DAT510 - Assignment 2

Andreas Pedersen - 265200

October 11, 2024

Abstract

This assignment presents the implementation and analysis of cryptographic security mechanisms and techniques for securing communication. The focus lies on the Diffie-Hellman key exchange, which enables secure key sharing over a public channel, and the usage of the secret key in HMAC to ensure message authenticity. Additionally, ratcheting mechanisms, both the single ratchet for chain keys and the double ratchet for Diffie-Hellman, to provide forward secrecy and post-compromise security. Through this assignment, the importance of modern cryptographic techniques for securing communication is shown.

1 Introduction

This assignment focuses on the implementation of key cryptographic protocols and mechanisms that are critical for secure communication. Concepts surrounding the Diffie-Hellman key exchange protocol and usage of the shared secret from Diffie-Hellman in Hashed Message Authentication Code (HMAC) will be studied, implemented, and discussed. Ratcheting mechanisms that provide forward secrecy and post-compromise security will also be implemented and studied. The Diffie-Hellman key exchange is a cryptographic method that allows two users to securely share encryption keys over a public channel without transmitting their conversation over the internet. The two users use symmetric cryptography to encrypt and decrypt their messages [5]. They end up with the same secret number without anyone else being able to guess it, even though a lot of the information is shared publicly. This shared secret can then be used further to encrypt their conversation. HMAC is a way to verify the integrity and authenticity of messages being exchanged. It is a cryptographic authentication technique that uses a hash function and a secret key [2]. It ensures that the message has not been tampered with and confirms the intended sender sent it. Ratcheting mechanisms, single ratchet for chain key and double ratchet for Diffie-Hellman, ensure a new key is generated for each session or message, providing forward secrecy. If one key is leaked, previous and future keys will remain secure. Double ratchet for Diffie-Hellman integrates symmetric ratcheting for the chain key with regular Diffie-Hellman key exchanges. These regular exchanges offer post-compromise security, ensuring that future messages stay protected even if previous keys were exposed.

2 Design and Implementation

2.1 Diffie-Hellman

The Diffie-Hellman key exchange protocol is implemented to secure communication between two parties. In this assignment, it is used to secure the communication between Alice and Bob. Each user agrees on shared parameters, which are a prime number p and a base g (a primitive root of the prime number). Then they both choose a private key, which is an integer. From this, they compute their public key (also an integer) as shown in Equation 1.

$$\text{public key} = g^{\text{private key}} \mod p \tag{1}$$

Further, both public keys are exchanged between Alice and Bob. Lastly, they use each other's public keys to compute the shared secret key as shown in Equation 2.

$$\text{shared secret key} = \text{public key}^{\text{private key}} \mod p \tag{2}$$

This results in the same shared secret due to the commutative property of multiplication, which means $ab{=}ba$. Therefore, both users will compute the same value S, shown in Equation 3.

$$S = g^{ab} \mod p \tag{3}$$

This ensures that both users get the same shared secret key without actually transmitting it. This logic is implemented in Python in a function called `diffie_hellman(p, g)`. The integer private keys are determined by calling `random.randint(1, p-1)` using `import random` for both Alice and Bob. Then, both public keys are determined by the *private key*, g, and p by using the function in Listing 1.

```
def mod_exp(base, exp, mod):
    return pow(base, exp, mod)
```

Listing 1: Modular exponentiation function using pow() in Python for easier readability.

For Alice, the public key would be: `alice_public = mod_exp(g, alice_private, p)`. Then shared secrets are computed for both Alice and Bob. For Alice, this would be `alice_shared_secret = mod_exp(bob_public, alice_private, p)`. After the shared secrets are computed, a check is performed at the end of the function to ensure that the shared secret is indeed the same for both Alice and Bob. If it is the same, the shared secret is returned, if not it returns the message `"Shared secret is NOT the same"`.

