Identity Delight Provider 2

Luigi Dell'Eva, 11/04/2024.

Background

The challenge is accessible through a shell using the command `nc cyberchallenge.disi.unitn.it 50303` and allows to generate passwords that depends only on the username, to login only by providing the password. The user can perform two action: **create new password** where the user is asked to provide a username and the is prompted with the encrypted password and **check password** which prints "Password is correct!" if exists, otherwise prints the decrypted password. This is basically an **RSA oracle**. An oracle is any system that can provide additional information about a system that would otherwise be unavailable. [1] Thus, in this case even without knowing the private key, the user could decrypt messages not known to the application.

RSA Cryptography, named after its inventors Rivest, Shamir, and Adleman, is a public-key cryptosystem that uses two keys: a public key and a private key. The public key is used to encrypt data and the private key is used to decrypt it. The security of RSA relies on the difficulty of factoring the product of two large prime numbers, known as the "factoring problem" [2]

The RSA algorithm involves four steps: key generation, key distribution, encryption, and decryption. It is based on the mathematical properties of modular exponentiation and the difficulty of finding the multiplicative inverse of a number modulo another number, which is the basis for the public and private keys. The public key consists of a modulus (`n`) and a public exponent (`e`), typically a prime number such as 65537, while the private key consists of the modulus (`n`) and the private exponent (`d`). The modulus (`n`) is the product of two large prime numbers (`p` and `q`). [3]

RSA has numerous vulnerabilities, including susceptibility to side-channel attacks, inadequate key lengths, flaws in prime number generation, fault-based attacks, and risks arising from stolen or lost keys. [4] For example, **Hastad's broadcast**, **LSB oracle (parity)**, **Wiener's** and it is also **malleable**. [5] To mitigate this vulnerability it is important to follow some rules such as using a strong random number generator, avoid weak prime number, use minimum key length of 2048 bits and others. [4]

Vulnerability

The fact that the application is an **oracle** allows to exploit of the malleability of RSA. RSA is said to be **malleable**. A cryptographic algorithm is considered malleable if an attacker can modify a ciphertext `c` in a manner that leads to the generation of a related plaintext, even without knowledge of the original plaintext. [6] Given some ciphertext `c=p^e` for any chosen `s` we may craft a new ciphertext `c'=c*s^e (mod n) = (ps)^e (mod n)` which is also a valid ciphertext.

Solution

The fact that the application is an oracle allows to exploit the malleability of RSA, by forging a new ciphertext (not known to the application) that will decrypt to a valid plaintext. The idea is to multiply the ciphertext `c` by a chosen value `s^e` and send it to the oracle. The oracle will decrypt the new ciphertext `c'` and return the plaintext `p'`. The attacker can then compute the plaintext `p` by calculating the modular inverse of `s` and multiplying it by `p'`.

The problem here is that to perform encryption and subsequent an eventual module to reduce the dimension of the plaintext we need to know the public key. The exponent `e` is known since its used the default value (65537) but the modulus `n` is unknown. To solve this problem I found a script online [7] which basically looks for a common factor between some given pairs of plain and ciphertext.

The script is the following:

```python
#!/usr/bin/env python3
from Crypto.Util.number import bytes_to_long, long_to_bytes, inverse
from pwn import *
from gmpy2 import *

conn = remote('cyberchallenge.disi.unitn.it', 50303)
#conn = process(['python3', 'challenge.py'])

conn.recvuntil(b'example password: ')
c = int(conn.recvline().strip())

def encrypt(m):
    conn.sendlineafter(b'> ', b'1')
    conn.sendlineafter(b'> ', str(m).encode())
    conn.recvuntil(b': ')
    return int(conn.recvline().strip())

list = [(bytes_to_long("2".encode()), encrypt(2)), (bytes_to_long("3".encode()), encrypt(3)),
(bytes_to_long("5".encode()), encrypt(5)), (bytes_to_long("7".encode()), encrypt(7))]

def recover_n(pairings, e):
    pt1, ct1 = pairings[0]
    N = ct1 - pow(pt1, e)

    # loop through and find common divisors
    for pt,ct in pairings:
        val = gmpy2.mpz(ct - pow(pt, e))
        N = gmpy2.gcd(val, N)

    return int(N)

e = 65537
N = recover_n(list, e)

# Craft c' = s^e * c mod n
s = 2
c_tampered = pow(s, e, N)*c

conn.sendlineafter(b'> ', b'2')
conn.sendlineafter(b'> ', str(c_tampered).encode())
conn.recvuntil(b': ')
m_prime = int(conn.recvline().strip())

# Extract the flag -> m = m' * s^-1 mod n
p = inverse(s,N)*m_prime%N
# decode the plaintext as string
p = long_to_bytes(p).decode()
print(p)
```

References

[1] Cryptographic Oracle: https://security.stackexchange.com/questions/10617/what-is-a-cryptographic-oracle

[2] RSA Cryptography https://en.wikipedia.org/wiki/RSA_(cryptosystem)

[3] RSA algorithm https://www.techtarget.com/searchsecurity/definition/RSA

[4] RSA mitigation https://www.splunk.com/en_us/blog/learn/rsa-algorithm-cryptography.html

[5] Ethical Hacking course slides

[6] RSA malleability https://medium.com/@prathyusha756/rsa-crypto-system-921ae3129c20

[7] Modulus recovery script: https://cryptohack.gitbook.io/cryptobook/untitled/recovering-the-modulus