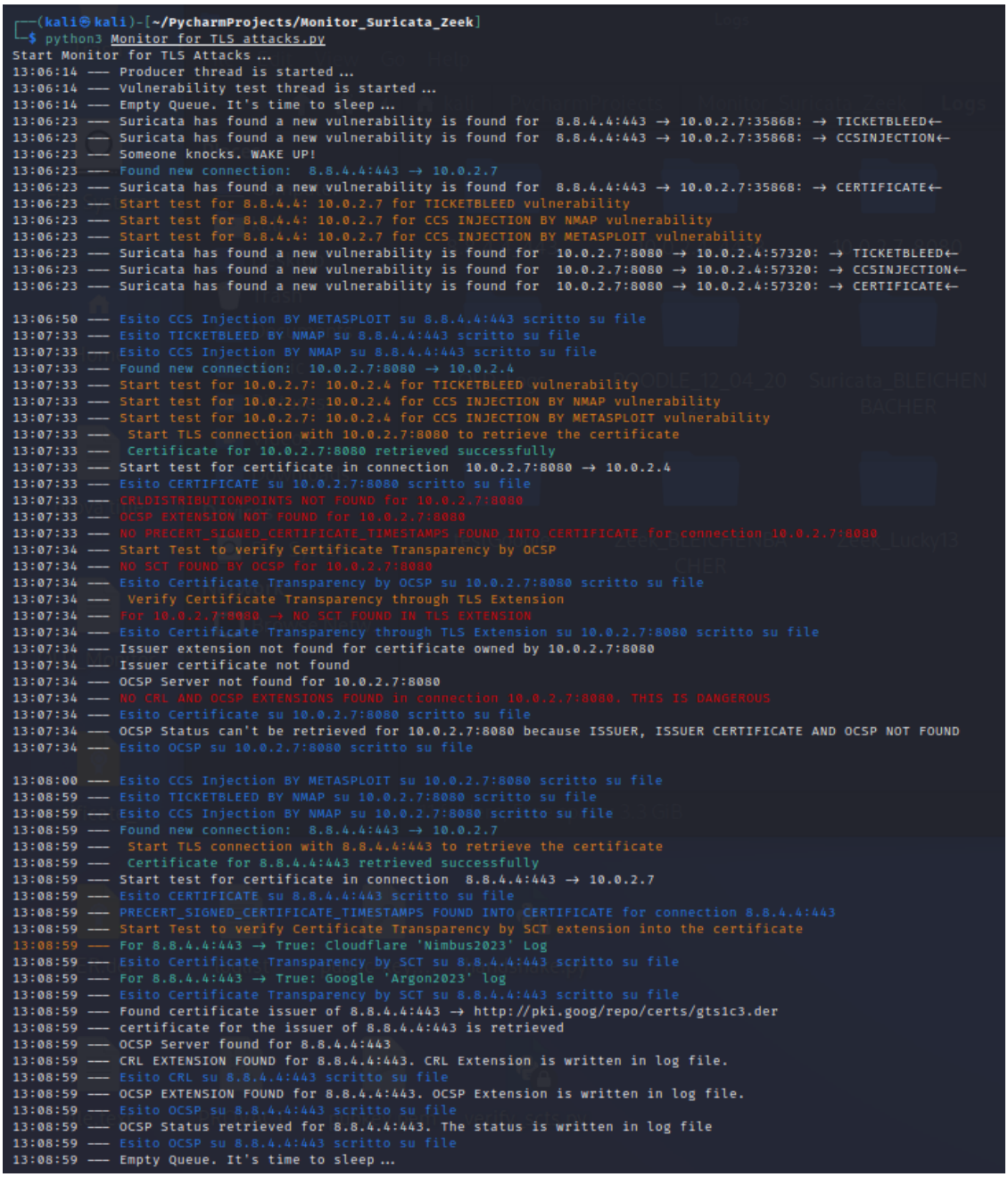


Figure 6.23. Monitor for TLS attacks tool output for MITM test

6.4 Stress-test

To analyze the Monitor for TLS attacks' behavior with more packets a custom script is written as described in section A.7. The script uses the OpenSSL library to establish a TLS connection with the 50th sites most visited in 2023. The script is used with both IDS to analyze also the different IDS behavior with more traffic. The results of this test are shown in figure 6.24. As shown by figure, the tool's behavior is dependent on the IDS used. As shown in the figure 6.24, the tool is faster when it uses Suricata than Zeek.

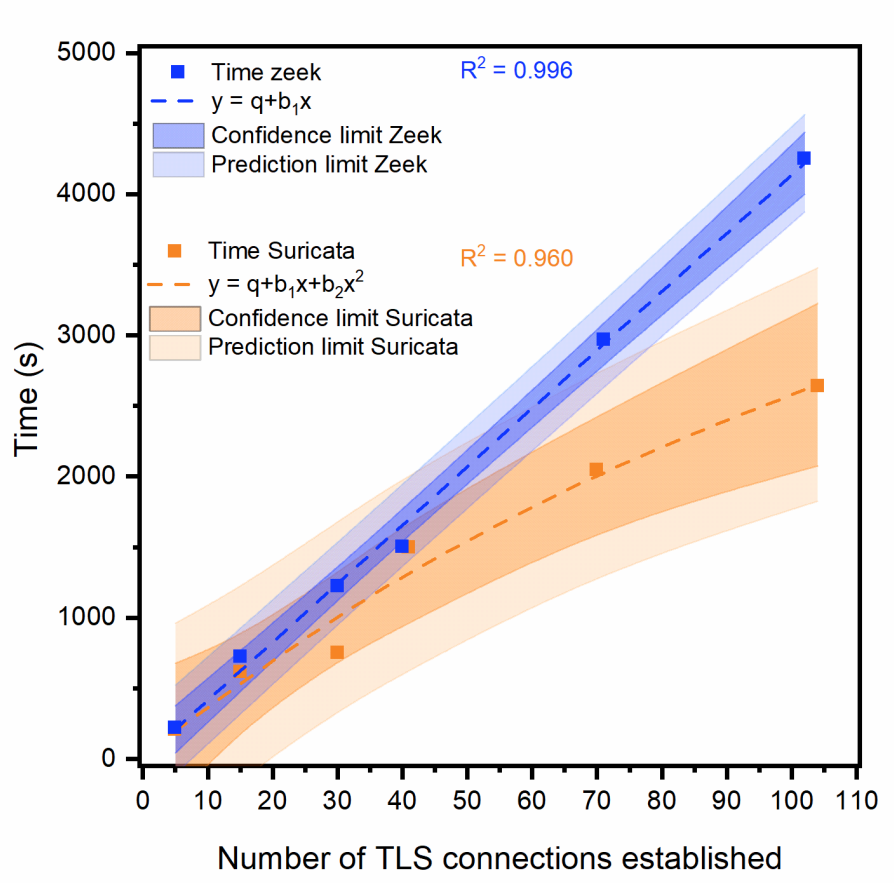


Figure 6.24. Tool's behavior with Suricata and Zeek. This picture represents how many times the Monitor for TLS attacks tool implies verifying different TLS connections. The picture is generated with OriginLab.