2.2 HMAC for Authentication

By implementing HMAC we can ensure authentic communication. Without it, an attacker could alter the message without the receiver knowing it had been changed. HMAC combines a hash function and a secret key to create a message authentication code [2]. Figure 1 shows the HMAC workflow.

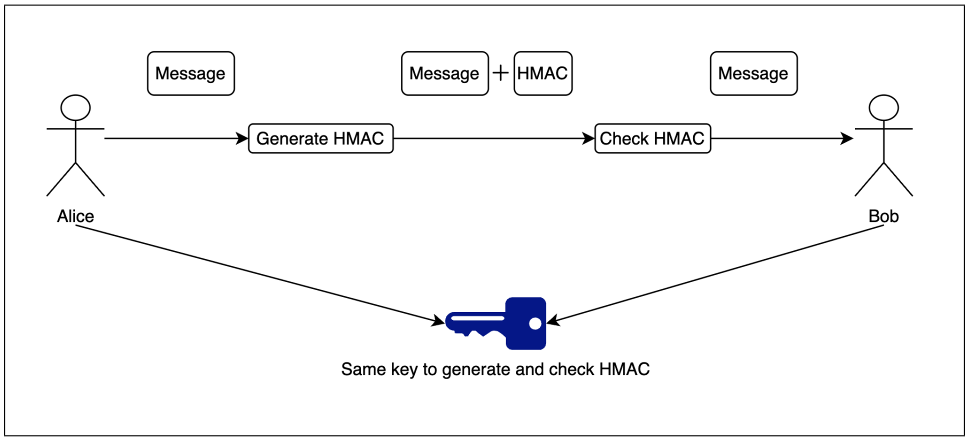


Figure 1: Hashed Message Authentication Code (HMAC) workflow

For the secret key in this implementation, we will use the shared secret from Diffie-Hellman in Sec. 2.1, and for the hash function, we use a basic XOR hash. XOR hashing is in general not a preferred hash function, but for this assignment, it is used due to its basic, fast operation and for demonstration purposes. HMAC is implemented using a function that takes a key and a message as parameters. First, the key is padded. If the key is longer than the set block size it will be hashed to create a shorter key. If the key is shorter than the block size, it is padded with zeros to match the block size. Then we add inner and outer padding, namely `ipad` and `opad`. We define these two fixed and different strings `ipad` and `opad` as follows [6]:

```
ipad = the byte 0x36 repeated to match the block size
opad = the byte 0x5c repeated to match the block size
```

Listing 2: Inner and outer padding with the bytes 0x36 and 0x5c resepectively.

After the padding shown in Listing 2, we start hashing. Both an inner and outer hash are performed using the previous padding `ipad` and `opad`. The hashing can be expressed as shown in Listing 3 [6].

```
H(K XOR opad, H(K XOR ipad, message))
```

Listing 3: Inner and outer hashing in HMAC where H is the hash function and K the key.

The XOR hashing is symmetric, so two different inputs can easily produce the same output. More about the XOR hash and its limitations in Section 4. In this HMAC implementation, we know that the XOR hash function returns only one byte as a result, therefore two hexadecimal digits from 0x00 to 0xFF [1]. The authentication tag should therefore be two hexadecimal digits in length.

Lastly, the function returns the authentication tag (which is the last outer hash) used to verify the authenticity of the messages Alice and Bob exchange. This ensures that there has been no altering of the messages.

2.3 Applying Encryption and Decryption

This section covers the encryption and decryption implementation for messages exchanged between Alice and Bob. The message, along with the authentication tag generated by HMAC (Sec. 2.2), is encrypted together in this task using Advanced Encryption Standard (AES). Encryption ensures confidentiality when Alice and Bob communicate which makes it hard for attackers to intercept and read their messages. AES is a powerful algorithm used to electronically secure sensitive data. It utilizes a secret key to scramble data into an unreadable format, making it useless without authorization. AES has been the encryption standard for the NIST since its full-scale adoption in 2002 [3]. A simple AES design is shown in Figure 2. When decrypted, the message can be read, and the authentication tag can be compared to make sure the message is authentic.

