Nicola Mores, 14/04/2024

Challenge: Identity Delight Provider 1 - 2

Background

These two challenges are about the asymmetric cryptography algorithm RSA. The system we are working on allows the users to use the two main functions of a cryptography algorithm: encrypt: after providing a username as plaintext, users will receive a password obtained from its encryption. The selected username will also be stored in a list of known usernames; decrypt: used in order to decrypt a ciphertext and check if the resulting plaintext was previously known. If this is not the case, the system will also provide to the user the obtained plaintext, otherwise it will just confirm the validity of the password. This particular implementation of RSA is vulnerable to malleability: a (usually unwanted) property of cryptosystem that allows attackers to modify the ciphertext in order to get a change in the resulting plaintext, thus violating the integrity of the encrypted message.

Vulnerability

In this case the vulnerability arises from the implementation of RSA, usually referred to as textbook RSA, which allows us to use its malleability in order to create a decryption oracle and thus obtain the secret flags. This type of attack could be prevented by using padding [1], such as PKCS#1: a technique in which we add some randomized data to the plaintext, in order to make every encryption of the same plaintext look different from one another.

Solution (Valid for both the challenges)

In order to complete this challenge we have to understand how RSA works and how it is possible to exploit its malleability in order to “decrypt” in a non conventional way the flag (presented to us after connecting to the server) through a chosen ciphertext attack [2, 3]. First of all let’s focus on the encrypted flag: let’s call the received ciphertext “*c*”, that formally results to $c = s^e \bmod n$ (the usual formula to encrypt in RSA; $s = c^d \bmod n$ is the decryption one), where *s* is the secret flag and *e* (= 65537) together with *n* are the two elements of the RSA public key (while the secret one is made of *n* and *d*). Our goal is to obtain the secret *s*. The server is implemented in order to not print the decrypted secret if it was previously encrypted through its function, so we won’t be able to obtain *s* using *c*, but we will have to handcraft a new variable, let’s call it “*C*”, that is somehow bound, through a reversible operation like a multiplication, to *c* and that after decryption will let us retrieve the secret *s*. Formally, we want a *C* for which $C^d = (\alpha \cdot c)^d = \alpha^d \cdot c^d = \alpha^d \cdot s^{de} = \alpha^d \cdot s$ holds (Remember that $\forall x,\ x^{de} = x^{ed} = x$ always holds due to how we defined *e* and *d*). If we had such *C*, we could just decrypt it using the server’s function, obtaining a new plaintext $m = \alpha^d \cdot s$ and then divide it by α^d, thus obtaining *s*. We now need a proper *α* to get a *C* for which the equation previously stated holds. Since we need to work with α^d, we are looking for an *α* such that α^d is a known value. This can be obtain by defining another variable, let’s call it *β*, and setting *α* as the value obtained by encrypting *β*, formally $\alpha = \beta^e$. In this way the decrypt of *C* will produce $m = C^d = (\alpha \cdot c)^d = \beta^{de} \cdot s^{de} = \beta \cdot s$. Once we have received this *m*, *s* is easily computable as $s = \frac{m}{\beta}$. In the following code [4] we use this idea, setting *β* = 2 and properly handling the conversion between bytes and integers.

```python
# Retrieve the encrypted Flag c = s^e mod n
conn.recvuntil(b":")
flag = int((conn.recvline(keepends=False)).decode())
print("Encrypted Flag (c): ", flag)
c = flag

# Set parameters used for the malleability attack
e = 65537
betha = 2

# Get alpha = betha^e mod n
conn.recvuntil(b"> ")
conn.sendline(b"1")

conn.recvuntil(b"> ")
conn.sendline(long_to_bytes(betha))

conn.recvuntil(b":" )
alpha = int(conn.recvline(keepends=False).decode())

print("Alpha: ", alpha)

# Compute C = c * alpha
C = c * alpha
print("C: ", C)

# Send C to the server
conn.recvuntil(b"> ")
conn.sendline(b"2")

conn.recvuntil(b"> ")
conn.sendline(str(C).encode())

# Get m from the server. m = C^d mod n = (c * alpha)^d mod n = s * betha
conn.recvuntil(b": ")
m = int(conn.recvline(keepends=False).decode())

print("M: ", m)

# Compute s from m. Notice how we use // (rather than / followed
# by a cast) in order to get a higher precision in the division
s = m // betha
print("S: ", s)
print("Flag: ", long_to_bytes(s).decode())
```


[1] RSA Padding Schemas: https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Padding_schemes
[2] Chosen Ciphertext Attack: https://en.wikipedia.org/wiki/Chosen-ciphertext_attack
[3] Paper - A chosen text attack on the RSA cryptosystem and some discrete logarithm schemes: https://www-users.cse.umn.edu/~odlyzko/doc/arch/rsa.attack.pdf
[4] Code Example: https://gitfront.io/r/NikoMrs/PrSpp9XjTGu2/EthicalHacking2024/blob/Assignment4/solution1.py