HTTPS connection and DoS attack

Course of Network Security

Authors: Matteo Cossu, Id: 70/90/00657

Doris Dearca, Id: 70/90/00666

Chiara Matta, Id: 70/91/00161

Abstract

In all communications it is important to ensure security in the exchange of informations between the client and the server. The aim of our project is to build a secure connection and to explore the vulnerability during communications, highlighting the potential for a malicious client, performed by an attacker, to launch an active attack. The server will be designed to block this malicious connection. To illustrate this vulnerability, this project will establish an HTTPS session between a client and a server starting from an HTTP base, during which a DoS (Denial of Service) attack, from a malicious client, will be simulated. The server will block the connection with the hacker. Specifically, we introduce a DoS attack that makes a service or network inaccessible to legitimate users by overloading it or exploiting its vulnerabilities. This penetration is recognized as an active attack that can be detected but not prevented. We don't use SSL libraries to build HTTP and HTTPS, in fact we manually write the code to implement the two protocols. This simulation mainly underscores the importance of robust security mechanisms and illustrates the potential risks associated with insecure communication protocols.

An example of the malicious scenario includes the use of a home smart camera. The camera is accessible via the manufacturer's app and allows real-time video streaming. Here's how it works: the camera opens an HTTP connection on a standard port (e.g., 8080); the user's mobile app connects to this port to view the video stream. No secure protocol is implemented (e.g., HTTPS or end-to-end encryption).

The malicious scenario (direct attack on port 8080):
Port 8080 is open and publicly accessible because the manufacturer hasn't configured proper restrictions or required the use of a firewall. An attacker could send malformed requests to the port to disrupt the streaming with a DoS (Denial of Service) attack. The attacker could identify the camera by scanning the local network using tools to find devices responding on port 8080. Then, send direct HTTP requests to the device to access the video feed or extract sensitive information. At the end, use known exploits for the camera's vulnerable firmware to gain full control of the device. The consequences could be that the attacker could spy on Anna and obtain personal information (e.g., knowing when the house is empty). If the device is part of a business network (e.g., a security camera in an office), the attacker could use it as an entry point to penetrate the main network.

How to prevent this scenario? The camera should be implemented by HTTPS on port 443 to encrypt data.

Keywords

HTTP, HTTPS, AES, RSA, TLS, TCP 3-way handshake, certificate check, DoS, active attack.

Introduction

Nowadays it is relevant to improve security of Internet communication and engineers still work to enhance more security in it. HTTP is a stateless protocol that provides a standard communication to simplify exchanges among client and server, without adopting any security mechanisms and services. The fact that it doesn't guarantee any protection from any possible attacks, either active or passive, represents problems also with the rapid development of web applications, web sites and IoT devices. However, an HTTP connection can be upgraded into an HTTPS one, in which the last "s" stands for "security". HTTPS implements TLS, or SSL (previous version of TLS), to encrypt HTTP requests and responses, making it resistant to most common attacks. To build HTTPS and HTTP architectures we don't use SSL libraries, but we manually write the entire code.

The figure 1 shows a report on statistical percentages regarding the implementation of HTTPS in italian public administration systems, conducted by CERT-AGID (Computer Emergency Response Team). The data reported are grouped into four categories: (1) sites without the implementation of the HTTPS protocol, (2) sites with significant security issues, (3) sites with configurations no longer aligned with modern standards and (4) sites with correctly configured HTTPS protocol. It's possible to see a duplication of sites that implement HTTPS between 2021 and 2022. Additionally, this value results quadruplicate in comparison to the year 2020. This trend is also attributed to automatic redirects from HTTP to ensure a more secure connection. [1]

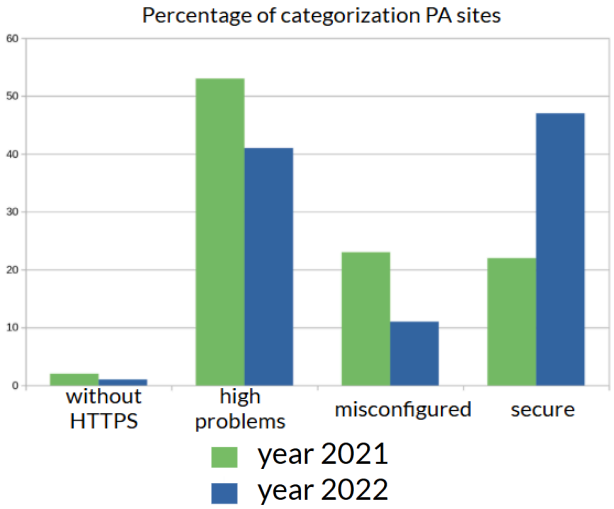

Figure 1: Statistical percentages of HTTPS implementations in italian public administration by CERT-AGID

Materials and methods

Main concepts

HTTPS, or HyperText Transfer Protocol Secure, is the secure version of HTTP. It uses encryption via SSL/TLS to protect the data exchanged between the browser and the server, ensuring privacy and data integrity during online communication. About the choice of the ports, the code cannot be run in online platforms, such as Colab, because these tools use port 80 to run.

A TCP server listens to and accepts client connections over the Transmission Control Protocol (TCP). It ensures reliable, ordered and error-checked delivery of data between the server and clients, commonly used in applications like web servers, email, and file transfer.

A socket is an endpoint for communication between two devices over a network. On the server side, a socket listens for incoming client connections and establishes communication once a connection is made. On the client side, a socket initiates the connection to a server and exchanges data once connected. Sockets are essential for creating networked applications using protocols like TCP (this case) or UDP.

