UNIVERSITÄT KOBLENZ · LANDAU

SINS: Security in Networked Systems WS2019/20

Assignment 2

Secure Protocol Design

Group 1:

Jan Bings (216 200 708)

Bhupender Kumar Saini (219 100 887)

Ravineesh Goud (219 100 836)

Marvin Schwenkmezger (216 201 620)

Supervisor: Prof. Dr. Andreas Mauthe

Co-supervisor: Mr. Mike Reuther

January 12, 2020

List of Abbreviations

CHAP ..... Challenge Handshake Authentication Protocol

DOS ..... Denial of service

IOT ..... Internet of things

IP ..... Internet protocol

JSON ..... JavaScript object notation

RSA ..... Rivest–Shamir–Adleman

UDP ..... User datagram protocol

List of Figures

List of Tables

Table of Contents

1 Protocol Description

The protocol designed is called “Secure_Protocol_Assignment_2“. Designed protocol can be utilized for various information sharing purposes and one of purpose of protocol can be to establish a connection between Industrial/ IOT devices to exchange status information of those devices. In this case, both, server and client can be in the same devices or can act separately in different devices.

Major requirements include that the protocol description should be reliable, and session orientated. Moreover, the messages inside the packets are encrypted and the integrity of the messages should be guaranteed.

ID	Requirement Description	Status
1	Client and Server require mutual authentication.	Implemented
2	All messages should be protected against modification and they must be authenticated.	Implemented
3	All data in DATA_RESPONSE messages must be transmitted confidentially (Encryption should be introduced).	Implemented
4	All messages should be protected against modification and they must be authenticated. (CRC32 and CHAP)	Implemented
5	Encryption key should be refreshed every 10 DATA_RESPONSE messages.	Not Implemented
6	Client starts with a HELLO message which is answered by the server with a HELLO_ACK message if the connection is accepted.	Implemented
7	DATA_REQUEST can only be initiated after client has Session_ID and Authorization_ID provided by the server.	Implemented
8	Each DATA_RESPONSE message contains a DATA field containing a string.	Implemented
9	Client should send a DATA_REQUEST every 30 seconds to server.	Implemented
10	The Client terminates the connection after 30 minutes.	Implemented
11	When closing the session, the Client transmits a final CLOSE message to the Server.	Implemented

Table 1: Requirement Matrix (Own illustration)

Practically, it is necessary to change keys after a while to keep the security intact which can cost the network bandwidth. Trade-off is required between security and network use, we

propose changing key every ten messages. In proposed protocol, asymmetric RSA encryption is used but exchange of keys are currently physically. However, exchange of public and private key can be implement using Diffie-Hellman in addition.
Requirement 9 is changed in the demo to every 2 seconds, to make demonstration easier. For the same reason, the client terminates the connection after 10 seconds instead (requirement 10).

