Identity Delight Provider 1 and 2

Author

Matteo Bordignon

13/04/2024

Background

Both systems are vulnerable to Chosen-Ciphertext Attacks (CCA) on RSA cryptosystem [1], more precisely to Decryption Oracle Attacks, this attack occurs when a decryption oracle is accessible. In a chosen-ciphertext attack [2], the attacker selects the ciphertext, sends it to the victim, and is given in return the corresponding plaintext or some part thereof. It is well known that plain RSA is susceptible to a chosen-ciphertext attack. An attacker who wishes to find the decryption $m \equiv c^d \pmod{n}$ of a ciphertext c can chose a random integer s and ask for the decryption of the innocent-looking message $c' \equiv s^e c \pmod{n}$. From the answer $m' \equiv (c')^d$, it is easy to recover the original message, because $m \equiv m' s^{-1} \pmod{n}$.
To mitigate the risk related to this type of attacks, this countermeasures should be applied:

- **Monitor and Limit Oracle Access**: If decryption oracles are used in cryptographic systems, restrict access to these oracles and implement monitoring mechanisms to detect and prevent potential attacks.
- **Use encrypt-then-mac approach** [3]: use a pre-shared secret to produce a message authentication code which the recipient can check *before* decrypting;
- **Use an authenticated encryption mode** [4 - 5]: is an encryption scheme which simultaneously assures the data confidentiality and integrity.

Vulnerability

The vulnerability is present because the server returns the decrypted message, which should represent an username, if the username is not present in the `known_username` list. This allows an attacker to exploit the principle explained in the previous section.

Solution

The main difference between the two exercises is the number of requests that the server accepts: in the first challenge the number of requests is unlimited, in the second challenge the server accepts only 5 requests.
However the solution showed below solves both challenges since the requests to the server are only 2.

Let's apply the principle explained above:

1. Since we have the possibility to encrypt messages as we wish, we send to the server a message to encrypt, the server will perform the encryption as $c = s^e \pmod{n}$ and it will return us c:

```
def  encrypt(msg:int) -> int:
    """
    Function that sends a msg to the server for encrytion
    @param msg (int): msg to encrypt
    @return ciphertext to integer
    """
    r.recvuntil(b' Check if a password is correct')
    r.sendlineafter(b'> ', '1'.encode())
    r.recvuntil(b'Choose a username')
    r.sendlineafter(b'> ', long_to_bytes(msg))
    r.recvuntil(b': ')
    c  =  r.recvline(keepends=False)
    return  int(c)

enc_flag = int(r.recvline(keepends=False).strip()) # the encrypted flag
s = 2
chosen_ciphertext  =  encrypt(s)  # chosen_ciphertext is our s^e (mod n)
```

2. Now we just have to multiply the encrypted flag previously returned by the system and which we already stored, times our ciphertext. Doing so we are applying the principle mentioned above: $c' = m^e \cdot s^e \pmod{n}$ where m is our flag and s our chosen message.

```
# c' = m^e * s^e (mod n), token is our c'
token =  enc_flag  *  chosen_ciphertext
```

3. The last step is send the new ciphertext to the server for decryption and divide the result by the message that we sent before. We are exploiting the decryption oracle to obtain the plaintext of the flag, mathematically speaking: $m' = (c')^d \pmod n = m^{ed} \cdot s^{ed} \pmod n$, it follows that $m = m's^{-1} \pmod n$

```python
def  decrypt(token:int) -> int:
    """
    Function that sends a ciphertext to the server for decryption
    @param token (int): token to decrypt
    @return plaintext to int
    """
    r.recvuntil(b' Check if a password is correct')
    r.sendlineafter(b'> ', '2'.encode())
    r.recvuntil(b'Insert the password')
    r.sendlineafter(b'> ', str(token).encode())
    r.recvuntil(b': ')
    plaintext  =  r.recvline(keepends=False)
    return  int(plaintext)

# m = m'/s
dec_flag  =  decrypt(token) //  s
print(long_to_bytes(dec_flag).decode())
```

Note: we don't need n to perform the modulus since the encryption and decryption operations are carried out by the server.

References

1. RSA cryptosystem
2. Bleichenbacher, D. (1998). Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: Krawczyk, H. (eds) Advances in Cryptology — CRYPTO '98. CRYPTO 1998. Lecture Notes in Computer Science, vol 1462. Springer, Berlin, Heidelberg.
3. Encrypt-then-mac
4. Authenticated encryption mode
5. AED wikipedia
6. Full solution python code