Cryptography

Contents

1 CBC Malleability

1.1 Attack

Cipher Block Chaining (CBC) is malleable and because we know the ciphertext, as well as the plaintext, we can change the last block from the one that we want to manipulate to the following:

$$\widetilde{C_{i-1}} = C_{i-1} \oplus P_i \oplus M$$

where C_{i-1} is the block before the one that we want to manipulate, P_i is the plaintext of the block that we want to manipulate and M is the string that we want to change the i-th block to. Therefore, if we were to manipulate block C_i, the whole message would then look like this:

$$C_1, C_2, ..., C_{i-2}, \widetilde{C_{i-1}}, C_i, ..., C_n$$

This attack will work in our case because, although the *REASON* is rendered as gibberish, that part of the transaction format is not really important because it is not processed by the code given in cbc-server.py.

1.2 Mitigation

In this case, data integrity is being violated, because the message is not consistent, i.e. it is being manipulated. Also, *AES-CBC* is not suitable for ensuring authenticity, considering the attack that we just performed. Based on our research, we would use *AES-GCM* [1, 2].

2 HMAC Length Extension Attack

2.1 Attack

When a *Merkle–Damgård* based hash is misused as a message authentication code with construction `H(secret‖message)`, and the message and length of secret are known, a length extension attack allows anyone to include extra information at the end of the message and produce a valid hash without knowing the secret [3].

This attack works because the digest of the *SHA256* hashing algorithm is also the state of it. Therefore, instead of starting with the initial state (6a09e667bb67ae853c6ef372a54ff53a510e527f9b05688c1f83d9ab5be0cd19), we can start with the value given in the exercise (69268ba87558295eedb751d8f4744b58bd2705ce5d09984f31927bb7fbfe9b97). This value is derived from the *secret* and the *message* concatenated with one another, together with the padding, and the eight bytes for the length of the secret plus the length of the message.

The *openssl* library in C++ gives us the ability to easily do this. We will therefore use C++ in order to calculate the HMAC and Python in order to manipulate the message and make requests. The Python program calls the C++ program with two parameters (total length of secret plus initial message, and the message we want to append) and reads the output of it, i.e. the HMAC, which it later uses in the requests.

The C++ source code inside blatt3/a2/sha256.cpp is based on the idea by *Ron Bowes* [4] and his *hash_extender* tool [5], as well as the source code example given by *Yola* [6]. In order to link and compile the source code and create an executable, we use the following command:

```
g++ sha256.cpp -o sha256 -lssl -lcrypto
```

Note: Because the size of the file motd.txt is 2984 bits ($0xba8$), the next hex value that ends in $0x48$ (**'H'**) is $0xc48$. We can use this to guess the length of the key: $0xc48 - 0xba8 = 0xa0$. The hex value $0xa0$ corresponds to the decimal value 160, and if we transform it from bits to bytes, we get that the key length is probably 20 bytes long (our guess is correct).

2.2 Why must the string start with the letter 'H'?

The length of the secret combined with the message located in motd.txt is equal to $20 + 373 = 393$ bytes. If we convert it to bits, we'll get $393 \cdot 8 = 3144$ bits. The hex value of 3144 is $0x0c48$ and it is appended at the end of the last block, but before our message that we want to append. Although $0x0c$ is a non-printable character, the $0x48$ part gets evaluated to the letter **'H'**.

It should be noted that in our case, the hex value $0x0c48$ is not the actual length of the message, but is only supplied so that the correct state is reached when the whole message is processed by the server. That is why it is evaluated as a character.

3 RSA Small Exponent

3.1 Observation

After extracting the public keys (n_i, e), we can see that $e = 3$ for all public keys. We can also observer that all n_i are pairwise coprime, i.e. $gcd(n_i, n_j) = 1$ $(1 \leq i, j \leq 3 \; and \; i \neq j)$.

3.2 Attack

The attack, known as the *Håstad's Broadcast Attack* [7, 8], states that if the same message m is encrypted and sent to a number of people $p_1, p_2, ..., p_k$ using the same small public exponent e and different moduli (n_i, e), then we would only need $k \geq e$ ciphertexts in order to find the plaintext m.

As we already know that the public exponent $e = 3$ for all n_i, we can find a value that satisfies all of the following equations:

$$c_1 \equiv m^3 \; mod \; n_1$$

$$c_2 \equiv m^3 \; mod \; n_2$$

$$c_3 \equiv m^3 \; mod \; n_3$$

Let's replace $C = m^3$ and use the *Chinese Remainder Theorem* in order to calculate the following:

$$C = \sum_{i=1}^{3} c_i N_i x_i$$

where $N_i = \dfrac{n_1 n_2 n_3}{n_i}$ and x_i is the inverse of N_i, $(N_i \cdot x_i \equiv 1 \; mod \; n_i)$.

Because no padding is used, the message is probably small and therefore $m < n_i$. It further applies that $m^3 < n_1 n_2 n_3$. Therefore, once we find C, we can easily find m by taking the cube root of C: $m = \sqrt[3]{C}$.

The relevant Python script main.py [9] is located inside blatt3/a3 and the output that it produces is `The answer to life the universe and everything = 42`.