1.1 State Machines

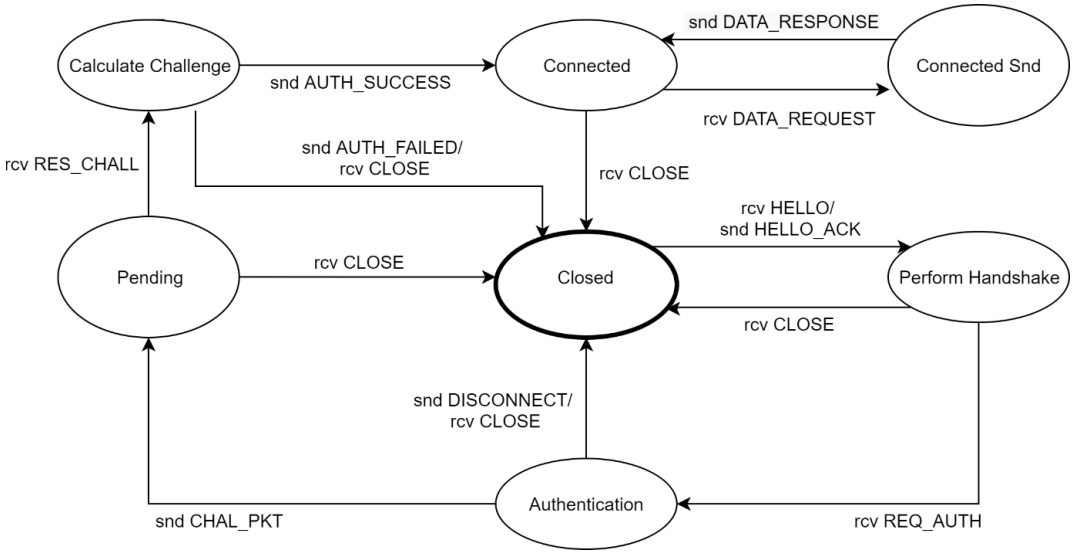

Figure 1: Server State Machine (Own illustration)

Figure 1 shows a state machine describing the states of the server. In total there are seven states, the server can enter. It always starts in the *closed/ listening* state. In this state, he is always listening for client requests. Whenever a CLOSE message is received, the server enters the *closed/ listening* state again. When the server receives a HELLO message or sends an acknowledge to it, it enters the *Perform Handshake* state. Afterward, it receives a request for the authentication (REQ_AUTH). This leads to the state *Authentication*. Then, the authentication is validated. Either the server ends into the closed state by sending a DISCONNECT message or the authentication was possible. Then, the server sends a challenge to the client and waits for the challenge (Pending state) of the client. There, the server decides to end the connection in the case of a failed authentication or sends a message of successful authentication to the client. Now the server enters the *Connected* state. In this state, there are two possible scenarios. One is that the connection is closed via a close message or the client sends a request for data that gets answered by the server with a data response. Corresponding to this the server switches between two states of being connected.

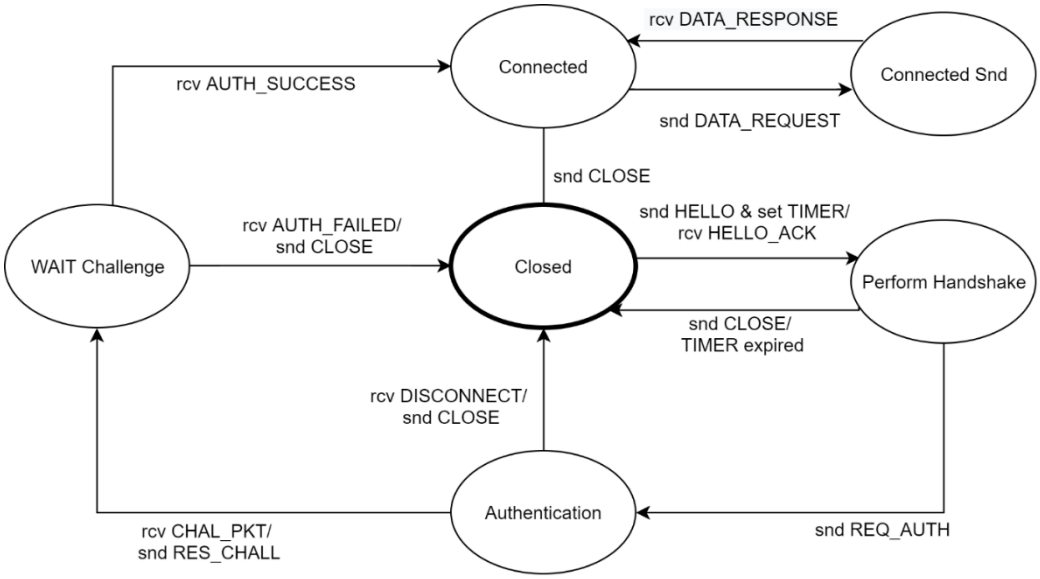

Figure 2: Client State Machine (Own illustration)

Figure 2 shows the client state machine. It also starts in a closed state. The client switches into the perform handshake state when it sends a HELLO message to the server or when an acknowledgment for it is received. Corresponding to the data request message a timer is set. If the timer expires, the connection is closed. After the handshake, an authentication request is sent to the server. This is either answered by a DISCONNECT message or a challenge. This message and the one on the challenge of the server are needed to enter *WAIT Challenge*. From here, the server can send a message on a failed authentication which would lead to a closed connection. Otherwise, it will send a success message. Then both devices are connected. In this state, we have a loop of data requests and responses until the connection is closed by the client.