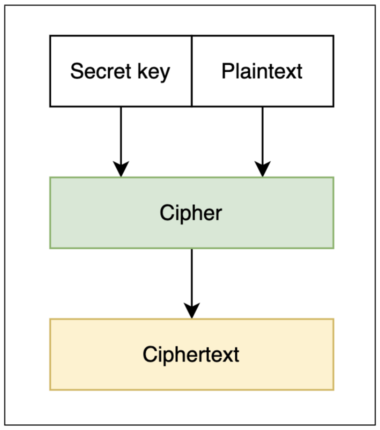


Figure 2: Simple AES design.

For the implementation, the AES library available in Python is used, and some of the code for AES encryption and decryption was provided by [4]. We use the shared secret from Diffie-Hellman for the secret key used in AES. The pseudo-code for the encryption and decryption is shown in Listing 4.

```
1. Create AES key (Diffie-Hellman shared key)
2. text = message + HMAC tag
3. Encrypt text using AES
4. Decrypt text using AES
5. Compare HMAC tags
```

Listing 4: AES encryption and decryption pseudo-code

The implemented code maintains confidentiality and integrity by encrypting and decrypting the messages and by comparing the HMAC authentication tags.

2.4 Single Ratchet for Chain Key

A single ratchet mechanism is implemented to ensure forward secrecy. This means that even if one key is leaked, previous keys remain secure. The single ratchet mechanism updates a chain key after each transmission, and that chain key is used further to derive a unique message key for each message. Since there is a creation of a new key for each sent message, we have achieved forward secrecy. However, since the new chain key is derived from the previous chain key, the security of future messages depends on the secrecy of the current chain key. If the current chain key is leaked, all future message keys derived from it could also be leaked.

The implementation for this follows a few steps shown in Listing 5.

```
1. Initialize the chain key using shared secret from
Diffie-Hellman
2. For each message do:
    2.1 Update the chain key using HMAC
    2.2 Derive a new message key with the updated chain key
    2.3 Use the message key for HMAC authentication
3. Update the chain key for future messages
```

Listing 5: Single ratchet implementation steps.

2.5 Double Ratchet for Diffie-Hellman

As the single ratchet mechanism (Sec. 2.4) provides forward secrecy, it lacks post-compromise security. To deal with this, we implement a double ratchet that combines the single ratchet with a Diffie-Hellman key exchange that continuously updates the keys in both directions. This ensures forward secrecy and post-compromise security because even if an attacker gains access to a key, they will not be able to decrypt future messages, as it is protected by the periodic Diffie-Hellman key exchanges [7].

The implementation for the double ratchet builds on the previous implementation for the single ratchet (Sec. 2.4). Listing 6 shows the steps of implementation for the double ratchet mechanism.

```
1. Initial Diffie-Hellman key pair and shared secret
2. Initial root key derived from shared secret
3. Use root key to generate first chain keys
4. After some messages, create new DH key pair
and exchange public keys
5. Derive new root key which resets chain keys
6. Update chain keys
```

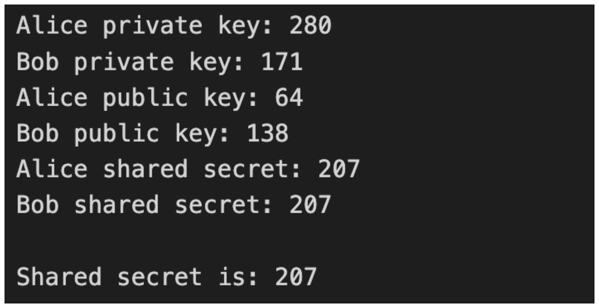
Listing 6: Double ratchet implementation steps.

