Report for the Course on Cyber-Physical Systems and IoT Security

Authentication of IoT Device and IoT Server Using Secure Vaults

Boscolo Meneguolo Luca - 2113488

1. Objective

In the past years, we have seen a rapid proliferation of Internet of Things (IoT) devices. The fast growth poses a problem under a lot of aspects, from networking[1] to authentication. Ensuring that IoT devices can securely identify and communicate with each other is crucial, as they share a wide variety of personal information.

In this work, we will refer to the paper [1] to explore a three-way authentication protocol. The authors have designed a multi key authentication mechanism, such that, even if the secret key (or a combination of keys) used for ongoing authentication is retrieved successfully by the attacker, the attacker cannot gain access to the unused authentication keys.

2. System Setup

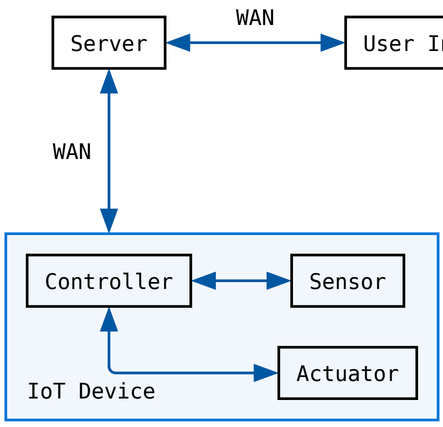
This project aims to implement the authentication protocol. The decision on the programming language to use to implement the simulator was based on some observations. We don't need the efficiency of C in such simulation. IoT devices offer low computational power, and although a real implementation would require a fast and versatile programming language (with direct access to the memory), for the sake of the demonstration we can use a modern language, although with limited memory management. The choice was then to adopt Kotlin, because it offers modern commodities and features, and it is 100% compatible with the well established Java Runtime Environment.

2.1. Overview of the System

An IoT System is composed by three classes of devices:

- IoT device: its purpose is to collect data from the sensors, and send them to the server;
- user interface: provides the user a configurable environment to interact with the system;
- server: collects data from the devices, and is responsible to bridge them to provide the user interface with useful information.

[1]https://en.wikipedia.org/wiki/Internet_of_things?utm_source=chatgpt.com#Addressability.



IoT devices

IoT devices need to perform lightweight operations, and may be battery-powered. For these reasons, they offer very low computational capabilities, and cost constraints limit the memory available. The System consists of d IoT devices, and to every IoT device is assigned a unique identification number.

Server

The server has a strongly protected database. The server is also considered to have unlimited resources, as it takes responsibility to authenticate every IoT device.

Vault

The devices have access to a vault K. The vault is substantially a secure storage containing n keys, each one of them being m bits long. The vault needs to be shared between one IoT device and the server, and must be consistent at all times. Obviously the server must have d different instances of the vault, where one corresponds to a specific IoT device. Parameters m and n are tuned by the developers, to balance security and computational power.

Channel

The devices send and receive messages on a reliable[2] communication channel, and the server usually is in a remote location, to provide a cloud infrastructure. The paper's authors specify that side channel attacks are not prevented. In fact the communication can be eavesdropped, but the authentication algorithm is secure from that point of view.

Authentication

1. The IoT device initiates the authentication process by sending message M_1.

   IoT device $\longrightarrow$ Server : M_1

   $M_1 = (\text{device\_ID}, \text{session\_ID})$

   where device_ID is the unique identifier of the device, and session_ID is a unique identifier of the session.
2. The Server responds by sending message M_2.

[2]No data loss in the process.

Server $\longrightarrow$ IoT device : M_2

$M_2 = (C_1, r_1)$

where challenge $C_1 = \{c_{11}, c_{12}, ..., c_{1p}\}$ is a set of p randomly picked distinct indices, and r_1 is a randomly generated number.

Note that $p < n$ and the values c are between 0 and $n - 1$.