By further analysis of the 2 IDS, it is possible to see that Suricata lost some packets in the network. The figure 6.25 shows the difference between sent packets and sniffed packets of the 2 IDS. If an IDS sniffs all the sent packets, the result is shown in the figure by a diagonal because there is a correspondence of 1:1. As shown in figure 6.25, Zeek results are represented by a diagonal, but Suricata's result is under the diagonal. This means that Suricata sniffs fewer packets than the packets sent. This can be possible for an error in custom rules written to sniff specific TLS packets, but further analysis of Suricata shows that in the default Suricata TLS log, the TLS packets are equal to the TLS packets sniffed with the customs rules. This means that the **tls** keyword used in Suricata's rules lost some packets. Instead, the high-level work done by Zeek is more efficient.

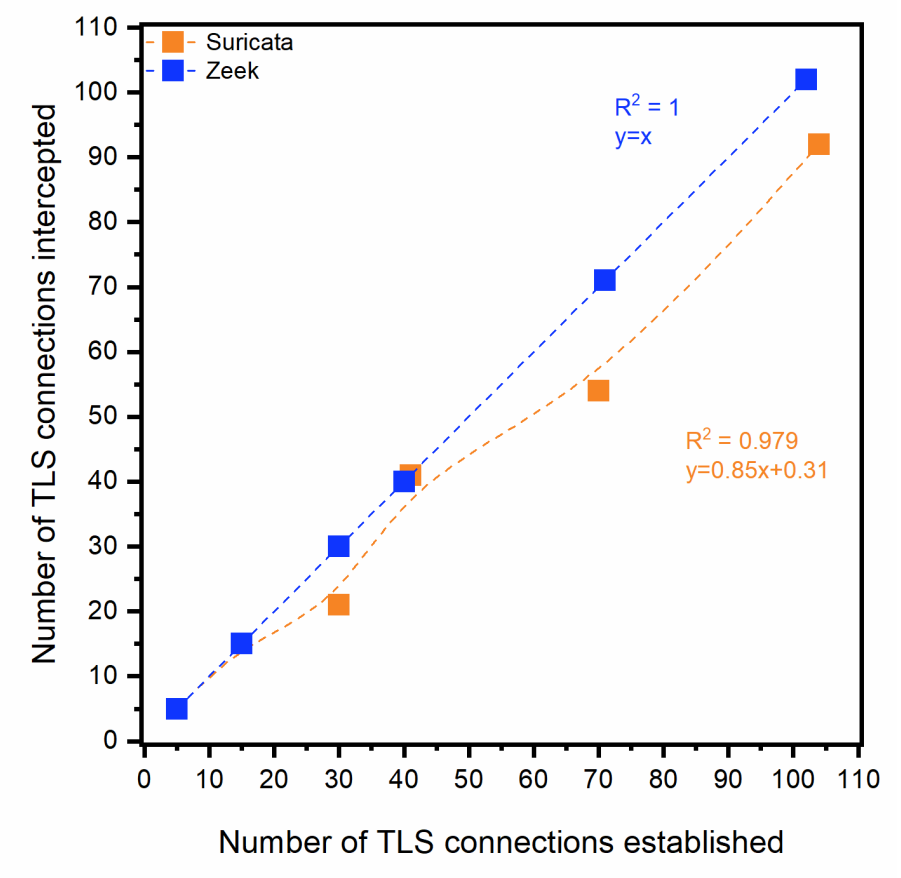


Figure 6.25. In this picture there is the representation of the TLS connections established compared with the TLS connections intercepted by the IDS

Chapter 7

Conclusions

Monitor for TLS attacks is a tool developed to monitor different targets to protect them against TLS attacks. It monitors the target on the network and if intercepts some possible vulnerable TLS connections, it starts the integrated TLS attacks tool to verify if the vulnerability is real or not. Monitor for TLS attacks uses 2 of the most famous IDS, Suricata and Zeek, to intercept the TLS connections and with some of the most famous attacks tool, like Metasploit and Nmap, it verifies the server's vulnerability. The tool must be in the same network of the server that wants to be tested.

As shown in figure 6.16 the attack tools verify quickly the vulnerabilities. This is possible for the multi-threaded structure of the tool. But when the tool intercepts a large amount of TLS connections, it spends a lot of time finishing all the attacks. As shown in figure 6.16, with Zeek the tool describes a linear function because it manages a lot of attack tools that use all the available threads and this causes CPU saturation. Instead, with Suricata that lost some packets, the tool implies less time. This is surely done because the Monitor for TLS attacks tool, with Suricata, must manage fewer attack tools than Zeek. This tool can protect a server in the network in real-time differently from the nowadays solutions, such as SSLLab or TLSAssistant. With it, a network manager can notice configuration changes and respond faster to fix them. The actual version of the tool has fewer old attacks, but in future work, other attacks tool can be easily added, because a strong point of this tool is the facility to add new attack tools to verify different vulnerabilities. As for adding attacks, an advantage of the tool is the fact that other IDS can easily be used by the tool, adding the TLS log file of the newer IDS in the Python script. A tool's disadvantage is that in the actual version, it can not elaborate a lot of TLS connections, but this can be resolved using the tool in a machine with much more CPU. The machine used for the above tests has 4 CPUs and 14GB of RAM and nowadays there are computers with better performance. For a future release, an important point to upgrade can be the markers used in the tool. In the actual version, the markers used are very generic and this leads to starts more TLS attack tools. If more specific markers will be found, the tool will use fewer CPUs than the actual version launching fewer TLS attack tools. Interesting future work can use Machine Learning techniques to intercept also the variant of today's known TLS attacks. Another interesting work can be running this tool on a real server machine, with more CPUs core to see how the tool behaves.

