Cryptography Attacks Scheme (Python part)

by Alessandro Loconsolo

Algorithms vulnerabilities:

- **Stream ciphers:** such as Salsa20, Chacha20
    - **Vulnerable to:**
        - Bit Flipping
        - Key Stream Reuse
- **Block ciphers in ECB mode:**
    - **Vulnerable to:**
        - Copy Paste
        - Adaptive Chosen Plaintext Attack (ACPA)
        - ECB or CBC mode detection
- **Block ciphers in CBC mode:**
    - **Vulnerable to:**
        - Bit flipping (partially vulnerable)
        - CBC padding oracle
        - ECB or CBC mode detection
- **Digest functions:** SHA-1, SHA-256, SHA-512, MD4, MD5 (not HMAC)
    - **Vulnerable to:**
        - Length extension attack
- **RSA:**
    - **Vulnerable to:**
        - Factorization, Fermat's Factorization, Common Modulus, Common Prime, Hastad's Broadcast, Low Public Exponent (See **How to break RSA**, below)
        - LSB Oracle

Understanding attacks

Bit flipping:

This attack consists in changing one or more bits in the ciphertext to make exact modifications in the plaintext, without actually knowing it.

Stream ciphers are particularly vulnerable to this attack because each bit of the ciphertext corresponds to the corresponding bit in the plaintext, i.e., $plaintext[i] \Rightarrow ciphertext[i]$

In block ciphers, an alteration of a single bit in the ciphertext can result in the decryption of an entirely different plaintext block due to the phenomenon of error propagation, that's why **this attack is useless with ECB**, while it is **partially possible with CBC**.

In order to perform this attack, we need to **know the exact position of the byte(s)** that we want to change in the ciphertext.

Hypothesis: the **plaintext** is "Hi, my name is Alessandro!" and it is **kept secret**, the attacker has somehow sniffed the ciphertext and somehow knows that in $i = 24$ it is stored the character 'o'.

Without knowing the text, he may want to change the letter 'o' with the letter 'a', by **creating a mask** XORing the Unicode values of the two characters. Then the mask will be XORed with the corresponding byte in the cyphertext ($ciphertext[24]$). The plaintext has indirectly changed to "Hi, my name is Alessandra!".

How is it possible with CBC?

In CBC the plaintext of a block is first XORed with the ciphertext of the previous block (or with the Initialization Vector in case of dealing with the first block) and then encrypted using the cipher (e.g., AES).

If we take the n^{th}-block in the ciphertext and we change the i^{th}-byte of that block, this will affect the whole n^{th}-block (i.e., plaintext lost) **plus the i^{th}-byte in the block $n + 1$** (bit flipping). This means that in CBC bit flipping occurs changing one byte in the previous block in order to be able to modify the correspondent byte in the next block.

The example of the professor was that we have a server in CBC mode and with bit flipping we may be able to **forge a cookie** changing the **flag admin from 0 to 1** using a mask (on the previous block and get the privileged access.

Key Stream Reuse:

This attack is particularly effective when we are dealing with stream ciphers that **encrypt** messages **using always the same key and nonce**.

The other requirement of this attack is that we have access to a **large number of cyphertexts**.

The purpose of this attack is **discovering the keystream** using a sort of brute force approach and then **recover the ciphertexts**.

Here's a step-by-step explanation:

1. Initialize a **frequency table with 256 entries** (one for each possible byte value) to zero. This table will be used to record how often each possible byte value results in a printable character when XORed with the first character of each ciphertext.
2. Loop over all possible byte values (from 0x00 to 0xFF). For each possible byte (potential keystream byte), XOR it with the first character of each ciphertext.
3. Check if the result is a printable ASCII character (typically in the range from 0x20 to 0x7E). If it is, increment the corresponding entry in the frequency table. This is based on the assumption that the plaintext is made of printable characters.
4. Once you have looped over all possible byte values, check the frequency table. The byte value that has the highest count is your best guess for the first byte of the keystream.
5. Repeat the process for each subsequent position in the ciphertexts to recover the full keystream, one byte at a time. This requires that the plaintexts are of sufficient length and that there is a sufficient number of ciphertexts.

This method works because, in a stream cipher with a reused keystream, the same keystream byte is used to encrypt the corresponding byte in each plaintext. So, if you guess that keystream byte correctly, you will be able to decrypt that byte in each ciphertext to a sensible plaintext byte (i.e., a printable character).

However, this attack is based on **statistical analysis** and **assumes** that the **plaintexts** are made up of **printable characters**. It may not work, or **may require a lot of ciphertexts**, if these conditions are not met.

The enhanced version of the attack uses statistical frequency analysis of English (or other languages) letters to improve the guessing process. Instead of just checking if a character is printable, it also checks if the character is a common English letter (or space). This can provide more accurate results, especially when there is a small number of ciphertexts.

ECB vs CBC:

The purpose of this attack is to **discover if a server encrypts the data in ECB or CBC mode**.