The three-way handshake is a process used in the TCP/IP protocol to establish a reliable connection between a client and a server. It ensures both parties are ready to communicate and can synchronize their data transmission. This handshake is crucial for ensuring the integrity and reliability of data exchange.

Using the Socket library, which provides a direct way to establish connection, we simulate the desired communication process, including performing the 3-way handshake and the certificate handshake. The decision to use this specific library was made to enable simulations of various handshakes and data exchange in a reliable and straightforward manner. With the Socket library, first the connection is established and the desired communication is simulated afterward. There are also more primitive libraries, such as Scapy, that allows the manually creation of raw packets (including the IP, payload, ..) but it may lead to inference issues with other web applications and it results unsuitable for real-time applications that require low latency and with Scapy no connection is established. However, in both cases the final result remains consistent.

The Transport Layer Security (TLS) is used to protect communications between clients and servers over the Internet, ensuring confidentiality (defining a secret key after a handshake), authentication (with an asymmetric encryption) and integrity (including a message integrity check based on MAC in the transport layer). Over time, the security protocol TLS was introduced to replace SSL (Secure Socket Layer), which had shown increasing vulnerabilities and limitations, providing significant advancements in security and performance. It's evident that TLS is an evolution of the outdated SSL. Specifically, TLS uses advanced encryption algorithms, such as AES (Advanced Encryption Standard), to ensure confidentiality and data integrity. Moreover, during this connection establishment, a negotiation process known as handshake is performed with the exchanging of certificates to authenticate the server and a selection of appropriate encryption algorithms and security parameters for the session.

For encryption, we implemented two algorithms: AES and RSA. AES was used for the session key and the final data exchange, while RSA was employed to securely exchange the AES key. Specifically, the AES key is encrypted using RSA, transmitted between the parties, and then decrypted. Once the key exchange is complete, all further communication uses AES for encryption.

This ensures a secure and efficient exchange process while adhering to best practices in cryptography.

Main libraries for Client and Server

http.client: a Python library for sending HTTP requests from the client side to a server. It allows sending GET, POST, and other HTTP methods.

http.server: a Python library for creating a basic HTTP server. It listens for client requests and server responses, often used for testing or simple hosting.

time: a Python library for handling time-related operations, such as measuring delays or timestamps in communications.

Main libraries for Encryption

rsa: a library that provides tools to implement RSA encryption and decryption, a widely used algorithm for secure data transmission.
hashes: part of the cryptography library, used to generate hash values, ensuring data integrity by creating a fixed-size unique fingerprint of input data.
padding: a library for adding or removing padding to plaintext, ensuring it conforms to the required block size of encryption algorithms.

Main libraries for Certification

Name and NameAttribute: used to define and handle distinguished names in certificates, such as "Common Name" or "Organization".
CertificateBuilder: a tool for creating and customizing X.509 certificates, which are essential for SSL/TLS connections.
NameOID: provides object identifiers (OIDs) for common certificate attributes like country or email.
Datetime: used for setting validity periods of certificates, such as start and expiration dates.

Secure connection process

The process that we implement consists of four parts that establish a secure connection between the client and the server. The first part consists of the TCP 3-way handshake in which three messages are exchanged: (1) the client sends a SYN packet to indicate its desire to start the connection, (2) the server responds with SYN + ACK to propose synchronization (ACK indicates the response of the segment that is received and SYN indicates which SN the segments should start with), (3) the client to confirm synchronization sends an ACK. Now the TCP connection is established.

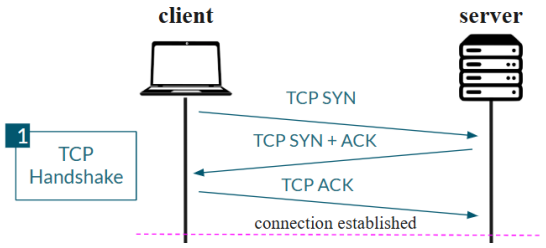

Figure 2: TCP Handshake structure

The second part is made up of the Certificate Check. First, the client sends a Client hello message including a list of supported cryptographic algorithms and a random number for session initialization. With a Server Hello the server specifying the chosen cryptographic algorithm and with another random number for the session. Then the server sends its digital certificate to the client for authentication, including its public key. The client verifies

the certificate with a validity check, signature and with issues control.

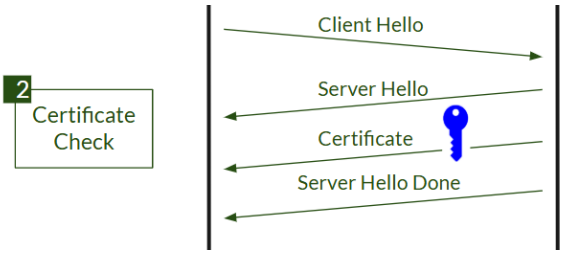

Figure 3: Certificate Check structure

The third part implements the key exchange to generate a session key. The client sends a pre-master key encrypted with the server's public key. Then the server decrypts the pre-master key using its private key and generates the session key. Both generate a shared symmetric key using the exchanged random numbers.