Appendix A

User's Manual

A.1 Monitor for TLS attacks

A.1.1 Requirements

To install and execute Monitor for TLS Attacks, the following tools are required:

- Metasploit
- Nmap
- TLSAttacker
- TestSSL

Monitor for TLS Attacks is written in Python and uses some python packages:

- crl-checker
- cryptography
- ocsp-checker
- ctutlz
- watchdog
- subprocess

A.1.2 Installation

To install all these packages it is possible to use *pip*, a python package monitor. To install *pip* and all these packages it is possible to use the INSTALL.sh file on the GitHub repository. It is possible to install Git with the following command on Ubuntu OS:

```
sudo apt install git-all
```

Then to install Monitor for TLS Attacks tool is necessary to run the following command to clone the Git Repository on your computer:

```
git clone https://github.com/s225904/Monitor_Suricata_Zeek.git
```

Alternatively, it is possible to download the .zip file and extract it on the local computer.

A.1.3 Commands

The tool has many options to customize it for user dependency. To know all the options the user can run the following command into the tool directory:

```
python3 ./monitor_for_tls_attacks -h
```

The command gives a brief description of the tool and then all the options available for the tool. in the figure A.1 is shown the output of the command.

Figure A.1. output of the help option

The tool options are:

- –full - this option allows the tool to use the testssl tool. Testssl is a powerful tool that takes a screenshot of the server configuration at that moment and summarizes the vulnerability of the tested server. This tool is optional because requires a lot of memory and CPU.
- –IDS=Suricata/Zeek - This option allows the tool to set the chosen IDS. The default IDS is Suricata but with this option is possible to use Zeek.
- –json ./pathToConfigJSONFile - This option allows the user to pass a JSON config file to the Monitor tool. This file contains the TLS versions, the ciphersuites and the certificate fingerprint that the user wants to allow on the network . Monitor for TLS attacks does not generate attaks against server with the configuration written in the JSON config file. An example of the JSON configuration file is shown in figure A.2

If the user wants to use the attack inside the tool without monitoring the network, he can use the following command:

```
python3 ./monitor_for_tls_attacks --attack [attack_name] --host [ip:port]
```

The attack_name can be:

- heartbleed
- crime
- drown

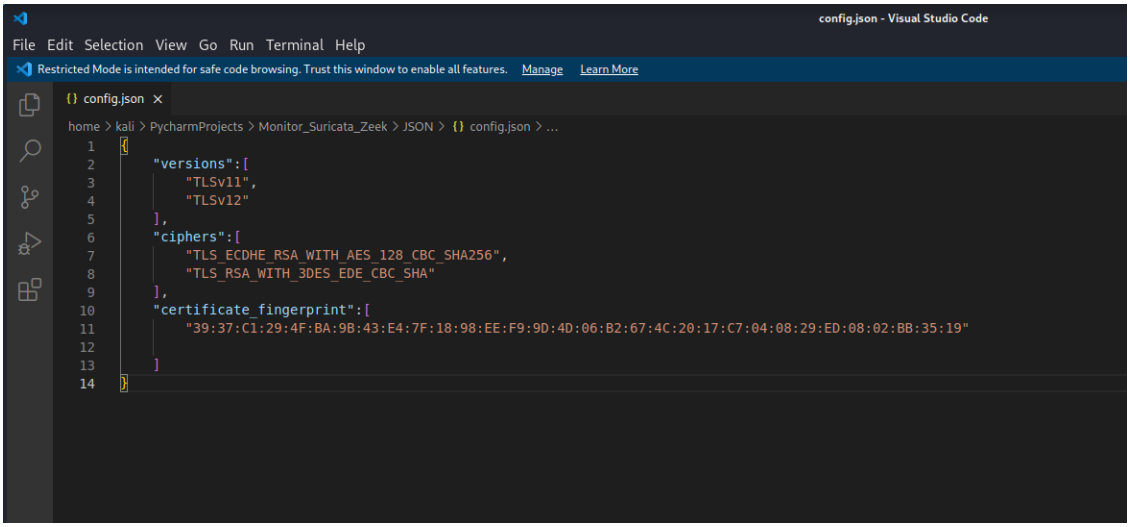


Figure A.2. Example of a config JSON file to pass

- bleichenbacher
- robot
- padding_oracle_attack
- sweet32
- logjam
- lucky13
- poodle
- ticketbleed
- ccs_injection
- roca

A.1.4 How it works

To use this tool the traffic generated from client to server must pass through the machine where this tool runs. This can be possible through a network with the promiscuous mode enabled or through a MITM attack before starts the IDS. There are many tools to start a MITM attack, but for the test, Ettercap is used with the following command:

```
sudo ettercap -T -M arp /IP_CLIENT// /IP_SERVER//
```

It is possible to start the MITM attack with more machines if the user wants to sniff more traffic. After this step, the user must run the IDS that wants to attach to the tool. To run Suricata the command is:

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

In case the user wants use Zeek, the command to run that IDS is:

```
./zeekctl deploy
```

This command must be run from the Zeek directory (in case of default path the directory is on `/usr/local/zeek/bin/`). In case the user wants to run Zeek a step more is necessary. The user have to run the following command from `/usr/local/zeek/logs/`:

```
sudo chmod 700 ./current
```

This command is necessary because every time zeek starts, it creates the `current` directory. This directory and the log files inside are used by Monitor for TLS attacks tool to read the network traffic and start the attack tools. Unfortunately, this directory has the read permission set only for admin, and the command above gives the read permission also to a normal user.
After starting the IDS the Monitor for TLS attacks can start. To run the tool is necessary to go into the tool directory and run the following command with the options that the user wants:

```
python3 ./monitor_for_tls_attacks
```

The tool starts and generates for each connection sniffed from the IDS a log file into the `Logs` directory. Each log file has the name of the IP address and port tested with the timestamp that represents when the test is started. It is possible to see the traffic intercepted from the IDS also into their log directory:

- Default log directory for Suricata: `/var/log/suricata/tls.log`
- Default log directory for Zeek: `/usr/local/zeek/logs/current/`

As mentioned in Section 6.1.1 this tool is designed based on the producer and consumer pattern. The producer is represented by a Watchdog, which is a thread that waits for modification on the IDS log file. The consumer is represented by a thread that waits until the producer stores a potential vulnerability into an associative array. The figure A.3 shows when the tool is started and the 2 threads wait until the IDS generates some alerts in the log file.