4 ECDSA Fixed K

4.1 Curve parameters of public key

By using the command

```
openssl ecparam -in vk.pem -name prime256v1 -param_enc explicit | openssl
↪ asn1parse
```

we get the following output:

```
1     0:d=0  hl=3 l= 247 cons: SEQUENCE
2     3:d=1  hl=2 l=   1 prim: INTEGER           :01
3     6:d=1  hl=2 l=  44 cons: SEQUENCE
4     8:d=2  hl=2 l=   7 prim: OBJECT            :prime-field
5    17:d=2  hl=2 l=  33 prim: INTEGER
      ↪ :FFFFFFFF000000010000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
6    52:d=1  hl=2 l=  91 cons: SEQUENCE
7    54:d=2  hl=2 l=  32 prim: OCTET STRING      [HEX
      ↪ DUMP]:FFFFFFFF000000010000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
8    88:d=2  hl=2 l=  32 prim: OCTET STRING      [HEX
      ↪ DUMP]:5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B
9   122:d=2  hl=2 l=  21 prim: BIT STRING
10  145:d=1  hl=2 l=  65 prim: OCTET STRING      [HEX
    ↪ DUMP]:046B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C2964FE
    ↪ 342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5
11  212:d=1  hl=2 l=  33 prim: INTEGER
    ↪ :FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551
12  247:d=1  hl=2 l=   1 prim: INTEGER           :01
```

If we parse them to integers [10], we get the following values (the numbers in the round brackets represent the lines in the above output):

- (5) $p = 115792089210356248762697446949407573530086143415290314195533631308867097853951$
- (7) $a = -3$
- (8) $b = 41058363725152142129326129780047268409114441015993725554835256314039467401291$
- (10) $G = (48439561293906451759052585252797914202762949526041747995844080717082404635286,$
  $36134250956749795798585127919587881956611106672985015071877198253568414405109)$
- (11) $n = 115792089210356248762697446949407573529996955224135760342422259061068512044369$
- (12) $h = 1$

It should be noted that in line 10 of the output, the first hex value ($0x04$) should be ignored before parsing the hex dump. The tuple $G = (a, b)$ then gets the following hex values:

- $a =$ 0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296
- $b =$ 0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5

4.2 Difference between the two signatures

By using the command `hexdiff msg1.sig msg2.sig`, we can see that the first 39 bytes are the same. This means that both of them have the same value r (bytes 6-37 $\rightarrow$ bf..74), i.e. the first signature has the values (r, s_1) and the second signature has the values (r, s_2).

4.3 Attack

Because both signatures share the same value r, we can use this in order to calculate the ephemeral key k_E, and then calculate the private key d [11]. We start by rearanging the formulas for calculating the s_i values for both signatures in order to calculate the ephemeral key k_E. Each of the s_i values can be calculated as $s_i = (h(m_i) + d \cdot r) \cdot k_E^{-1} \ mod \ n$. By using the elimination method, we get:

$$\begin{aligned}
s_1 - s_2 &\equiv (h(m_1) - h(m_2) + \cancel{dr} - \cancel{dr}) \cdot k_E^{-1} \ mod \ n \\
s_1 - s_2 &\equiv (h(m_1) - h(m_2)) \cdot k_E^{-1} \ mod \ n \\
s_1 - s_2 &\equiv (h(m_1) - h(m_2)) \cdot k_E^{-1} \ mod \ n \\
k_E \cdot (s_1 - s_2) &\equiv (h(m_1) - h(m_2)) \ mod \ n \\
k_E &\equiv (s_1 - s_2)^{-1} \cdot (h(m_1) - h(m_2)) \ mod \ n
\end{aligned}$$

By having calculated the ephemeral key k_E, we can now calculate the private key d by, once again, using one of the formulas for calculating s_i. For this example, we will take the first one, i.e. s_1.

$$\begin{aligned}
s_1 &\equiv (h(m_1) + d \cdot r) \cdot k_E^{-1} \ mod \ n \\
k_E \cdot s_1 &\equiv (h(m_1) + d \cdot r) \ mod \ n \\
k_E \cdot s_1 - h(m_1) &\equiv d \cdot r \ mod \ n \\
d &\equiv r^{-1} \cdot (k_E \cdot s_1 - h(m_1)) \ mod \ n
\end{aligned}$$

The relevant Python code [12] for calculating the ephemeral key k_E and the private key d, as well as signing a new message m_3 is located in blatt3/a4/main.py.

4.4 Signing and verifying a new message

We have now calculated every value that we need in order to sign a new message. By using the same r value, as well as the newly calculated values k_E and d, we can create a new signature (r, s_3) for the message m_3 by calculating

$$s_3 \equiv (h(m_3) + d \cdot r) \cdot k_E^{-1} \ mod \ n$$

We can then save the new signature (r, s_3) in the file msg3.sig and execute the modified script ecdsa-openssl-verify.sh in order to verify all three signatures. As mentioned before, the Python script also signs the new message located in msg3.txt, i.e. creates the signature file msg3.sig. All of the relevant files are located inside blatt3/a4.

5 Requirements

In order to install all requirements, execute the following commands in the terminal:

```
apt install -y g++ libssl-dev libmpc-dev
pip3 install pycryptodome requests sympy gmpy2 ecdsa
```

References

1. https://datatracker.ietf.org/doc/html/rfc5084
2. https://isuruka.medium.com/selecting-the-best-aes-block-cipher-mode-aes-gcm-vs-aes-cbc-ee3ebae173c
3. https://en.wikipedia.org/wiki/Length_extension_attack
4. https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks
5. https://github.com/iagox86/hash_extender/blob/master/hash_extender_engine.c#L465
6. https://stackoverflow.com/questions/2262386/generate-sha256-with-openssl-and-c
7. https://en.wikipedia.org/wiki/Coppersmith%27s_attack#H%C3%A5stad%27s_broadcast_attack
8. http://koclab.cs.ucsb.edu/teaching/cren/project/2017/chennagiri.pdf
9. https://www.johndcook.com/blog/2019/03/06/rsa-exponent-3/
10. https://www.rfc-editor.org/rfc/rfc3279.html#section-2.3.5
11. https://blog.trailofbits.com/2020/06/11/ecdsa-handle-with-care/
12. https://github.com/bytemare/ecdsa-keyrec/blob/master/ecdsa-nonce_reuse-crack.py