1.2 Handshake

As already shown in the state machines, the process begins with a three-way handshake.

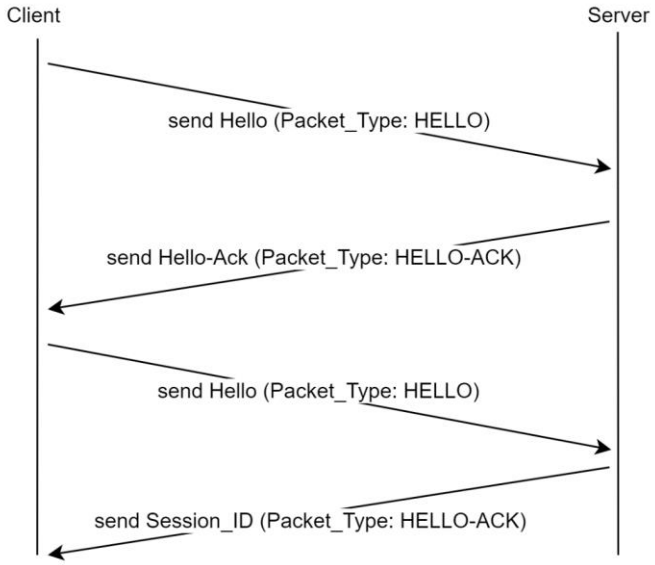

Figure 3: Three-Way Handshake (Own illustration)

In the beginning, a three-way handshake is performed as shown in Figure 3. After successful execution, the server transmits the session key to initiate the session between the client and the server. Afterward, the Challenge Handshake Authentication Protocol (CHAP) is used to authenticate the client. Only if the client passes the authentication successfully, the connection is not closed.

The Handshake itself starts with the HELLO message sent by the client to the server. In the further description, it is referred as packet type 1. The HELLO message is answered by the server with an acknowledgment. When the client receives it, he will also send an acknowledgment of the HELLO message. When the server receives this, it sends a session id to the client. The session id is unique for every client. Afterward, the connection is in the authentication state. Therefore, CHAP is introduced.

1.3 Authentication with Chap

Figure 4 shows a schema of the functionality of the CHAP authentication. First, the client sends a REQ_AUTH. It is answered by the server through the challenge, which is a random integer between 0 and 255 (CHAL_PKT) called identifier. The client then sends the response aa hash-value (RES_CHAL). It is calculated by the previously received identifier combined with a predefined codeword (in the demo: “Password”) back to the server, which acknowledges this with an authorization ID (AUTH_SUCCESS).

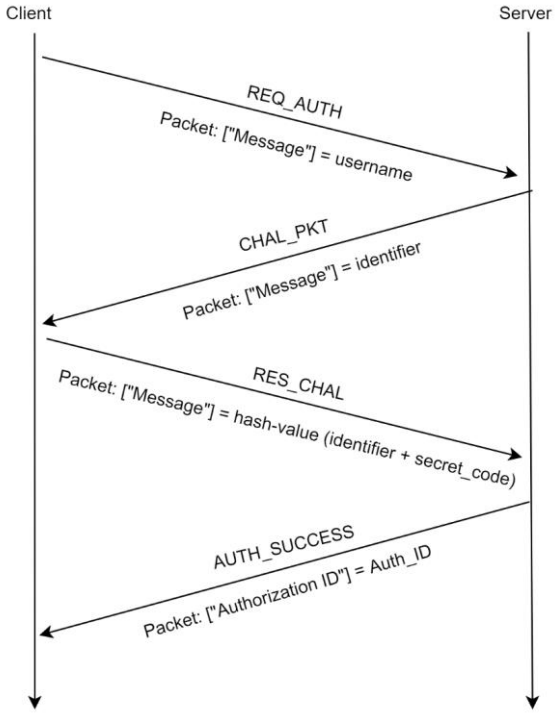

Figure 4: CHAP Authentication (Own illustration)