3 Test Results

In this section, the results of the implemented software are presented, i.e. outputs of the code. Further discussion of these results will be performed in Sec. 4.

3.1 Diffie-Hellman Result

Output from implementation described in Sec. 2.1 is shown in Figure 3. We can see both the selected private keys for Alice and Bob, computed public keys, and finally the shared secret between Alice and Bob. Later in development, the function is changed to just output the shared secret.

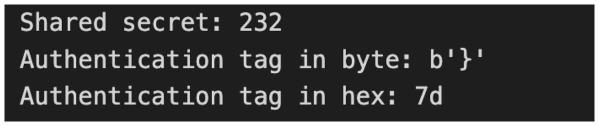


```
Alice private key: 280
Bob private key: 171
Alice public key: 64
Bob public key: 138
Alice shared secret: 207
Bob shared secret: 207

Shared secret is: 207
```

Figure 3: Diffie-Hellman implementation output.

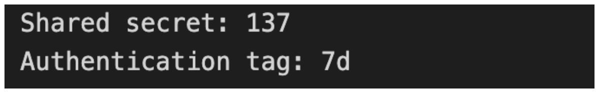
3.2 HMAC Result

Output from implementation described in Sec. 2.2 is shown in Figure 4. We can see the shared secret being used by the HMAC function and the returned authentication tag in both byte and hexadecimal format. Later in development, the function outputs exclusively in hexadecimal format, shown in Figure 5, to make it more readable.



```
Shared secret: 232
Authentication tag in byte: b'}'
Authentication tag in hex: 7d
```

Figure 4: HMAC Authentication tag in byte and hexadecimal format.

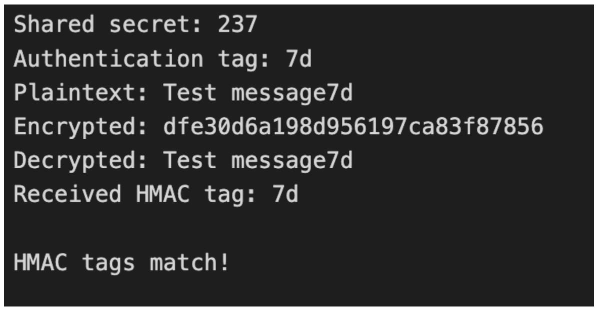


```
Shared secret: 137
Authentication tag: 7d
```

Figure 5: HMAC Authentication tag in just hexadecimal format.

3.3 Encryption and Decryption Result

Output from implementation described in Sec. 2.3 is shown in Figure 6. Here we can see shared secret and authentication tag, together with the plaintext, encrypted text, and decrypted text. A comparison of the received authentication tag and the computed one is made, and an output showing whether or not these match.

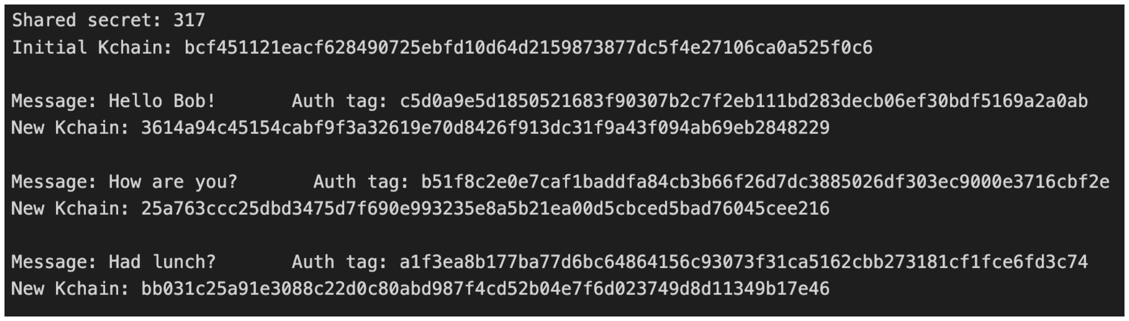


```
Shared secret: 237
Authentication tag: 7d
Plaintext: Test message7d
Encrypted: dfe30d6a198d956197ca83f87856
Decrypted: Test message7d
Received HMAC tag: 7d

HMAC tags match!
```

Figure 6: Encryption and decryption implementation output.