3. The IoT device responds by sending message M_3.

   IoT device $\longrightarrow$ Server : M_3

   $M_3 = E_{k_1}(r_1 \parallel t_1 \parallel C_2 \parallel r_2)$

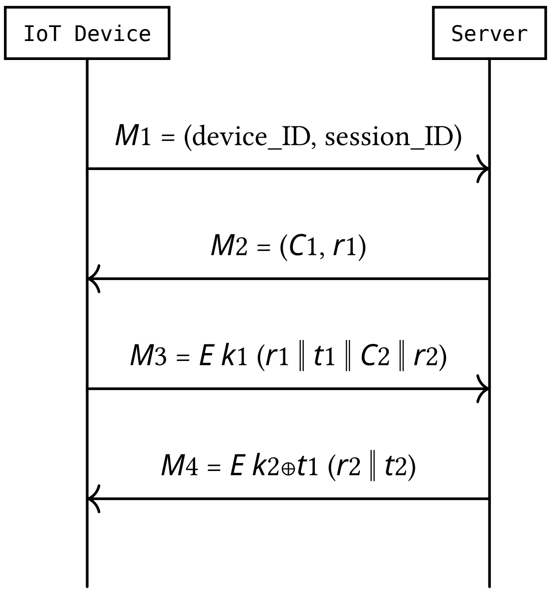
   where $k_1 = K[c_{11}] \oplus K[c_{12}] \oplus ... \oplus K[c_{1p}]$, and t_1 is a randomly generated number. Challenge C_2 is a set of q randomly picked distinct indices, and r_2 is a randomly generated number. Note that $C_1 \neq C_2$. E_{k_1} refers to a symmetric encryption algorithm, using key k_1.
4. The Server decrypts M_3, by computing the correct key K_1 from its vault. The Server then responds by sending message M_4.

   Server $\longrightarrow$ IoT device : M_4

   $M_4 = E_{k_2 \oplus t_1}(r_2 \parallel t_2)$

   where $k_2 = K[c_{21}] \oplus K[c_{22}] \oplus ... \oplus K[c_{2q}]$ and t_2 is a randomly generated number.

   Moreover, the Server computes $t = t_1 \oplus t_2$ and is used as key to encrypt future messages.
5. The IoT device decrypts M_4, by computing the correct key K_2 from its vault and checks if the received value for r_2 matches the sent one. Moreover, the IoT device computes $t = t_1 \oplus t_2$ and is used as key to encrypt future messages.



Vault update

When the session is terminated, the vault must be updated to both the Server and IoT Device. After every session, the value of secure vault is changed based on the data exchanged between the server and the IoT device.

Given $h = H_{k_h}(K)$,

where H is a HMAC function, K is the current vault, k_h is the key for HMAC, obtained from the exchanged messages.

The current value of secure vault is divided into j equal partitions of k bits. The partition are `xor` operated with $h \oplus i$, where i is the index of the partition. If the size of the secure vault is not divisible by k bits, 0 is padded at the end to create j equal partitions.

3. Software engineering

Message.kt

This class wraps messages. Every `Message` object has three fields:

- `val source: UByteArray`
- `val destination: UByteArray`
- `val payload: MutableList<UByteArray>`

The payload is designed as a mutable list, in order to accomodate the need to send more distinct messages in one go. This will primarily be exploited in this implementation in order to let the receiving device know which kind of message has been sent. The base implementation consists of three kinds of message types:

1. `AUTH = ubyteArrayOf(0u)`, for authentication processes;
2. `DEAUTH = ubyteArrayOf(1u)`, for deauthentication;
3. `MESG = ubyteArrayOf(2u)`, to send messages, upon a successful authentication.

The message identification will always be the first element in the payload, and any message with unknown identification will be discarded.

Device.kt

This class is the base class for a generic device in the System. Every device has a unique ID and an initial vault. Instances of the `Device` class have an implementation of a pseudo random number generator

```
return UByteArray(size) {
  Random.nextInt(0, 256).toUByte()
}
```

and a basic AES encryptor/decryptor

The `Device` base class is extended by `IoTDevice` and `Server`, and they provide the implementation for custom logic to handle the authentication phases.

