Network Security — Exercise 2: Cryptographic Protocols

Noah Link, Jan Pfeifer, Julian Weske

May 2, 2024

1 Integrity and replay protection (Time spent: 40min)

1. Alice authenticates with the AS and receives a ticket encrypted with the public key of Alice, containing the session key (S) for the Service S. Alice sends the ticket and a nonce encrypted with her private key to the Service S. The vulnerability lies in the fact that there's no mechanism to ensure the integrity of the ticket or the nonce. An attacker could potentially intercept Alice's message to the Service S, modify the ticket or nonce, and then forward it to the Service S, impersonating Alice.

   To prevent forgery and ensure message integrity, we can use digital signatures or MACs. Here's how it could be modified:

   (a) After receiving the ticket from the AS, Alice signs the ticket with her private key. The ticket should also include a timestamp or something similar to prevent replay attacks.

   (b) Alice sends the ticket along with the signed nonce and the nonce itself to S.

   (c) The S verifies the signature on the ticket using Alice's public key. If the signature is valid, it proceeds with the authentication process.

2. The approach contains no mechanism to verify the identity of the entity receiving the response from Service S. Here's a breakdown:

   S receives the ticket and nonce from Alice. S generates a session key (K_A,S) for further communication with Alice. S sends the session key encrypted with Alice's key (K_A) to Alice. Since S is simply encrypting the session key with Alice's key and sending it back, there's no assurance that the entity receiving the session key is indeed Alice. An attacker could intercept this response and masquerade as Alice to obtain the session key.

   To prevent masquerade attacks, we can introduce a challenge-response mechanism:

   After receiving the ticket and nonce from Alice, Service S generates a random nonce and sends it to Alice. Alice receives the challenge and encrypts it with the session key (K_A,S) and sends it back to S. S verifies that the response to the challenge is correct by decrypting it with the session key. If the response is valid, Service S can be reasonably sure that the entity is Alice.

3. There is no mechanism in place for immediate revocation of Alice's access rights. Even though her access should be terminated, she may still possess the ticket and session key, allowing her to potentially access Service S until the ticket expires (if this even is a possibility). In addition, there is no provision for the AS to notify Service S about the revocation of Alice's access rights. Therefore, Service S may continue to accept Alice's requests until it explicitly checks with AS for the validity of Alice's ticket. The AS should have the ability

to immediately revoke Alice's access rights, invalidate the ticket associated with Alice, and notify Service S about the revocation. Furthermore, to invalidating the ticket, the session key (K_A, S) associated with Alice should also be invalidated to prevent any ongoing communication between Alice and Service S. Even without revocation, tickets should have a limited validity period. This helps in reducing the window of opportunity for unauthorized access in case a ticket falls into the wrong hands.

2 Cipher Malleability (Time spent: 30 min)

1. Implement Encryption and Decryption Functions:

```
def generate_key(message_length):
    return bytes([os.urandom(1)[0] for _ in range(message_length)])

def encrypt(message, key):
    if len(message) != len(key):
        raise ValueError("Message and key must be of the same length")

    encrypted_message = b""
    for m, k in zip(message, key):
        encrypted_message += bytes([(m + k) % 256])
    return encrypted_message

def decrypt(encrypted_message, key):
    if len(encrypted_message) != len(key):
        raise ValueError("Message and key must be of the same length")

    decrypted_message = b""
    for em, k in zip(encrypted_message, key):
        decrypted_message += bytes([(em - k) % 256])
    return decrypted_message
```

2. Create a function that can alter the ciphertext:

```
def alter_ciphertext(ciphertext, position, alteration):
    altered_ciphertext = bytearray(ciphertext)
    altered_ciphertext[position] = (altered_ciphertext[position] + alteration) % 256
    return bytes(altered_ciphertext)
```

3. Conduct an Experiment:

```
message = b"I like cooking, my family, and my pets."
key = generate_key(len(message))
encrypted_message = encrypt(message, key)
decrypted_message = decrypt(encrypted_message, key)
```

```
altered_position = 14
alteration_value = -12
altered_ciphertext = alter_ciphertext(encrypted_message, altered_position, alteration_value)

altered_position = 25
alteration_value = -12
altered_ciphertext = alter_ciphertext(altered_ciphertext, altered_position, alteration_value)
altered_plaintext = decrypt(altered_ciphertext, key)


Original message: b'I like cooking, my family, and my pets.'
Encrypted message: b'\x9bkf\x8f\xc8B\x7f\x08\xefi\x99\xfc\xcc\xf6\xaf\x10>\x99
                    \x82!\xf1\xdb\xdb\xbd(\x0f\\P\x81#\x14|\x89\xa2\xcaPP\xe0\xee'
Altered ciphertext: b'\x9bkf\x8f\xc8B\x7f\x08\xefi\x99\xfc\xcc\xf6\xa3\x10>\x99\
                     x82!\xf1\xdb\xdb\xbd(\x03\\P\x81#\x14|\x89\xa2\xcaPP\xe0\xee'
Altered plaintext: I like cooking my family and my pets.
```

3 Padding Oracles (Time spent: 1.5h)

1. A padding oracle attack exploits weaknesses in padding schemes, notably in algorithms like CBC. By interacting with a cryptographic system, attackers determine if a ciphertext decrypts to plaintext with valid padding. In many encryption schemes, plaintext messages must be padded to match the block size before encryption. Attackers access a system acting as a padding oracle, which checks if decrypted ciphertexts have valid padding. Crafting modified ciphertexts, attackers manipulate plaintext upon decryption, sending these to the oracle. Depending on padding validity, the oracle responds differently. By observing these responses, attackers infer plaintext information byte by byte.