3.4 Single Ratchet Result

Output from implementation described in Sec. 2.4 is shown in Figure 7. From this, we can see the initial chain key, the messages being sent (with their authentication tag), and that a new chain key is generated for each message.

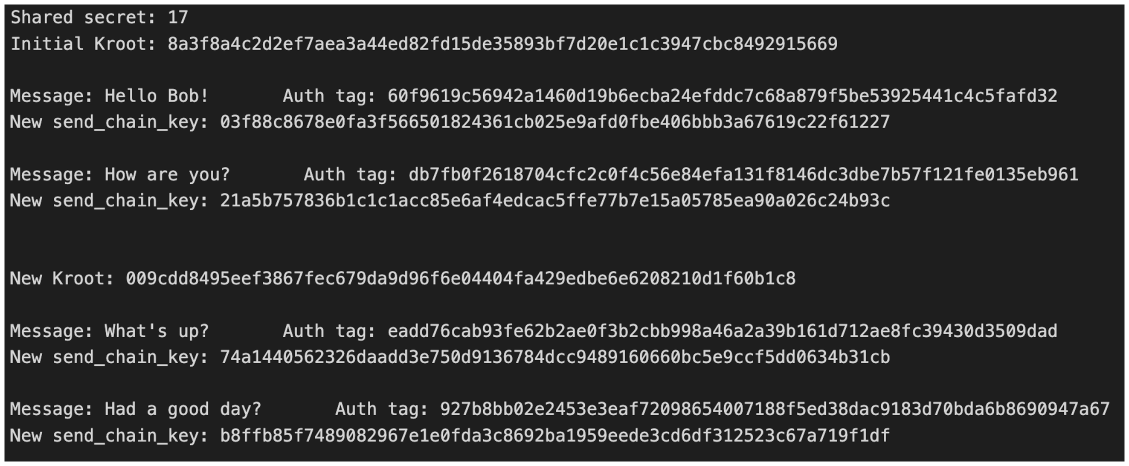


```
Shared secret: 317
Initial Kchain: bcf451121eacf628490725ebfd10d64d2159873877dc5f4e27106ca0a525f0c6

Message: Hello Bob!       Auth tag: c5d0a9e5d1850521683f90307b2c7f2eb111bd283decb06ef30bdf5169a2a0ab
New Kchain: 3614a94c45154cabf9f3a32619e70d8426f913dc31f9a43f094ab69eb2848229

Message: How are you?       Auth tag: b51f8c2e0e7caf1baddfa84cb3b66f26d7dc3885026df303ec9000e3716cbf2e
New Kchain: 25a763ccc25dbd3475d7f690e993235e8a5b21ea00d5cbced5bad76045cee216

Message: Had lunch?       Auth tag: a1f3ea8b177ba77d6bc64864156c93073f31ca5162cbb273181cf1fce6fd3c74
New Kchain: bb031c25a91e3088c22d0c80abd987f4cd52b04e7f6d023749d8d11349b17e46
```

Figure 7: Single ratchet implementation output.