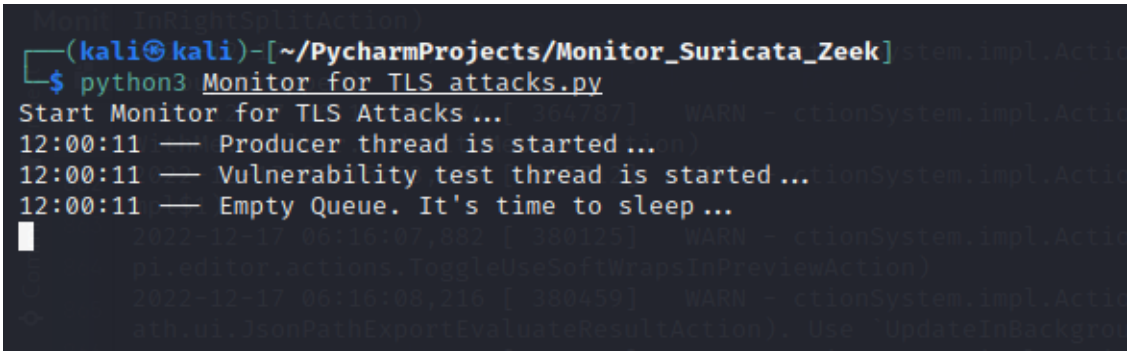


Figure A.3. Start tool

When the IDS has found some potentially vulnerable traffic through the rule, it generates alerts in the log file and this starts the producer thread because the log file has been modified. Optionally, the producer checks if the the cipher suite and the TLS versions are the same of the JSON config file if it is passed to the tool. If not the producer starts with string manipulation and produces in the associative array the connection and pontentially vulnerability that will be tested. Once the associative array is no longer empty, the consumer thread wakes up and starts taking the connection and the vulnerability from the array. The consumer thread tests all the found vulnerabilities by starting the corresponding attacks and writing the result in a log file in the `Log` directory. An example of the tool running is shown in figure A.4.

If the producer does not find new potential vulnerabilities and the associative array is empty, the consumer thread goes to sleep until a new vulnerability has been added to the thread. To give the user an immediate visual impact the tool uses different colors depending on the risk of the alert.

- Yellow - This color is used to informs the user that an attack is started;
- Blue - This color informs the user that an attack started is finished and the result has been writted in the log file;
- Red - This color is used to high risk alerts, for example in case of self signed certificate.

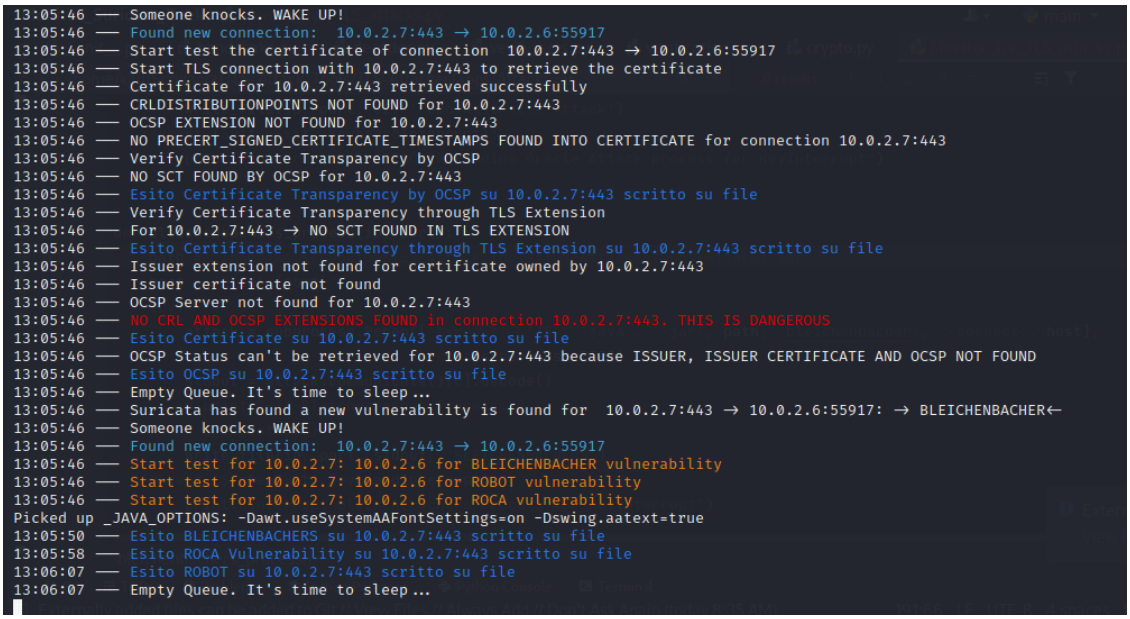


Figure A.4. Tool Running

A.2 Testbed installation

A.2.1 Openssl

Before installing a specific version of the Openssl library, the user must run the following commands to install the requirements for Openssl.

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

To download a specific version of the Openssl library, the user can go on https://www.openssl.org/source/old/ or run the following command:

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

This command downloads the 1.1.1s version of the library, but by changing the version it is possible to download any version of the OpenSSL library. To extract the library from the zip file the user can use the following command:

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

Finally, it is possible to install the downloaded library version run 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 version downloaded the installation of the library can be done by running one of these commands:

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

At this point, the downloaded library has been installed and the following steps are necessary to use from the command line the installed version. The user must write **/usr/local/ssl/lib** in the configuration file, for example with:

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

It is possible to reload the configuration file with:

```
sudo ldconfig -v
```

Then, it is possible to change the environment variable by appending the **:/usr/local/ssl/bin** string to the **/etc/environment** and reload the file by the command:

```
source /etc/environment
```

Finally, it is possible to know what version of the OpenSSL library is used by the command:

```
openssl version -a
```

A.2.2 Apache2

To use Apache2 the following packages are required:

1. libpcre3-dev l
2. ibaprutil1
3. libaprutil1-dev l
4. ibapr1
5. libapr1-dev

It is possible to install them with the following commands:

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

It is possible to download the apache2 version through the URL https://archive.apache.org/dist/httpd/ or running the command:

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

and extract the downloaded version with the command:

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

Then it is possible to install Apache2 version with the commands:

```
-sudo ./configure --enable-ssl --with-ssl=/usr/local/ssl/
    --prefix=/usr/local/apache2/
-sudo make
-sudo make install_sw
```

At this point, the downloaded version is installed and SSL mode can be enabled by uncomment in the file **/usr/local/apache2/conf/httpd.conf** these 2 lines with:

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

A.2.3 Certification Transparency test

This test is done to verify the truthfulness of the certificate verification. In a local network with the following 3 machines:

1. Monitor - The machine where the Monitor for TLS attacks tool runs;
2. Client - The machine that wants to visit a server's web page;
3. Server - The machine visited for a specific web page.