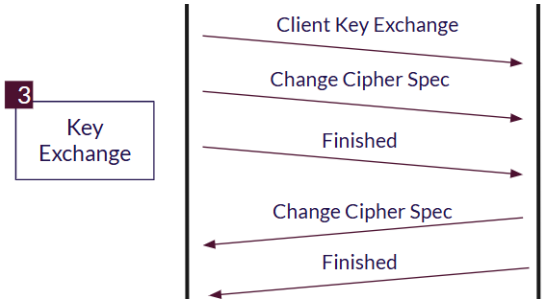

Figure 4: Key Exchange structure

In the last part of the process, known as "Data Transmission", the client and server exchange a "Change Cipher Spec" message, indicating they will start using symmetric encryption for communication. Additionally, the session key permits the encryption of the subsequent messages and data is securely transmitted using symmetric encryption.

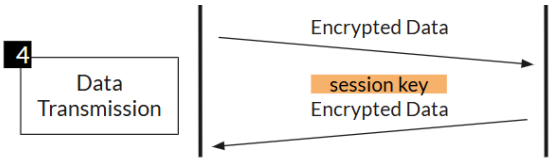

Figure 5: Data Transmission structure

Implementation

HTTP Server and Client

In this project, we start from building an HTTP server and an HTTP client to simulate basic communication over a network. The purpose is to lay the foundation for a secure communication system, starting with HTTP and moving towards HTTPS. At the beginning, the server listens on two different ports: one for handling HTTP requests and the other for implementing a simple handshake process. At the core of this implementation is the HTTP server, which is designed to accept POST requests. The server receives messages from clients, prints them out, and sends back a simple confirmation response. This setup represents the basic mechanism of HTTP communication, where data is transferred in plain text, without any encryption or security measures. On the client side, the HTTP client sends POST requests to the server, simulating a simple communication. The client sends a message (like "Hi!") to the server and expects a response. This behavior is standard for HTTP, where communication is open and unencrypted. In order to make the connection more secure, we implement a custom three-way handshake protocol for server-client communication. This process involves three steps: the client sends an initial

greeting ("Hi!"), the server responds with a question ("How are you?"), and finally, the client acknowledges with "Fine!". This handshake simulates the basic idea of connection establishment used in more complex protocols like TCP or SSL/TLS, which ensures that both parties are ready for secure and reliable data exchange.

The next goal is to evolve this basic HTTP system into a more secure HTTPS model. In the following steps, the handshake mechanism could be enhanced with encryption, certificates, and secure key exchanges to make the communication private and tamper-resistant, which is the basis of HTTPS. By first setting up an HTTP server and client with a basic handshake protocol, we prepare the groundwork for implementing these advanced security features, focusing on the next steps toward encrypted communication.

Generation private RSA key

We follow by generating a private RSA key, which serves as the cornerstone of the encryption mechanism. This key is created using a secure public exponent and a key size of 2048 bits, ensuring a strong level of cryptographic security. The private key will later be used to both sign the certificate and decrypt data encrypted by clients.

Creation of the certificate

Next, we construct the certificate itself. The subject of the certificate, representing the entity it identifies, is set to "localhost", making it suitable for local development or testing environments. Since this is a self-signed certificate, the issuer is the same as the subject, meaning the server both creates and vouches for its own identity. The certificate includes important metadata such as validity dates (starting from the current time and lasting one year) and a unique serial number. The cryptographic signing of the certificate is performed using the private key and the SHA-256 hashing algorithm. This step ensures the authenticity and integrity of the certificate, making it tamper-proof during communication.

Later, the private key and the certificate are serialized and saved to separate files, in the PEM format. The private key file will be kept securely on the server, while the certificate file will be shared with clients to establish trust during the TLS handshake. By generating this self-signed certificate, the code lays the groundwork for encrypting and securing communications over HTTPS, addressing key vulnerabilities present in plain HTTP. While self-signed certificates are sufficient for testing purposes, they would typically be replaced with certificates issued by trusted Certificate Authorities (CAs) in production environments.

Certificate Verification

Then we implement a mechanism to verify the validity of an X.509 certificate against a trusted certificate, such as a Certificate Authority (CA). At first, we load the received certificate and the trusted certificate from their respective PEM-encoded files. The trusted certificate's public key is then used to verify the signature of the received certificate, ensuring its authenticity and that it has not been tampered with. Additionally, the code checks the certificate's validity period to confirm that

it is being used within its specified time frame. If both the signature and the validity checks pass, the function returns True, indicating that the certificate is valid, otherwise, it returns False. This forms a basic but essential component of establishing trust in a secure communication system.

AES process

We proceed writing the code to let the client connect to the server on port 443, the standard port for HTTPS communication. It sends a simple GET request to the root path. If the server responds, the client decodes and prints the response content, then closes the connection and exits the loop.

In case the server is unavailable or an error occurs (e.g., the server is not yet ready to accept connections), the client catches the exception, logs a retry message with the error details, waits for one second, and attempts to reconnect. This ensures the client remains resilient and continues attempting until the server becomes accessible.

Testing

A crucial part of our code is the testing, which demonstrates the encryption and decryption of a secret message using AES (Advanced Encryption Standard) with a 256-bit key, providing a secure method for protecting sensitive data.

The process begins by generating a random 256-bit AES key using the get_random_bytes function. This key is critical for both encrypting and decrypting the data. The message to be encrypted is defined as a plaintext string.

During encryption, the message is transformed into ciphertext, ensuring that it becomes unreadable without the corresponding AES key and additional parameters. The encryption process also generates a nonce (a unique number used once) and a tag (used for authentication) to ensure data integrity and prevent tampering.

For decryption, the ciphertext, key, nonce, and tag are used to reconstruct the original message. If the decryption succeeds, it confirms that the ciphertext has not been modified and that the correct key was used. The decrypted message is then printed to verify that the process has successfully restored the original plaintext. This secure encryption-decryption flow is vital for protecting data in secure communication systems.