This attack is effective if we are **able to send a message of at least two blocks**. After sending the input, the **server** will **encrypt** it and **send the ciphertext**.

In order to discover the encryption mode, the attacker must send to the server **two identical blocks** (a block 16 byte), e.g., 32 "A" characters.

If the server responds with a **ciphertext containing 2 equal blocks, it means that the encryption was performed in ECB mode**, **otherwise in CBC**.

Copy Paste:

This is a way to **forge a cookie** for a **server** operating in **ECB mode**. There are **two servers**, **one** that **generates and encrypts the cookies** and the other **one** that **decrypts and validates** them and **checks the role** (user or admin).

This attack is effective if we **know** the **format of the cookie** generated by the server, e.g., `email=blabla@blabla.bla,role=user`, and there are **no checks on the email**.

An attacker may want to create a cookie starting from a **fake email** to **isolate the word *user*** in another block:

```
| email=aaaa,role= | userPPPPPPPPPPPP |
```

P is the padding.

He sends `aaaa` as email to the generator server, then receives the cyphertext of this cookie as a response.

Hence it will **create another cookie** with a block `| adminPPPPPPPPPPP |` as email, to get the corresponding ciphertext from the generator server.

The cookie will be something like:

```
| email=aaaaaaaaaa | adminPPPPPPPPPPP | role=userPPPPPPP |
```

Now he can **modify the first encrypted cookie**, substituting the (encrypted) *user* block with the (encrypted) *admin* block, and **finally send the new forged** (encrypted) **cookie to the authenticator** server that will give the privileged access.

Adaptive Chosen Plaintext Attack (ACPA):

The goal of this attack is **discovering the secret value** managed by **an oracle**, without having access to the key. The server operates in **ECB mode**.

The **server receives** an input message and then **returns another string**, **encrypted in ECB** mode, that contains a **fixed message**, the **input message** and the **secret**.

This attack is effective if we **know** the **fixed message** sent by the server.

The strategy is to **manipulate the input and try to guess the secret byte by byte** using the vulnerability of ECB - identical plaintexts produce identical ciphertexts.

For instance, if the server returns this encrypted message `Here is the msg:{plaintext} - and the sec:{secret}`.

The first block contains the server fixed message `Here is the msg:` (16 bytes), the other constant part, "` - and the sec:`", is 15 bytes long.

An attacker can exploit this information to construct a custom plaintext, which is then used to guess the secret one byte at a time through several iterations.

i = 0:

```
…| - and the sec:s|AAAAAAAAAAAAAAAA| - and the sec:s|ecretsecretsecrP
```

i = 1:

```
…|- and the sec:se|AAAAAAAAAAAAAAA |- and the sec:se|cretsecretsecrPP
```

i = 2:

```
…| and the sec:sec|AAAAAAAAAAAAAA -| and the sec:sec|retsecretsecrPPP
```

…

In this schema, the red part is the input message, the blue part is the fixed message from the server and the secret, and the green part are the guessed bytes.

With this example we compare the second block with the fourth block, making the nth-character of the secret appear in the last byte of the second block. When the two blocks match, it implies that the character has been guessed correctly, otherwise we try again forcing another printable character. The algorithm proceeds to the next iteration, shifting the message by one character, which allows for an additional byte to be compared each time. This process gradually uncovers each character of the secret.

CBC Padding Oracle:

This is a particular case of bit flipping attack. We have a **server** that operates in **CBC** mode. In response to a **client's request**, the **server returns** information indicating whether a **ciphertext** has a **valid** or **invalid padding**.

The **attacker**, who **knows** the **ciphertext** and the **IV**, can exploit the properties of CBC (where each block of plaintext is XORed with the previous ciphertext block before being encrypted) and knowledge of the padding's the validity of a ciphertext to **decrypt** the ciphertext **without knowing the key**.

Here's a basic outline of the padding oracle attack:

1. The attacker modifies a byte in a ciphertext block and sends it to the oracle.
2. If the oracle confirms the padding is correct, the attacker knows that the corresponding byte in the plaintext block forms a valid padding byte.
3. Using this information and the property of XOR operation, the attacker can work out the plaintext byte.
4. The attacker then moves on to the next byte, modifying it and checking the oracle's response until all bytes in the block are discovered.
5. Once the entire block is decrypted, the attacker can then move to the previous block. This is because, in CBC, the decryption of one block depends on the previous one.

To decrypt the first block, the attacker needs to manipulate the bytes of the Initialization Vector.

Length Extension:

The target of this kind of attack is a cryptographic system that utilizes **MD4 or MD5 hashing algorithms** for creating key digests. These algorithms operate by first hashing the key and then the message, and when the last block isn't completely filled, padding is applied.

In order to conduct this attack, the attacker needs to have knowledge of a **digest** (hash output) and the **length of the message** that has been hashed. This information can potentially be gathered by employing network sniffing techniques.