On this network, the client visits the server's web page. In this test the client uses the following command:

```
openssl s_client -connect 8.8.8.8:443
```

But every command that established a TLS connection can be used.

A.2.4 MITM tesst

For this test is required **mitmproxy** tool on the attack machine. If the machine hasn't this tool, it is possible to install **mitmproxy** on Ubuntu through the following command:

```
sudo apt install mitmproxy -y
```

If the attacker machine uses Kali, mitmproxy is already installed. Once The tool is installed, it can start through the following command on the command line:

```
mitmproxy
```

Once mitmproxy start, the attacker machine can be used like a proxy. If the attacker can access the client's machine, he can set the attacker's machine like a proxy on the network settings, like figure A.5.

To hide any warning due to the untrusted certificate of the mitmproxy, the attacker can visit the `mitm.it` and download the mitm certificate. Once downloaded he can add this certificate to the browser's trusted certificate, as shown in the figure A.6

Once added, the client will not notice any alert from the browser, but all the traffic from his machine passes through the attacker's machine. It is possible to see that on the attacker's machine by mitmproxy or through the Monitor for TLS attacks tool on the monitor's machine. For this test, the following command is run from the client's command line:

```
openssl s\_client -connect 8.8.4.4:443
```

The result is shown in figure 6.23.

A.2.5 Configuration file test

As mentioned in section A.1.2, a JSON file can be passed as an argument to the tool, through –**json [Path_to_JSON_file]**. An example of this file is shown in figure A.2. The server's certificate fingerprint which will not verify by the tool, can be retrieved by the command:

```
openssl x509 -fingerprint -noout -in <certificate\_to\_test>
```

Once retrieved, it will be added in the JSON file in the **certificate_fingerprint** section. Once added, from the client machine the following command can be run:

```
openssl s_client -connect 10.0.2.15:4433 -cipher DES-CBC3-SHA -tls1
openssl s_client -connect 10.0.2.15:4433 -cipher DES-CBC3-SHA -tls1_1
```