Performing the DOS attack

Once the HTTPS connection is established, we execute an active attack in which a malicious client intercepts the existing secure connection between the client and server, to control and manipulate the data flowing to the user. This kind of attack is known as “Denial of Service” (DoS) and its particular aim is to consume network bandwidth and/or processing capacity. To implement this, the malicious client maintains a persistent connection to the server on port 443 by reconnecting every 10 seconds. The attack works because the server has a weakness: it has a queue of length 5 but allows only one client at a time to connect to port 443. As a result, when the benevolent client attempts to connect, its connection fails.

The malicious client can now connect only once, afterward, the server memorizes that IP and adds it to a blacklist. This means that if the malicious client tries to reconnect, it will be immediately rejected. Additionally, the IP will remain blocked for at least 60 seconds. This procedure ensures that once the legitimate client attempts to connect, it will succeed without interference from the malicious client. If this is done on localhost, both the legitimate and malicious clients would share the same IP, resulting in both being blocked. However, when using Virtual Machines with different IPs, the code functions as intended. We also add a modification that allows the received packet from the server to be read and displayed. This makes it possible to observe that, once the malicious client connects, it does not send packets (as its sole intent is to disrupt the connection). In the localhost scenario, the legitimate client is also blocked, and the server does not receive anything because the IP is already blacklisted.

Results

Vision of terminal's output: main_malicious

Initially, the malicious client attempts to connect to the server with no results: the DoS attack cannot begin until the malicious client establishes a connection with the server. As soon as the server listens on port 127.0.0.1:433 (in which the 433 is a common port of HTTPS connections), the malicious client can attempt the desired connection to send, later, some requests to compromise the server. On the other hand, the legitimate client fails to connect to the server due to the destabilization made by the malicious.

In this figure (Figure 6), it's possible to see the output of the malicious scenario (taken from "interface.py"). The server starts listening on 127.0.0.1:443, and a malicious client attempts to connect. On the first connection, no data is sent (a typical DoS behavior), so the server blocks the IP for 60 seconds. The malicious client keeps trying to reconnect, but the server denies all attempts because the IP is blacklisted. Later, a legitimate client connects and sends a message (Hello, server!), but since it's using the same IP (127.0.0.1 in localhost), it also gets blocked, showing how shared IPs can complicate detection. This test highlights how blacklisting mitigates DoS attacks but creates issues for legitimate clients in localhost setups.

Vision of terminal's output: main_benevolent

At the beginning, the user has to write the port you want to use as HTTPS port. Then, the benevolent client successfully generates the certificate and the private key that are necessary for establishing a secure connection with the server during the TLS handshake. The client successfully connects to the HTTP server on port 80 and, in this way, it can now communicate using HTTP protocol. (Figure 7)

To start HTTPS protocol and achieve an encrypted connection, first the client and the server complete the 3-way handshake successfully, establishing a reliable TCP connection: the connection is now ready for data exchange. (Figure 8)

The client and server start to exchange information to negotiate security parameters, such as keys, to obtain authentication in the connection. A process that continues until the client verifies the server's certificate and finds it valid. (Figure 9)

During the shared session key, the client sends an encrypted key to the server to enable the generation of session keys for encryption. (Figure 10)

Both endpoints generate the same session key as a result of the key exchange algorithm. Then the client sends an encrypted message ("hello") to the server, which receives and decrypts it successfully. (Figure 11)

Even though the connection is established before the simulation, it remains consistent in the course of the process, staying faithful to the problem's requirements. This approach, while effective, introduces slight asymmetries in the output, particularly with print() statements. Asymmetric prints can occur because of the simultaneous execution of multiple operations. Using the Threads library it's possible to execute functions simultaneously, in which the client may print "Connections established" before the server has finished listening. Similarly, with the Socket library, the client may send a connection request and continue its execution, printing a message, while the server may still be waiting to receive the request. Additionally, printed messages can be influenced by output buffering, where messages generated at different times may appear out of order due to how they are handled by the operating system. However, the logical flow of the program remains unaffected.

Additionally, we used the Thread library, which enables us to run multiple files simultaneously. This feature further contributed to minor output inconsistencies but it didn't impact the final results or the correctness of the simulation.

Traffic Analysis through WireShark: benevolent scenario

- HTTP Traffic (Port 80)

Figure 12

Packet 1: SYN from 127.0.0.1:60522 to 127.0.0.1:80. This is the initial request from the client to establish a TCP connection with the HTTP server on port 80.

Packets 2-3: Completion of the 3-way handshake: SYN-ACK packet (server responds to the request), ACK packet (client confirms and establishes the connection).

These 3 last packets resample the TCP Handshake structure on Figure 2.

Packets 4-6: The client sends a message to the server: PSH, ACK packet contains the message "Hi server\n". The server responds with an ACK to confirm receipt of the message.

Packets 8 and 10: The server replies.

Packet 14: RST, ACK from client to server in order to end the HTTP communication.

- HTTPS Traffic (Port 443)

Figure 13.

(In order to perform it, breakpoints were used between the various phases)

Packets 15-17: This marks the start of a new connection on port 443. The traffic includes: 3-way TCP handshake.

Packets 18-23: Simulation of the 3-way handshake, in addition to the one with the socket.

Packets 24-31: Simulation of the certificate check: we can see that these packets have a longer length; for example the packet 28 is certain to be the certificate, while the 24 and 26 are the client hello and server hello.