Channel

This class is responsible for the communication between devices. Every device has a message inbound and outbound queues, and after every iteration, the `Channel` sends the messages to their desired destination. This way, it's possible to coordinate all messages to the correct queue. In every channel we have one and only one `Server`, with special ID $= 255$, and up to 255 instances of `IoTDevices`, with ID starting from 0.

```kotlin
protected fun encrypt(message: UByteArray, key: UByteArray): UByteArray {
    val secretKey = SecretKeySpec(key.toByteArray(), "AES")
    val cipher = Cipher.getInstance("AES")
    cipher.init(Cipher.ENCRYPT_MODE, secretKey)
    val encryptedBytes = cipher.doFinal(message.toByteArray())
    return encryptedBytes.toUByteArray()
}

protected fun decrypt(encryptedMessage: UByteArray, key: UByteArray): UByteArray {
    val secretKey = SecretKeySpec(key.toByteArray(), "AES")
    val cipher = Cipher.getInstance("AES")
    cipher.init(Cipher.DECRYPT_MODE, secretKey)
    val decryptedBytes = cipher.doFinal(encryptedMessage.toByteArray())
    return decryptedBytes.toUByteArray()
}
```

Snippet 1: AES encryption algorithm implementation.

4. Experiments, Results and Discussion

The program is already compiled. The following command is required to execute it:

```
java -jar authentication.jar
```

Just after running the program, we can input the number of IoT devices in the System. We insert 5. Then we are prompted with the vault specifications: we can choose the number of keys n and the key length m. We input $n = 50$ and $m = 16$.[3]

We print the channel information by pressing 1:

```
1) [IoT]        ID: 0           Auth: false
2) [IoT]        ID: 1           Auth: false
3) [IoT]        ID: 2           Auth: false
4) [IoT]        ID: 3           Auth: false
5) [IoT]        ID: 4           Auth: false
6) [SERVER]     ID: 255         Authenticated IoT device(s):
```

As we can see, none of the devices are authenticated, and this is consistent also on the server.

We press 2, to start a new authentication. We choose IoT device number 0. A total of 4 messages are rapidly exchanged between the device and the server. We will only take a look at the first:

```
Source: 0, Destination: 255
Payload contains 3 field(s):
1) 0
2) 0
3) 78, 244, 155, 150, 88, 195, 209, 227, 125, 98, 210, 165, 91, 42, 220, 240
```

We can see that the message has source = 0 and destination = 255. 0 is the ID of the chosen IoT device, while 255 is the ID of the server. The payload consists of three fields:
1. the AUTH flag, to let the server know it's an authentication request;
2. the device ID;
3. the randomly generated session ID.

All four messages follow exactly the specifications of the algorithm, and this is confirmed by pressing 1 again to print the status:

[3]For the AES encryption, only 16, 24 and 32 bits are supported.

```
1) [IoT]        ID: 0           Auth: true      Key: 218, 176, 75, 154, 14, 24, 230, 239, 41, 49, 38, 2, 92, 150, 119, 21
2) [IoT]        ID: 1           Auth: false
3) [IoT]        ID: 2           Auth: false
4) [IoT]        ID: 3           Auth: false
5) [IoT]        ID: 4           Auth: false
6) [SERVER]     ID: 255         Authenticated IoT device(s):
                                IoT ID: 0       Key: 218, 176, 75, 154, 14, 24, 230, 239, 41, 49, 38, 2, 92, 150, 119, 21
```

Both the IoT device and the server share the same key t.

Now, we try to authenticate IoT device number 3:

```
1) [IoT]        ID: 0           Auth: true      Key: 218, 176, 75, 154, 14, 24, 230, 239, 41, 49, 38, 2, 92, 150, 119, 21
2) [IoT]        ID: 1           Auth: false
3) [IoT]        ID: 2           Auth: false
4) [IoT]        ID: 3           Auth: true      Key: 0, 155, 67, 58, 139, 137, 75, 77, 212, 84, 247, 194, 32, 139, 153, 8
5) [IoT]        ID: 4           Auth: false
6) [SERVER]     ID: 255         Authenticated IoT device(s):
                                IoT ID: 0       Key: 218, 176, 75, 154, 14, 24, 230, 239, 41, 49, 38, 2, 92, 150, 119, 21
                                IoT ID: 3       Key: 0, 155, 67, 58, 139, 137, 75, 77, 212, 84, 247, 194, 32, 139, 153, 8
```

We can see that the authentication succeeded again.