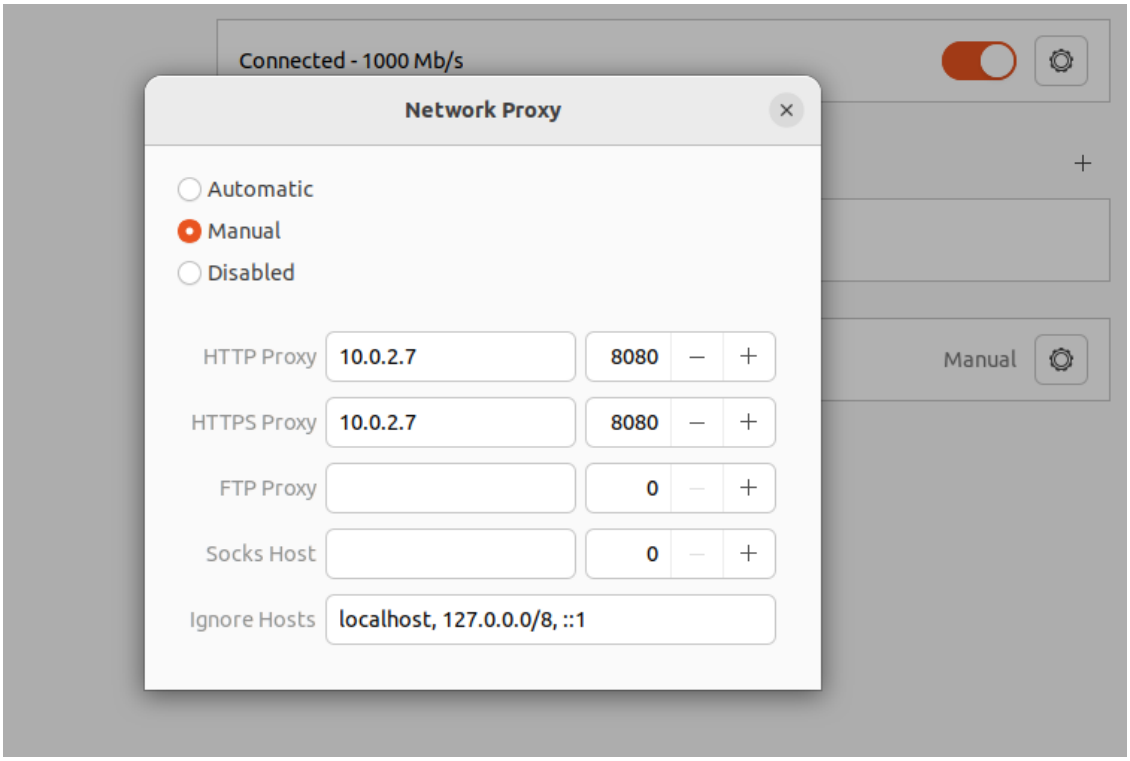


Figure A.5. Setting attacker's machine like proxy

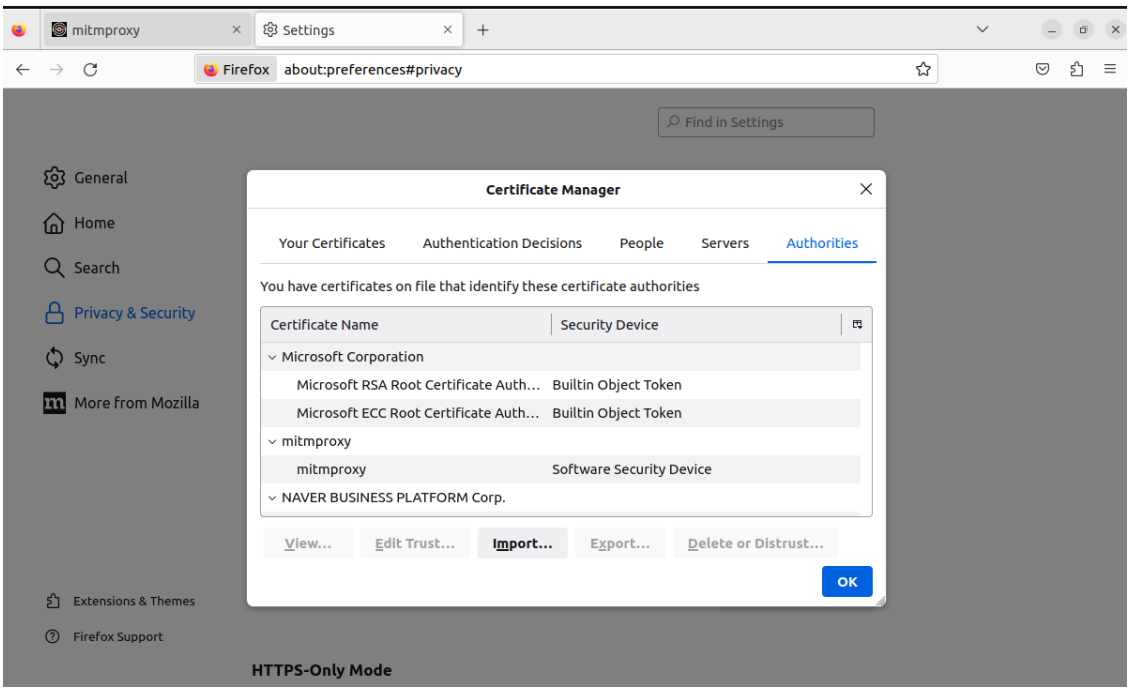


Figure A.6. Add mitm certificate to the browser's Authorities certificate

These commands start a TLS handshake with the server. Both use the same cipher suite **DES-CBC3-SHA**, which is present in the JSON file. In fact, no one cipher suite attack starts against the server although the DES and CBC are used. The first command uses TLSv1 which is not present in the JSON file and this will start attacks against the server for the TICKETBLEED

and CCS_INJECTION attacks. The second command uses the TLSv1_1 version, which is present in the JSON file and this disables the attack for the possible vulnerability that this version can have. The result of this test is shown in figure 6.17.

A.2.6 Measure Monitor for TLS attacks for single vulnerability

For this test, the Monitor for TLS attacks tool was modified to run only the tools for one specific vulnerability. The line of code for the other attacks was commented on. Only the line of codes for the specific vulnerability installed on the vulnerable server was uncommented. This is done to measure how many times the tool implies verifying the vulnerability with all the attacks. For this test, the OpenSSL vulnerable library and Apache2 were installed on the client machine as described in section A.2. After this installation, before starting the tool, on the monitor machine ettercap was started to sniff all the packets between the client and the server machine with the following command:

```
sudo ettercap -T -M arp /192.168.0.118// /192.168.0.121//
```

Then, the IDS was started with the following command:

```
sudo systemctl start suricata
sudo /usr/local/zeek/bin/zeekctl deploy
```

And finally, the tools can be started:

```
time sudo python3 ./Monitor_for_TLS_attacks.py
```

The command **time** was used to calculate how much time the program takes. Once the tool was started, from the client machine the TLS connection was initialized with wget or OpenSSL commands like:

```
wget https://192.168.0.121:443
openssl s_client -connect 192.168.0.121:443
```

For vulnerability that requires a specific cipher suite or TLS version, such as Bleichenbacher, Ticketbleed or LogJam, it is used the following OpenSSL commands:

```
openssl s_client -connect 192.168.0.121:443 -cipher DES-CBC3-SHA
openssl s_client -connect 192.168.0.121:443 -tls1
openssl s_client -connect 192.168.0.121:443 -cipher EXP-DES-CBC-SHA -tls1
```

Once the tool had finished the analysis, the tool was stopped with the CTRL+C command on the command line that was used to raise the Keyboard Interrupt. This interrupt will end the tool. Once the tool ended, before the next measure, the IDS log file must be cleaned. In this way, when the tool starts, it will start the verification only with the new sniffed packets. Zeek cleans its log file when it ends. The following commands end Zeek IDS:

```
sudo /usr/local/zeek/bin/zeekctl stop
```

Suricata cleans its log file when it starts because the Suricata yaml file does not open the Suricata log file in append mode. For each test suricata IDS was restarted with the command:

```
sudo systemctl restart suricata
```

For each vulnerability, 20 packets were sent. The result of this test is shown in figure 6.15.

A.2.7 Stress test for Monitor for TLS attacks tool

This test is done to evaluate how many times the tool takes to verify more packets. For this test, a custom Python script is done to start more TLS connections. The figure A.7 shows the custom script. This script uses a list of the most 50 visited websites in 2023.

```
if __name__ == '__main__':
    num =0
    for i in range(0,50):
        time.sleep(2)
        num=num+1
        print(f'Openssl: {siti[i]} -----------> {num}')
        print(f"timeout 3 openssl s_client -connect {siti[i]}:443")
        prova =os.system(f"timeout 3 openssl s_client -connect {siti[i]}:443")
```

Figure A.7. Custom script to start more TLS connections

This script is used in the network shown in figure A.2.6.

Appendix B

Developer's Reference Guide

B.0.1 Suricata

Suricata configuration file in:

`/etc/suricata/suricata.yaml`

Suricata log file in:

`/var/log/suricata/mio_fast.log`

Suricata's rules file:

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

B.0.2 Zeek

Zeek configuration file in:

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

Zeek ssl log file in:

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

Zeek's SSL rules file:

`/usr/local/zeek/share/zeek/base/protocols/ssl/mail.zeek`

B.0.3 ctutlz

This library is used to verify the Certificate Transparency. To use this library for the tool, the following functions must be edited in `/usr/local/lib/python3.10/dist-packages/ctutlz/sct/verification.py`:

1. find_log - This function is used to find the id log in the sct. The modified function is shown in figure B.1;
2. verify_signature - This function verifies it signature of the **signature_input** was created using **digest_algo** by the private key of the **pubkey_pem**. The modified function is shown in figure B.2;
3. verify_sct - This function is used to verify the single sct found. The modified function is shown in figure B.3.

and in file `/usr/local/lib/python3.10/dist-packages/ctutlz/tls/handshake.py` must edit the **do_handshake** function with the following **addr** variable:

```
addr=(domain,int(port))
```

```python
def find_log(sct, logs):

    if type(logs)==dict:
        for log in logs["operators"]:
            logs2= log["logs"]
            for log2 in logs2:
                if log2["log_id"]== sct.log_id_b64:
                    return log2

    return None
```