This segment resample the Certificate Handshake on Figure 3.

Packets 32-33:There is the key exchange, from the client to the server.

Packets 34-37: These are the “Change Cipher Spec” and “Finished” from Client to server and vice versa.

We can refer to figure 4 for this exchange.

Last Packets 38-39: Sending the encrypted message from the client to the server. We refer to figure 5.

Traffic Analysis through WireShark: malevolent scenario

Figure 14

Packets 1-20: connection attempts by the malicious client.

Packets 21-27: the malicious client successfully connects to the server.

Packets 28-37: the benevolent client tries to connect to the server but fails.

General Observations

The timings are consistent. Each phase (HTTP, HTTPS handshake) occurs in short intervals. Delays between packets (e.g., 0.1s) align with the time.sleep() calls in the code. The TLS/SSL handshake sequence is correctly initiated and completed. Once the connection is established, all messages are encrypted. Continuation Data packets confirm the encryption of communication.

Simulation with VirtualBox:

Once the project was carried out locally, it was evolved by implementing virtual machines that impersonated the role of server and client. Windows 10 was used as the operating system for all machines, in this way it was necessary to insert new rules in the firewall of each one that allowed the reception and sending on the desired ports by the interested IP addresses. Thanks to the division of code, it was possible to carry out each phase in a more sequential manner. Below there are the images of the virtual machines screens:

In the benevolent case, it was enough to use a single virtual machine, together with the basic PC to simulate the communication. We refer to Figure 15 below.

Whereas in the malevolent case it was necessary to create 2 virtual machines to match the scenario roles, we put the addition of the malicious client in Figure 16.

Discussion

In conclusion, with this project we explore the critical importance of securing communications between the client and server, with a particular focus on the vulnerabilities exploited by the attack. From an HTTP protocol we have developed the connection to the secure one HTTPS implementing TCP Handshake, Certificate Check, Key Exchange and Data Transmission. Then, by simulating a DoS attack, we demonstrate how this insecure transition from HTTPS to HTTP can expose sensitive data to malicious interception and manipulation. Through this simulation, it is evident that while active attacks can be detected, they cannot be entirely prevented, underscoring the necessity of continuous advancements in security protocols. The results highlight the organizations' need to adopt very strong encryption standards and secure connections practices to protect user data and ensure more secure communications. Furthermore, this project reflects some evolving cybersecurity's challenges and the importance of robust measures to safeguard digital environments.

References

Bibliography

[1] https://cert-agid.gov.it/news/terzo-monitoraggio-sullutilizzo-del-protocollo-https-e-sullo-stato-di-aggiornamento-dei-cms-sui-sistemi-della-pa/