The objective of the attack is to **extend the original message** with additional data, **without having access to the original message**. This is accomplished by creating a new digest that represents the original message, the padding, and the appended data. Consequently, the final output appears as: original message | original padding | appended data | new padding. This process effectively adds an entire new block to the existing data.

HMAC functions are invulnerable to this kind of attack.

How to break RSA:

RSA has two public parameters e and n. By definition, n is the product of two prime factors.
If we able to obtain these factors, we can effectively break RSA, that is, we can compute the private exponent $d = e^{-1}(\text{mod}\Phi(n))$, where $\Phi(n) = (p-1)(q-1)$.
Alternatively, it's possible to decrypt a ciphertext without knowing p and q by exploiting certain vulnerabilities in the RSA algorithm under specific circumstances.
There are several attacks that can be performed to break RSA, knowing a public key.

Hastad's broadcast:

Requirements:

- We have different public keys with low exponent e (at least e keys)
- We have an RSA server that sends the same message, encrypted with all the keys, to all the recipients
- We have sniffed the ciphertexts

Result: WE OBTAIN THE PLAINTEXT

Low public exponent:

Requirements:

- e is a really small number (3, 17)
- We have a ciphertext $c < n$

Result: WE OBTAIN THE PLAINTEXT

Common modulus:

Requirements:

- We have two public keys with same modulus n, but different e
- The server sends the same message to both recipients (encrypted with their keys)

Result: WE OBTAIN THE PLAINTEXT

Note that for the three attacks above we need one or more ciphertexts and we directly obtain the ciphertext without discovering the factors p and q.

Common prime:

Requirements:

- We have two or more public keys
- The n parameters have one of the two factors in common

Implementation:

- We perform the gcd between the n parameters of the keys
- If gcd is not 1 we have found the common prime
- $n_1 = p \cdot q_1$ and $n_2 = p \cdot q_2$
- Knowing n_1, n_2 and the common prime p, it is easy to find q_1 and q_2

Result: WE OBTAIN THE FACTORS

Note: This attack is highly unlikely to succeed because a well-implemented RSA cryptosystem using a reliable random number generator will always yield distinct prime numbers. However, given the speed and simplicity of the common prime attack, it may be worth attempting if multiple keys are available and we don't know how to exploit them.

Fermat's factorization:

Requirements:

- p and q are really close numbers
- We have two moduli that share about > 50% of their most significant nums*

Note: This attack is performed on one key, if all the attacks above don't meet the conditions it may be worth attempting this algorithm, but if the two factors are not close this factorization will take an impractically long time.

*In this situation the Fermat's factorization will be immediate if we use the common part as an initial estimate of the square root for both n1 and n2.

Factorization:

Requirements:

- n must be really small (each factor should be less than 150 bits long)

Explanation: there's a database that contains the factors of known numbers, called FactorDB. If our n is small enough we may find it in the database and discover the factors. There are other approaches used to factorize, but we have studied just this one.

Note: If n is a small number, finding its factors in FactorDB will be the first attempt I would try. Otherwise, this is our last chance if we have a public key and we don't know what to do. If n is too big, finding the two factors using an algorithm will take forever.

LSB Oracle:

The **client sends a ciphertext** encrypted with the public key e **to an RSA server**. The server then decrypts the message using the private key d, and **returns** the value of the **least significant bit of the plaintext**.

An attacker, given access to the public parameters (e, n) and a ciphertext $\boldsymbol{c < n}$, can use this information to progressively narrow down the range of possible plaintext values. Note that **the original message must be encrypted using the public exponent** $\boldsymbol{e}$.

Here's a high-level description of how the attack works:

1. Initialize the upper and lower bounds of the plaintext message. At first, this will be from 0 to n (where n is the modulus in the RSA scheme).
2. Multiply the ciphertext by the encryption of 2. This effectively doubles the plaintext value (modulo n), or in other words, shifts its binary representation one bit to the left. Although, this operation does not alter the actual message, it does change its encoding.
3. Send this new ciphertext to the server, which will return the least significant bit of the decrypted plaintext.
4. Update the bounds based on the received bit:
    a. If the received bit is 1, this indicates the plaintext is greater than half the modulus. Consequently, the lower bound should be updated to the midpoint between the current upper and lower bounds.
    b. If the received bit is 0, this means the plaintext is less than half the modulus. Consequently, the upper bound should be updated to the midpoint between the current upper and lower bounds.
5. Repeat steps 2-4. Each repetition will incrementally reduce the range of possible plaintext messages. Over time, this range narrows down to such an extent that only the correct plaintext message remains.

The aim of this guide is to help you identify the various attacks discussed during the course, and understand when and why it's necessary to employ one attack method over another, based on the specific conditions and circumstances at hand. It's recommended that you supplement your understanding by viewing the video lectures, where the algorithms and tactics used are discussed in greater detail.

Implementations of all these attacks, as conducted by professor Basile, can be found in his GitHub repository, which will be accessible during the examination.