Figure B.1. find_log function

```python
def verify_signature(signature_input, signature,
                     pubkey_pem, digest_algo='sha256'):

    pub=f'''-----BEGIN PUBLIC KEY-----\n{pubkey_pem.decode()}\n-----END PUBLIC KEY-----\n'''

    cryptography_key = serialization.load_pem_public_key(pub.encode('ascii'),backend)
    pkey = pkey_from_cryptography_key(cryptography_key)

    auxiliary_cert = X509()
    auxiliary_cert.set_pubkey(pkey)

    try:
        verify(cert=auxiliary_cert, signature=signature,
               data=signature_input, digest=digest_algo)
    except OpenSSL_crypto_Error:
        return False

    return True
```

Figure B.2. verify_signature function

To change the Certificate Transparency log list the following file must be edit:

`/usr/local/lib/python3.10/dist-packages/ctutlz/ctlog.py`

```
def verify_sct(ee_cert, sct, logs,
               issuer_cert, more_issuer_cert_candidates,
               sign_input_func):
    log = find_log(sct, logs)
    if log:
        verified = verify_signature(
            signature_input=sign_input_func(ee_cert, sct, issuer_cert),
            signature=sct.signature,
            pubkey_pem=log["key"].encode('ascii')
        )

        if not verified and more_issuer_cert_candidates is not None:

            for issuer_cert in more_issuer_cert_candidates:
                verified = verify_signature(
                    signature_input=sign_input_func(ee_cert, sct, issuer_cert),
                    signature=sct.signature,
                    pubkey_pem=log["key"].encode('ascii')
                )
                if verified:
                    break

        return SctVerificationResult(ee_cert, sct, log, verified)
    return SctVerificationResult(ee_cert, sct, log=None, verified=False)
```

Figure B.3. verify_sct function

B.0.4 Monitor for TLS attacks tool

Starting from the source directory of the tool, it is possible to find the arrays' cipher suite in the file:

`./ciphers.py`

The tool's log file in:

`./Logs/<IPServer:IPPort>`

Bibliography

[1] C. Simoiu, W. Nguyen, Z. Durumeric "An Empirical Analysis of {HTTPS} Configuration Security", arXiv, Vol. 1, No. 1, Nov 2021, DOI 10.48550/arXiv.2111.00703

[2] D.G. Berbecaru, A. Lioy, and C. Cameroni, "Providing Login and Wi-Fi Access Services with the eIDAS Network: A Practical Approach," IEEE Access, 2020, Vol. 8, pp. 126186-126200, DOI 10.1109/ACCESS.2020.3007998

[3] D. Berbecaru, A. Lioy, and C. Cameroni, "Supporting Authorize-then-Authenticate for Wi-Fi access based on an electronic identity infrastructure," Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 2020, 11 (2), pp. 34-54, DOI 10.22667/JOWUA.2020.06.30.034

[4] D. Berbecaru, A. Atzeni, M. De Benedictis and P. Smiraglia, "Towards Stronger Data Security in an eID Management Infrastructure" 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), St. Petersburg, Russia, 2017, pp. 391-395, DOI 10.1109/PDP.2017.90

[5] D. Berbecaru, A. Lioy, (2007). "On the Robustness of Applications Based on the SSL and TLS Security Protocols", In J. Lopez ,P.Samarati, J.L. Ferrer, (eds) in the boook Public Key Infrastructure. Euro PKI 2007. Lecture Notes in Computer Science, vol 4582. Springer, Berlin, Heidelberg. DOI https://doi.org/10.1007/978-3-540-73408-6_18

[6] Qualys' SSL Server Test, https://www.ssllabs.com/, last visit 11-06-2023

[7] S. Manfredi,S. Ranise,G. Sciarretta, "Lost in TLS? No More! Assisted Deployment of Secure TLS Configurations" in the book "Data and Applications Security and Privacy XXXIII" Springer International Publishing., 2019, pp. 201-220, DOI 10.1007/978-3-030-22479-0_11

[8] S. Turner, T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC-6176, 1995, DOI 10.17487/RFC6176

[9] A. Freier, P. Karlton, P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", RFC-6101, 1996, DOI 10.17487/RFC6101

[10] T.Dierks, C. Allen, "The TLS Protocol Version 1.0", RFC-2246, January 1999, DOI 10.17487/RFC2246

[11] T.Dierks, E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC-4346, April 2006, DOI 10.17487/RFC4346

[12] T.Dierks, E.Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC-5246, August 2008, DOI 10.17487/RFC5246

[13] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3", RFC-8446, August 2018, DOI 10.17487/RFC8446

[14] A. Ghedini, V. Vasilievr, "TLS Certificate Compression", RFC-8879, December 2020, DOI 10.17487/RFC8879

[15] D. Eastlake 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC-6066, January 2011, DOI 10.17487/RFC6066

[16] N. Sullivan, "Exported Authenticators in TLS", https://datatracker.ietf.org/doc/draft-ietf-tls-exported-authenticator/15/

[17] W. Stallings, "Cryptography and Network Security - Principles and Practice", Seventh edition, Pearson, 2018, ISBN: 978-93-325-8522-5

[18] V. Chinnasamy, "Enabling TLS 1.3 Certificate - Are You Ready for Moving Forward" https://www.indusface.com/blog/enabling-tls-1-3-certificate-are-you-ready-for-moving-forward/, last visit 23-06-2023

[19] R. Shirey, "Internet Security Glossary, Version 2", RFC-4949, August 2007, DOI 10.17487/RFC4949

[20] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, R. Nicholas, "Internet X.509 Public Key Infrastructure: Certification Path Building", RFC-4158, September 2005, DOI 10.17487/RFC4158

[21] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC-5280, May 2008, DOI 10.17487/RFC5280

[22] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC-6960, June 2013, DOI 10.17487/RFC6960

[23] Berbecaru, D., Casalino, M.M. and Lioy, A. (2013), "FcgiOCSP: a scalable OCSP-based certificate validation system exploiting the FastCGI interface", in the book Softw. Pract. Exper., Vol. 43, pp. 1489-1518. DOI https://doi.org/10.1002/spe.2148

[24] B. Laurie, E. Messeri, R. Stradling, "Certificate Transparency Version 2.0", December 2021, DOI 10.17487/RFC9162