3.5 Double Ratchet Result

Output from implementation described in Sec. 2.5 is shown in Figure 8. Here we can see the similarity from the single ratchet output in Figure 7. The difference now is the new root key generated each time the two users make a new Diffie-Hellman key exchange. For each message, the chain key (`send_chain_key`) is updated. After two messages, a new Diffie-Hellman key exchange is made, therefore a new root key.



```
Shared secret: 17
Initial Kroot: 8a3f8a4c2d2ef7aea3a44ed82fd15de35893bf7d20e1c1c3947cbc8492915669

Message: Hello Bob!       Auth tag: 60f9619c56942a1460d19b6ecba24efddc7c68a879f5be53925441c4c5fafd32
New send_chain_key: 03f88c8678e0fa3f566501824361cb025e9afd0fbe406bbb3a67619c22f61227

Message: How are you?       Auth tag: db7fb0f2618704cfc2c0f4c56e84efa131f8146dc3dbe7b57f121fe0135eb961
New send_chain_key: 21a5b757836b1c1c1acc85e6af4edcac5ffe77b7e15a05785ea90a026c24b93c


New Kroot: 009cdd8495eef3867fec679da9d96f6e04404fa429edbe6e6208210d1f60b1c8

Message: What's up?       Auth tag: eadd76cab93fe62b2ae0f3b2cbb998a46a2a39b161d712ae8fc39430d3509dad
New send_chain_key: 74a1440562326daadd3e750d9136784dcc9489160660bc5e9ccf5dd0634b31cb

Message: Had a good day?       Auth tag: 927b8bb02e2453e3eaf72098654007188f5ed38dac9183d70bda6b8690947a67
New send_chain_key: b8ffb85f7489082967e1e0fda3c8692ba1959eede3cd6df312523c67a719f1df
```

Figure 8: Double ratchet implementation output.

4 Discussion

The results of the implementations and the assignment as a whole show the importance of cryptographic security mechanisms and algorithms. The Diffie-Hellman key exchange protocol worked well in computing a shared key between two users, however, the prime number should be much larger in practice than what is used in this assignment. This is due to the amount of security you get against attackers. A larger prime prevents brute-forcing (because it is harder) and reverse-engineering from the publicly available information.

In the HMAC implementation, a basic XOR hash function is used. This is not a good hash function, as it is vulnerable to key collisions and tampering. As XOR is symmetric, two different inputs can easily produce the same output, as seen in Figure 4 and Figure 5 with different shared secret keys. Later, in the implementation of the ratcheting mechanisms, where HMAC also is used, the SHA-256 hash function is used instead. The SHA-256 hash function provides resistance to collisions and is a much better hash function than using XOR.

5 Conclusion

In conclusion, the implemented cryptographic security mechanisms have been very useful in making a communication more secure. This has been done by adding a secure channel to communicate through, integrity in the sent messages, confidentiality by encrypting, forward secrecy, and post-compromise security provided by ratcheting mechanisms. For future improvements, the prime number being used by the Diffie-Hellman function would be much larger to extend security, and a better hash function (e.g. SHA-256 or SHA-3) would be implemented for the HMAC function described in Sec. 2.2. In short, the future focus would be to improve security even more in the already implemented software.

References

[1] The Hexadecimal Number System Explained. https://www.freecodecamp.org/news/hexadecimal-number-system/, 2020. [Accessed 11-10-2024].

[2] HMAC (Hash-Based Message Authentication Codes) Definition. https://www.okta.com/identity-101/hmac/, 2024. [Accessed 05-10-2024].

[3] What Is Advanced Encryption Standard (AES)? https://www.pandasecurity.com/en/mediacenter/what-is-aes-encryption/, 2024. [Accessed 09-10-2024].

[4] Basile. AES Encryption & Decryption In Python: Implementation, Modes & Key Management. https://onboardbase.com/blog/aes-encryption-decryption/, 2022. [Accessed 08-10-2024].

[5] Alexander S. Gillis. What is Diffie-Hellman Key Exchange? https://www.techtarget.com/searchsecurity/definition/Diffie-Hellman-key-exchange, 2022. [Accessed 05-10-2024].

[6] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. RFC 2104: HMAC: Keyed-Hashing for Message Authentication. https://datatracker.ietf.org/doc/html/rfc2104, 1997. [Accessed 09-10-2024].

[7] Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf, 2016. [Accessed 11-10-2024].