Figure's list

Figure 6: Vision of terminal's output: main_malicious

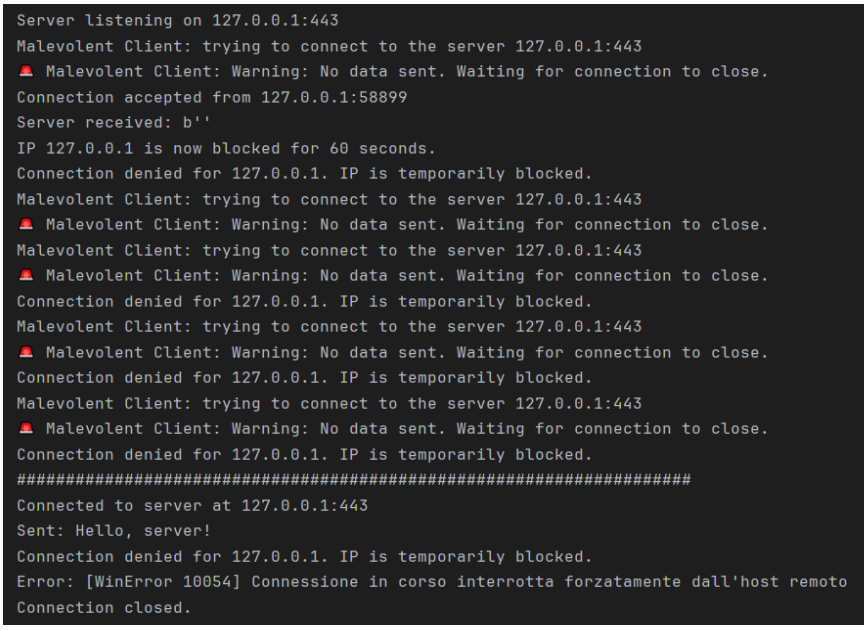

Figure 7: Vision of terminal's output: main_benevolent

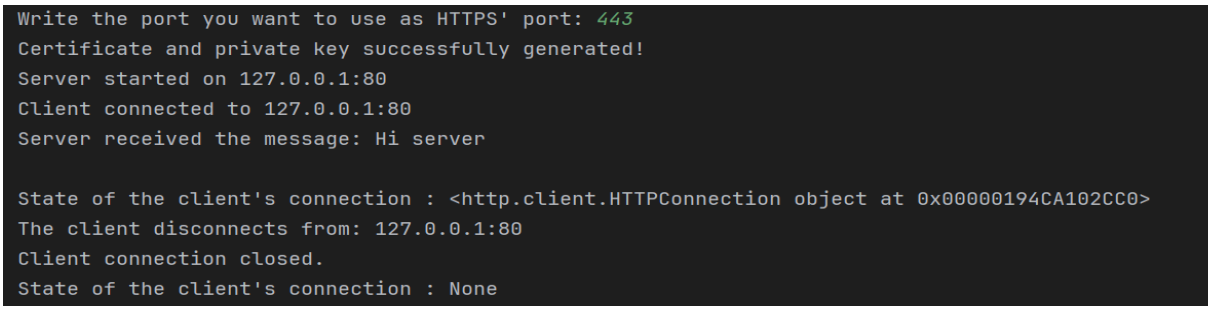

Figure 8: Vision of terminal's output: main_benevolent, start of HTTPS protocol

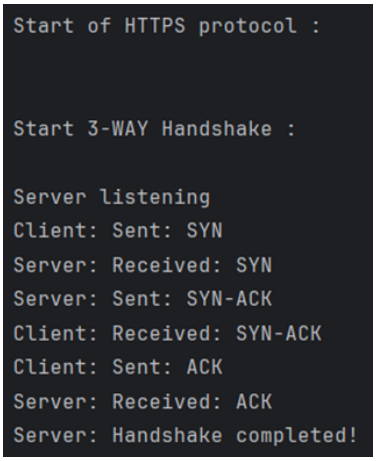

Figure 9: Vision of terminal's output: main_benevolent, start Certificate Check

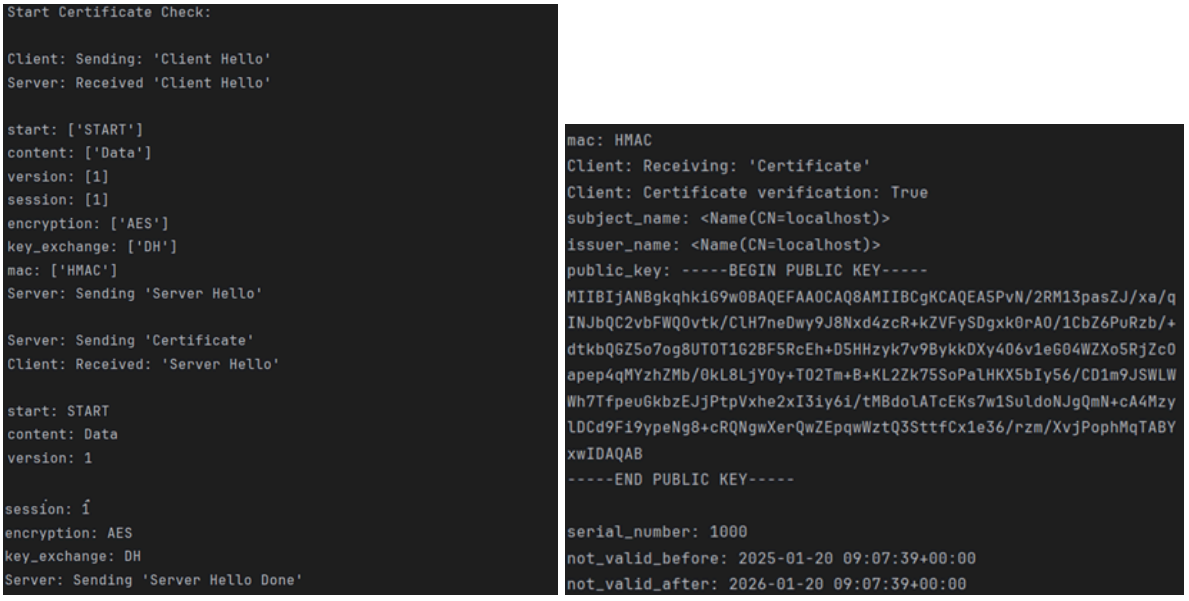

Figure 10: Vision of terminal's output: main_benevolent, start of Key Exchange

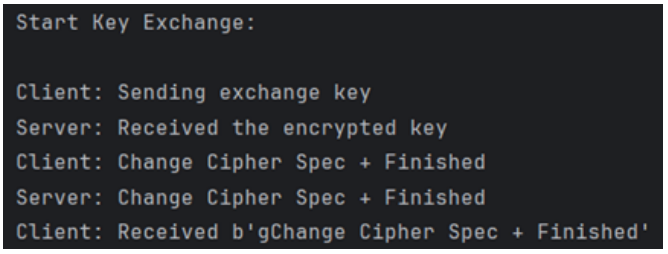

Figure 11: Vision of terminal's output: main_benevolent, session key

```
SESSION KEYS:

Client: b"\x9c\xb9\x82\xdd\xe6E\xcb\x00\xaa\x11\xbeT\x81\x9c\x9f\xfdr6\xe0\x96\xb0d\xc6\xb4t\x9a'\xb1\x91\x0b\x1c\xe1"
Server: b"\x9c\xb9\x82\xdd\xe6E\xcb\x00\xaa\x11\xbeT\x81\x9c\x9f\xfdr6\xe0\x96\xb0d\xc6\xb4t\x9a'\xb1\x91\x0b\x1c\xe1"
Message's test:
Write message to be sent: hello
Server: Received the message: b'\xb6\xcep\xe7\xe0\xa0Y\xdb\xa61\x0e\xe4X\xbc\xb0\xcfi\xc8\xc7|\xb5\x15S\x86\xf2\xa8\x05\x9d\xd0]a\xc2'
Server: Decrypted: hello

Process finished with exit code 0
```