1.4 Security Features

The protocol design enables multiple security features. One attack might be bypassing the first HELLO message by starting with sequence number 2. Thereby, the handshake will be skipped, but no further action is possible as the server won't provide further authorization. The following additional security elements are included.

1.4.1 Authentication

First, CHAP is used for user authentication as described above. Moreover, the protocol is confidential, and integrity is always provided.

1.4.2 Integrity

CRC-checksums are used to ensure the message's integrity. This is done by building and validating 32-bit checksums via .crc32 bit command. Validation is done by comparing the original checksum received with the packet and a checksum calculated with a copy of the original packet minus the checksum contained.

1.4.3 Confidentiality

Confidentiality is provided as the protocol uses RSA, an asymmetrical procedure for encryption. RSA typically uses a combination of private and public keys. Those are stored as .pem file. The public key of the device receiving the new message is used for encryption. So, it

can use its own private key for decryption. The keys have a length of 2048 bit. Therefore, the module “cryptography” by hazmat[1] is used. Hashes with a size of 256 bit are further used in the encryption process. This protocol can be enhanced by securing public and private keys with an additional password.

1.5 Message Format

A packet is structured according to the following JSON scheme (Figure 5: Packet Structure (Own illustration)):

```
Packet_Structure = {
        "Source_Address": "",
        "Destination_Address":  "",
        "S_PortNumber": "",
        "D_PortNumber": "",
        "Packet_Type": "",
        "Protocol_ID": "",
        "Sequence_Number": "",
        "Session_ID": "",
        "Authorization_ID":"",
        "Message": "",
        "Timestamp": "",
        "Checksum": "",
}
```

Figure 5: Packet Structure (Own illustration)

1.5.1 Source Address

The source address field contains the 32-bit IPv4 address of the sender.

1.5.2 Destination Address

It contains the 32-bit IPv4 address of the receiver.

1.5.3 Source Port

The field provides the 2-byte value of the sender’s port.

[1] https://cryptography.io/en/latest/hazmat/primitives/

1.5.4 Destination Port

The destination port field contains the 2-byte value of the destination port.

1.5.5 Packet Type

The field displays a hex value related the packet type. They are shown in chapter 1.6. In the appendix' demo code, string values are used instead for easier comprehensibility.

1.5.6 Protocol ID

Protocol ID is used to determine the protocol. It contains a string value for demonstration purposes. In a real scenario, it is unique ID should be used.

1.5.7 Sequence Number

The sequence number indicates the order of the messages. It is a random 16-bit number between 0-65535. When 65535 is reached, it will continue with 0.

1.5.8 Session ID

It is a 16-byte value which is unique for every session. It is assigned by the server.

1.5.9 Message

In the *Message* field, the actual data of the message.

1.5.10 Timestamp

The 32-bit timestamp is an iso-format of the UTC time.

1.5.11 Checksum

A 32-bit checksum on the ten packet fields mentioned above.

1.6 Message Type

Packet type name	Value	Direction of flow	Description
HELLO	0x01	Client-Server	Connection request
CLOSE	0x02	Client-Server	Close connection
HELLO_ACK	0x03	Server-Client and Client-Server	Reply to HELLO
DATA_REQUEST	0x04	Client-Server	Requesting data

DATA_RESPONSE	0x05	Server-Client	Sending data
REQ_AUTH	0x06	Client-Server	Authentication exchange
RES_CHALL	0x07	Client-Server	Challenge exchange
CHAL_PKT	0x08	Server-Client	Challenge exchange
AUTH_SUCCESS	0x09	Server-Client	Authentication successful
AUTH_FAILED	0x10	Server-Client	Authentication failed
DISCONNECT	0x11	Server-Client	Wrong username wanted to start connection
RESERVED	0x12	-	-
RESERVED	0x13	-	-
RESERVED	0x14	-	-
RESERVED	0x15	-	-