[25] 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

[26] C. Boyd, A. Mathuria,D. Stebila, "Attacks Overview", in the book "Protocols for Authentication and Key Establishment", Springer Berlin, Heidelberg, 2019, pp. 283-314, DOI https://doi.org/10.1007/978-3-662-58146-9

[27] 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

[28] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, Y. Yarom, Yuval, "The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations", in the book "2019 IEEE Symposium on Security and Privacy (SP)" 2019, pp. 435-452, DOI 10.1109/SP.2019.00062

[29] 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

[30] J. Rizzo, T. Duong, "Practical Padding Oracle Attacks", Proceedings of the 4th USENIX Conference on Offensive Technologies, Washington, DC, May 25 2010, pp. 1-8 https://www.usenix.org/legacy/event/woot10/tech/full_papers/Rizzo.pdf

[31] CVE MITRE, CVE-2014-3566, https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-3566

[32] E. S. Alashwali and K. Rasmussen, "What's in a Downgrade? A Taxonomy of Downgrade Attacks in the TLS Protocol and Application Protocols Using TLS", in the book CoRR, Vol. abs/1809.05681, 2018, DOI https://doi.org/10.48550/arXiv.1809.05681

[33] 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

[34] N. Sullivan, "A Detailed Look at RFC 8446 (a.k.a. TLS 1.3)", CloudFlare, https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/ last visit 23-06-2023

[35] CVE MITRE, CVE-2014-8730, https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-8730

[36] 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

[37] D. G. Berbecaru, A. Lioy, "Attack Strategies and Countermeasures in Transport-Based Time Synchronization Solutions" In: Camacho, D., Rosaci, D., Sarné, G.M.L., Versaci, M. (eds) 2022 Intelligent Distributed Computing XIV. IDC 2021. Studies in Computational Intelligence, vol 1026. Springer, Cham. DOI https://doi.org/10.1007/978-3-030-96627-0_19

[38] NIST, CVE-2012-4929 https://nvd.nist.gov/vuln/detail/CVE-2012-4929

[39] G. Gincy, "Importance of TLS 1.3: SSL and TLS Vulnerabilities", https://beaglesecurity.com/blog/article/importance-of-tls-1-3-ssl-and-tls-vulnerabilities.html last visit 23-06-2023
[40] M. Vanhoef, T. V. Goethem, "HEIST: HTTP Encrypted Information can be Stolen through TCP-windows", Black Hat US Briefings, Las Vegas, USA, August 2016, https://www.blackhat.com/docs/us-16/materials/us-16-VanGoethem-HEIST-HTTP-Encrypted-Information-Can-Be-Stolen-Through-TCP-Windows-wp.pdf
[41] N. Drucker and S. Gueron, "Selfie: reflections on TLS 1.3 with PSK", in the book J Cryptol, Vol. 34, No. 27, 25 May 2021, DOI 10.1007/s00145-021-09387-y
[42] NIST, CVE-2014-0160 https://nvd.nist.gov/vuln/detail/CVE-2014-0160#vulnCurrentDescriptionTitle
[43] 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/ last visit 23-06-2023
[44] NIST, CVE-2014-0224, https://nvd.nist.gov/vuln/detail/cve-2014-0224
[45] CVE MITRE, CVE-2011-3389 https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2011-3389
[46] A. Khraisat, I. Gondal,P. Vamplew et al. "Survey of intrusion detection systems: techniques, datasets and challenges." Cybersecur 2, 20, 2019, DOI https://doi.org/10.1186/s42400-019-0038-7
[47] R. Singh,H. Kumar, R. K. Singla, R.R. Ketti, "Internet attacks and intrusion detection system: A review of the literature", Online Information Review, Vol. 41 No. 2, pp. 171-184. (2017), DOI https://doi.org/10.1108/OIR-12-2015-0394
[48] Snort Web Page, https://www.snort.org/, last visit 11-06-2023
[49] Suricata Web Page, https://suricata.io/ last visit 11-06-2023
[50] Zeek Web Page, https://zeek.org/ last visit 11-06-2023
[51] TLSAssistant v2, GitHub, https://github.com/stfbk/tlsassistant, last visit 11-06-2023
[52] W. Park, S. Ahn, "Performance Comparison and Detection Analysis in Snort and Suricata Environment", in the book Wireless Pers Commun, Vol. 94, pp. 241-252 (2017), DOI https://doi.org/10.1007
[53] H. Alyami,MTJ Ansari, A. Alharbi, W. Alosaimi, M. Alshammari, D. Pandey, A. Agrawal, R. Kumar, R.A. Khan, "Effectiveness Evaluation of Different IDSs Using Integrated Fuzzy MCDM Mode", in book Electronics, Vol. 11, n. 6, article n. 859, Mar 2022, ISSN 2079-9292, https://www.mdpi.com/2079-9292/11/6/859 DOI 10.3390/electronics11060859
[54] Suricata User Guide, https://suricata.readthedocs.io/en/latest/index.html last visit 11-06-2023
[55] Suricata Web Page, https://suricata.io/download/ last visit 11-06-2023
[56] Zeek Download Page, https://zeek.org/get-zeek/ last visit 11-06-2023
[57] Nmap Web Page, https://nmap.org/ last visit 11-06-2023
[58] Metasploit Web Page, https://www.metasploit.com/ last visit 11-06-2023
[59] Ettercap Web Page, https://www.ettercap-project.org/ last visit 11-06-2023
[60] 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. DOI https://doi.org/10.1145/2976749.2978411
[61] TestSSL Web Page, https://testssl.sh/
[62] OpenSSL's vulnerabilities page, https://www.openssl.org/news/vulnerabilities.html
[63] CVE-2012-0884, https://www.openssl.org/news/secadv/20120312.txt
[64] CVE MITRE, CVE-2014-0224, https://www.cve.org/CVERecord?id=CVE-2014-0224
[65] CVE MITRE, CVE-2013-0169, https://www.cve.org/CVERecord?id=CVE-2013-0169
[66] CVE MITRE, CVE-2016-2183, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2183
[67] CVE MITRE, CVE-2016-2107, https://www.cve.org/CVERecord?id=CVE-2016-2107
[68] CVE MITRE, CVE-2017-15361, https://www.cve.org/CVERecord?id=CVE-2017-15361
[69] CVE MITRE, CVE-2016-9244, https://www.cve.org/CVERecord?id=CVE-2016-9244

[70] CVE MITRE, CVE-2016-0800, https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0800