Figure 12: Http communication via WireShark

No.	Time	Source	Destination	Protocol	Lengtl	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	56	63715 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
2	0.000052	127.0.0.1	127.0.0.1	TCP	56	80 → 63715 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
3	0.000077	127.0.0.1	127.0.0.1	TCP	44	63715 → 80 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
4	0.000105	127.0.0.1	127.0.0.1	TCP	153	63715 → 80 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=109 [TCP PDU reassembled in 6]
5	0.000114	127.0.0.1	127.0.0.1	TCP	44	80 → 63715 [ACK] Seq=1 Ack=110 Win=2619648 Len=0
6	0.000131	127.0.0.1	127.0.0.1	HTTP	54	POST / HTTP/1.1 (text/plain)
7	0.000137	127.0.0.1	127.0.0.1	TCP	44	80 → 63715 [ACK] Seq=1 Ack=120 Win=2619648 Len=0
8	0.001326	127.0.0.1	127.0.0.1	TCP	164	80 → 63715 [PSH, ACK] Seq=1 Ack=120 Win=2619648 Len=120 [TCP PDU reassembled in 12]
9	0.001342	127.0.0.1	127.0.0.1	TCP	44	63715 → 80 [ACK] Seq=120 Ack=121 Win=2619648 Len=0
10	0.001375	127.0.0.1	127.0.0.1	TCP	73	80 → 63715 [PSH, ACK] Seq=121 Ack=120 Win=2619648 Len=29 [TCP PDU reassembled in 12]
11	0.001381	127.0.0.1	127.0.0.1	TCP	44	63715 → 80 [ACK] Seq=120 Ack=150 Win=2619648 Len=0
12	0.001417	127.0.0.1	127.0.0.1	HTTP	44	HTTP/1.0 200 OK (text/plain)
13	0.001440	127.0.0.1	127.0.0.1	TCP	44	63715 → 80 [ACK] Seq=120 Ack=151 Win=2619648 Len=0
14	0.129936	127.0.0.1	127.0.0.1	TCP	44	63715 → 80 [RST, ACK] Seq=120 Ack=151 Win=0 Len=0

Figure 13: Https communication via WireShark

15	0.302857	127.0.0.1	127.0.0.1	TCP	56	63983 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
16	0.302937	127.0.0.1	127.0.0.1	TCP	56	443 → 63983 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
17	0.302987	127.0.0.1	127.0.0.1	TCP	44	63983 → 443 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
18	0.303045	127.0.0.1	127.0.0.1	TCP	47	63983 → 443 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=3 [TCP PDU reassembled in 22]
19	0.303055	127.0.0.1	127.0.0.1	TCP	44	443 → 63983 [ACK] Seq=1 Ack=4 Win=2619648 Len=0
20	0.303207	127.0.0.1	127.0.0.1	TCP	51	443 → 63983 [PSH, ACK] Seq=1 Ack=4 Win=2619648 Len=7
21	0.303219	127.0.0.1	127.0.0.1	TCP	44	63983 → 443 [ACK] Seq=4 Ack=8 Win=2619648 Len=0
22	0.303346	127.0.0.1	127.0.0.1	TCP	47	63983 → 443 [PSH, ACK] Seq=4 Ack=8 Win=2619648 Len=3
23	0.303360	127.0.0.1	127.0.0.1	TCP	44	443 → 63983 [ACK] Seq=8 Ack=7 Win=2619648 Len=0
24	0.404175	127.0.0.1	127.0.0.1	TCP	181	63983 → 443 [PSH, ACK] Seq=7 Ack=8 Win=2619648 Len=137
25	0.404193	127.0.0.1	127.0.0.1	TCP	44	443 → 63983 [ACK] Seq=8 Ack=144 Win=2619648 Len=0
26	0.504539	127.0.0.1	127.0.0.1	TCP	167	443 → 63983 [PSH, ACK] Seq=8 Ack=144 Win=2619648 Len=123
27	0.504577	127.0.0.1	127.0.0.1	TCP	44	63983 → 443 [ACK] Seq=144 Ack=131 Win=2619648 Len=0
28	0.504739	127.0.0.1	127.0.0.1	TCP	1017	443 → 63983 [PSH, ACK] Seq=131 Ack=144 Win=2619648 Len=973
29	0.504798	127.0.0.1	127.0.0.1	TCP	44	63983 → 443 [ACK] Seq=144 Ack=1104 Win=2618624 Len=0
30	0.504887	127.0.0.1	127.0.0.1	TCP	93	443 → 63983 [PSH, ACK] Seq=1104 Ack=144 Win=2619648 Len=49
31	0.504896	127.0.0.1	127.0.0.1	TCP	44	63983 → 443 [ACK] Seq=144 Ack=1153 Win=2618624 Len=0
32	0.609127	127.0.0.1	127.0.0.1	SSL	300	Continuation Data
33	0.609145	127.0.0.1	127.0.0.1	TCP	44	443 → 63983 [ACK] Seq=1153 Ack=400 Win=2619392 Len=0
34	0.765175	127.0.0.1	127.0.0.1	TCP	74	63983 → 443 [PSH, ACK] Seq=400 Ack=1153 Win=2618624 Len=30
35	0.765193	127.0.0.1	127.0.0.1	TCP	44	443 → 63983 [ACK] Seq=1153 Ack=430 Win=2619136 Len=0
36	0.865443	127.0.0.1	127.0.0.1	TCP	74	443 → 63983 [PSH, ACK] Seq=1153 Ack=430 Win=2619136 Len=30
37	0.865458	127.0.0.1	127.0.0.1	TCP	44	63983 → 443 [ACK] Seq=430 Ack=1183 Win=2618368 Len=0
38	2.372927	127.0.0.1	127.0.0.1	SSL	76	Continuation Data
39	2.372944	127.0.0.1	127.0.0.1	TCP	44	443 → 63983 [ACK] Seq=1183 Ack=462 Win=2619136 Len=0