Table 2: Message Flows (Own illustration)

The message flows are described in the table above. They can also be found in the state machines. The packet type names assigned are used in the protocol header in the field *packet_type*. In difference to real implementations, the one shown in the appendix uses the strings in the packet type field instead of the hex values assigned above. The last four fields are reserved for future purposes.

2 Test Cases Description

Apart from major requirement of successful communication between client-server.
Below mentioned are the Very high- and high-test cases need to perform in order to test this protocol.

ID	Test case Description	Priority	Expected outcome
1	Sending correct CRC to server and client	VERY HIGH	Communication should not break
2	Sending wrong CRC calculation to server or client both	VERY HIGH	Communication should break with DISCONNECT

ID	Test case Description	Priority	Expected outcome
3	Modify packet in between and send to both server and client	VERY HIGH	Communication should DISCONNECT with proper error message
4	Run the communication session where the sequence number increases beyond 65535	VERY HIGH	Sequence number should reset to 0
5	Perform handshaking with server with sequence number starts with 1	VERY HIGH	The client should get a session ID once the server gets the sequence number 2
6	Perform handshaking with server with incorrect sequence number	VERY HIGH	Server should disconnect communication without sharing session ID
7	Verify response of the server after sending correct challenge-response	VERY HIGH	Server should reply AUTH_SUCCESS and share the Authorization_ID
8	Verify response of the server after sending incorrect challenge-response	VERY HIGH	Server should reply AUTH_FAILED and do not share the Authorization_ID
9	Verify the time interval between the data request packet	VERY HIGH	The time should be 30 seconds
10	Verify that the client terminates the session after every 30 minutes	VERY HIGH	The client closes the connection after every 30 minutes with sending the last message to the server with CLOSE packet type
11	Verify the time period of closing connection if no packets received	VERY HIGH	Server should close the session if no packets received in 20 seconds
12	Verify the packet type DATA_RESPONSE contains message in encrypted format	VERY HIGH	Message should be encrypted with RSA 2048 key size
13	Verify server and client communication with other pairs of public and private key	VERY HIGH	Communication should be successful with other pairs of public and private keys too

ID	Test case Description	Priority	Expected outcome
14	Establish connection with wrong session ID	HIGH	Communication should disconnect with the DISCONNECTED error message
15	Establish connection with correct session ID and wrong authorization ID	HIGH	Communication should disconnect with the DISCONNECTED error message
16	Establish connection with correct session ID and authorization ID	HIGH	Communication should be successful
17	Run server and client in low memory configured computer system	HIGH	Communication should work without any problem.
18	Run multiple client instances and single server configuration	HIGH	Communication should not terminate for any client
19	Verify the data message in Wireshark	HIGH	It should be encrypted.

Table 3: Test Case Description (Own illustration)

3 Potential Attacks

3.1.1 Replay-Attack

The first potential attack is a replay attack. Based on this, a man-in-the-middle attack is then possible, or you can take over the session (session hijacking).

3.1.2 Physical theft of public and private keys and secret code (“Password”) and username

Another way to attack the protocol is to steal the public and private keys on the physical medium. In addition, the loss of the username and password can also be a security vulnerability and result in a potential attack. But only access with username is not possible because you need the public and private keys.

3.1.3 Denial of Service (DOS)

If client sends a request data to the server with source address (255.255.255.255) in packet, the server forwards it to all devices in the network. This allows a TCP sync float attack, which reduces performance.

Another possible attack can be SYN-FLOOD where client keeps sending HELLO messages in between timeout interval and don't reply to server response – server always responds with HELLO-ACK and then waits till timeout.

Appendix

The complete server and client python code are included on the following pages:

Appendix A: Client and Server Code