Now we press 4, to test the communication, we select IoT device number 3, and we type "Hello, CPS!" as message. Two messages will be sent through the channel.

The first message contains the string chosen by the user. It is encrypted using the previously established key t. The server receives and decrypts the message, and appends another secret string to it. The message is then encrypted again for the transmission, and is sent back to the source. Finally, the IoT device decrypts the message and displays the content.

The final decrypted message is:

```
This part was sent by IoT: Hello, CPS! | this part was added by SERVER: Return back to sender
```

This proves that the shared key t is in fact equal. If they were not, the server wouldn't have been able to correctly add the second portion of the message.

If we want to deauthenticate a device, we can send a `DEAUTH` message to the server.

```
Source: 0, Destination: 255
Payload contains 1 field(s):
1) 1
0) 0
```

The IoT device sends a short coded message with type `DEAUTH`, and is correctly interpreted by the Server as it removes the device whose ID is 0.

```
1) [IoT]        ID: 0           Auth: false
2) [IoT]        ID: 1           Auth: false
3) [IoT]        ID: 2           Auth: false
4) [IoT]        ID: 3           Auth: true      Key: 0, 155, 67, 58, 139, 137, 75, 77, 212, 84, 247, 194, 32, 139, 153, 8
5) [IoT]        ID: 4           Auth: false
6) [SERVER]     ID: 255         Authenticated IoT device(s):
                                IoT ID: 3       Key: 0, 155, 67, 58, 139, 137, 75, 77, 212, 84, 247, 194, 32, 139, 153, 8
```

5. Conclusions

This encryption algorithm is secure against the most common attacks.

The most known attack is the man in the middle. This attack assumes that the malicious entity can pose itself between the two devices, and manage both session to make the victims believe it's one. This attack is not possible because the key t is generated using two separate numbers t_1 and t_2, and the messages that share those two numbers are encrypted using the keys stored on the vault.

Another powerful class of attacks that can break AES are the side channel attacks based on power, memory and temperature analysis. The single key can technically be retrieved by the attacker, though it's impossible to retrieve the vault keys that generate it. The xor operation is just enough to protect such information. Hence, it's impossible to create a duplicate IoT device or inject false messages.

The authors of the paper conducted some tests on performance. In order to have accurate results, they needed to choose a real IoT device to test the algorithm on, because doing it on a simulator is not representative of the real scenario. They decided to use Arduino, and conducted tests based on power drawn from the supply. The tests were conducted were on which algorithm to use, to balance between execution time and energy consumed, and the dimensionality of the vault.

Algorithm	**Execution time**	**Energy consumed**
AES−128 bits	$2.5ms$	$248.75\mu J$
SHA 512	$1ms$	$99.5\mu J$
SHA 512 w/ HMAC	$1.5ms$	$149.25\mu J$
ECC	$1105ms$	$109.95mJ$
AES−256 bits	$4ms$	$398\mu J$

Table 1: Energy consumption.

Their proposed algorithm requires one AES encryption/decryption, and one HMAC operation, making the total energy consumption equal to $646.75\mu J$. This value is low compared to the ECC based authentication algorithm.

m	**n**	**bits**	**Prediction complexity**
1	128	128	1
2	128	256	2^{128}
2	256	512	2^{256}
4	128	512	$3 \cdot 2^{128}$
4	256	1024	$3 \cdot 2^{256}$
8	128	1024	$7 \cdot 2^{128}$
8	256	2048	$7 \cdot 2^{256}$

Table 2: Password prediction complexity.

Also, the authors suggest some values for parameters m and n to reach a target complexity.

Bibliography

[1] T. Shah and S. Venkatesan, "Authentication of IoT Device and IoT Server Using Secure Vaults." [Online]. Available: https://ieeexplore.ieee.org/document/8455985