Figure 14: connection attempts by the malicious client and by the benevolent client to the server via WireShark

No.	Time	Source	Destination	Protocol	Length	Info
4	0.500674	127.0.0.1	127.0.0.1	TCP	44	443 → 64029 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
5	1.001319	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64029 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
6	1.001348	127.0.0.1	127.0.0.1	TCP	44	443 → 64029 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
7	1.501996	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64029 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
8	1.502012	127.0.0.1	127.0.0.1	TCP	44	443 → 64029 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
9	2.002925	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64029 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
10	2.002939	127.0.0.1	127.0.0.1	TCP	44	443 → 64029 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
11	7.003570	127.0.0.1	127.0.0.1	TCP	56	64030 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
12	7.003582	127.0.0.1	127.0.0.1	TCP	44	443 → 64030 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
13	7.503934	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64030 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
14	7.503948	127.0.0.1	127.0.0.1	TCP	44	443 → 64030 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
15	8.004618	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64030 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
16	8.004631	127.0.0.1	127.0.0.1	TCP	44	443 → 64030 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
17	8.505308	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64030 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
18	8.505322	127.0.0.1	127.0.0.1	TCP	44	443 → 64030 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
19	9.005678	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64030 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
20	9.005694	127.0.0.1	127.0.0.1	TCP	44	443 → 64030 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
21	14.006434	127.0.0.1	127.0.0.1	TCP	56	64032 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
22	14.006487	127.0.0.1	127.0.0.1	TCP	56	443 → 64032 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
23	14.006507	127.0.0.1	127.0.0.1	TCP	44	64032 → 443 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
24	14.006803	127.0.0.1	127.0.0.1	TCP	44	443 → 64032 [FIN, ACK] Seq=1 Ack=1 Win=2619648 Len=0
25	14.006837	127.0.0.1	127.0.0.1	TCP	44	64032 → 443 [ACK] Seq=1 Ack=2 Win=2619648 Len=0
26	14.016927	127.0.0.1	127.0.0.1	TCP	44	64032 → 443 [FIN, ACK] Seq=1 Ack=2 Win=2619648 Len=0
27	14.016958	127.0.0.1	127.0.0.1	TCP	44	443 → 64032 [ACK] Seq=2 Ack=2 Win=2619648 Len=0
28	24.007221	127.0.0.1	127.0.0.1	TCP	56	64033 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
29	24.007234	127.0.0.1	127.0.0.1	TCP	44	443 → 64033 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
30	24.507686	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64033 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
31	24.507707	127.0.0.1	127.0.0.1	TCP	44	443 → 64033 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
32	25.008432	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64033 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
33	25.008455	127.0.0.1	127.0.0.1	TCP	44	443 → 64033 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
34	25.509149	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64033 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
35	25.509179	127.0.0.1	127.0.0.1	TCP	44	443 → 64033 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
36	26.009922	127.0.0.1	127.0.0.1	TCP	56	[TCP Port numbers reused] 64033 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
37	26.009936	127.0.0.1	127.0.0.1	TCP	44	443 → 64033 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Figure 15: Client and server using VirtualBox

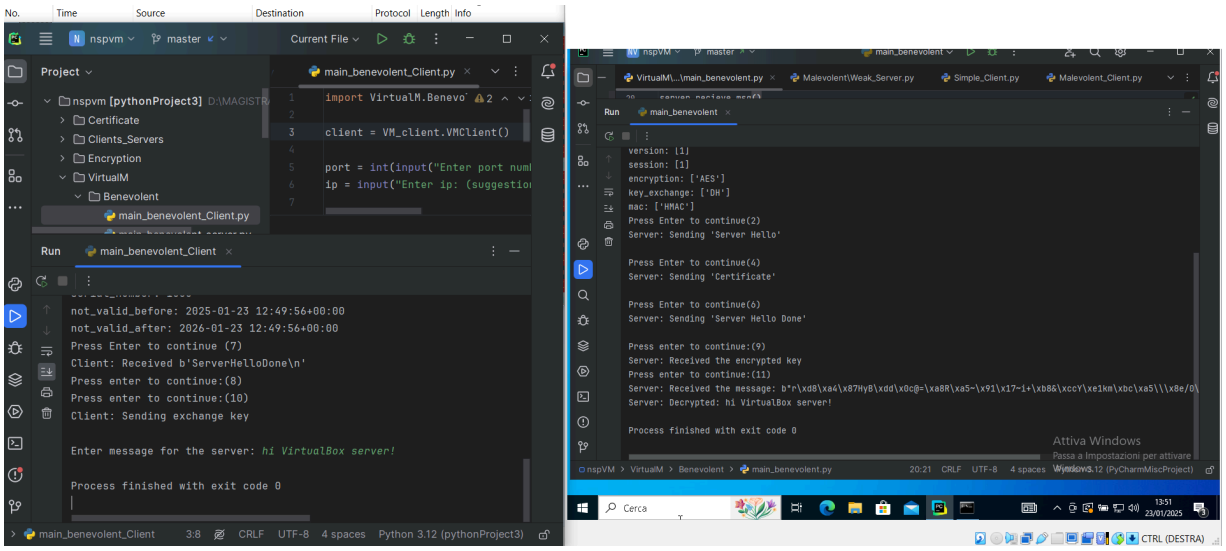

Figure 16: Malevolent Client using VirtualBox