2. Setup and Pseudocode:

   - Block cipher (AES) using CBC mode with PKCS#7 padding.
   - Access to an encryption oracle.

```
def padding_oracle_attack(ciphertext, oracle):
    plaintext = b"" # Initialize an empty plaintext

    # Iterate over each block of the ciphertext except the IV
    for i in range(1, len(ciphertext) // 16):
        # Initialize a list to store intermediate bytes
        intermediate_bytes = [0] * 16

        # Iterate over each byte in the block in reverse order
        for j in range(15, -1, -1):
            # Iterate over possible values for the byte
            for guess in range(256):
                # Modify the ciphertext
                modified_ciphertext = bytearray(ciphertext)
                modified_ciphertext[16 * i + j] ^= guess ^ (15 - j + 1) # Padding value guess

                # Check if the modified ciphertext is valid using the oracle
                if oracle(modified_ciphertext):
                    # Update the intermediate byte and plaintext
                    intermediate_bytes[j] = guess ^ (15 - j + 1)
                    for k in range(j + 1, 16):
                        modified_ciphertext[16 * i + k] ^= (intermediate_bytes[k] ^ (j + 2))
                    break

        # Append the recovered plaintext bytes to the plaintext
        plaintext += bytes(intermediate_bytes)

    return plaintext
)
```

3. Step by step:
   1. Initialize an empty plaintext to store the result.
   2. Loop over each block of ciphertext except the IV.

3. For each ciphertext block, initialize a list to store intermediate bytes.
4. For each byte in the block, start iterating in reverse order.
5. For each byte, try all possible values (0-255) as the guessed byte. The guessed byte is XORed with the original byte from the ciphertext, and also with the padding value (which starts from 1 and increases).
6. Modify the ciphertext by XORing the guessed byte with the byte being attacked and with the padding value.
7. Pass the modified ciphertext to the oracle function to check if the padding is valid.
8. If the padding is valid, update the intermediate byte and the rest of the plaintext bytes by XORing them with the padding value.
9. Append the recovered plaintext bytes to the plaintext.
10. Repeat this process for each block of the ciphertext.
11. Return the recovered plaintexts

4 Cryptographic algorithms (Time spent: 1h)

1. Implement Diffie Helmann by yourself:
   Our code can be found in our solution.zip in **dh.py**

2. Implement Signals Symmetric Key Ratchet:
   Our code can be found in our solution.zip in **rachet.py**
   Logging:

```
Initial constant: b'abcdef00'
Initial chain key: b'ffaabcc'

Root key after KDF:
    b':UU\xf4\xbd\xe28\x8e<\x9b\x1e\x8d\xe7\xb5\xd6\xb7\xc3D\xd8\x00\xb9\xf0\xd3‘o\x9e\xf5\x1d\x969\xf8\xe7'
Chain key after KDF:
    b'\x1d\xc0z\x99\x07\x96\xff*\xb3/\x94\x00\xb4\xa3uY\xcd\x95\xce\x91-LE\x006\x93\x1b\xd7P\x01\x96\x87'

Bob encrypts: b'Hello Alice!'
Bob's cipher text: b'\x1cp\x11\xc1\x82\xc8\xcd\xfb\xb7\x8d\x0ck'
Alice decrypts: b'Hello Alice!'

Alice encrypts: b"I'm good, thank you."
Alice's cipher text: b'1KJ\x81e|OD\xec[\x9d\xc4\x1e\x9do\x11\x1e\xce\xa2I'
Bob decrypts: b"I'm good, thank you."

Bob encrypts: b"I'm also fine, what are you up to?"
Bob's cipher text:
    b"\x1c\xdd\xe8\x0e\xa6\x0c\x04\xf0\xef'-MiI\t\xa5\xca\x97\x1di:d\xe4\x14\x00b\xdb\xef\xd7\xd8\x87\x9dl\xea"
Alice decrypts: b"I'm also fine, what are you up to?"

Alice encrypts: b"I'm working on the NetSec homework."
Alice's cipher text:
    b'.}\xe9\\Q\xc5\x1e\xbe\xb10\x05t\x87x\xac\xdc\x13\xab\xba\x8a\xc2\xc1FSC\xaa\r\xac\x7f^\xaf%G\xa7\x1a'
```

```
Bob decrypts: b"I'm working on the NetSec homework."

Bob encrypts: b'Then good luck with it!'
Bob's cipher text: b'\x1d\xe4g\xcc)zN\x03\x89\x88\x84\xa8\x12n\xd1~I\n\xda\x0e\xb8\x86\x99'
Alice decrypts: b'Then good luck with it!'
```

3. The DH ratchet in the Signal Protocol establishes shared secret keys through, ensuring forward secrecy and cryptographic security. It constantly updates keys with each message exchange, preventing decryption of past or future messages if one key is compromised, providing robust end-to-end encryption.