Background

Modern cryptographic systems rely on mathematical principles to safeguard sensitive data. Public key cryptography, and in particular RSA [6], is widely used because it provides a means to securely encrypt and decrypt messages using separate keys for public sharing and private decryption. However, this widely system contains certain properties that can lead to vulnerabilities.

One property is **multiplicativity**. In its basic form, RSA encryption is defined as $c = m^e \ mod \ n$. Due to the multiplicative nature of exponentiation modulo n, the encryption function has the homomorphic property such that for any two plaintexts m_1 and m_2, the product of their encryptions is: $E(m_1) * E(m_2) \equiv (m_1 m_2)^e \ mod \ n$.

This makes the system vulnerable to multiplicative attacks if the protocol is not protected with appropriate countermeasures such as padding schemes (e.g. OAEP [5]) or additional integrity checks.

Integrity protection is a critical aspect of cryptographic protocols because even if an encryption scheme is mathematically strong, it remains vulnerable if the ciphertexts can be modified without detection. In many systems, integrity is maintained through the use of Message Authentication Codes [4] or authenticated encryption modes such as AES-GCM [3]. These ensure that any modifications to the ciphertext or associated data will be detectable at decryption [1].

However, when integrity checks are implemented using hash-based methods, there are several problems. For example, bcrypt [2] processes only the first 72 bytes of input. If integrity tokens are constructed by concatenating various fields and then hashed with bcrypt, an attacker can exploit this truncation. Any data beyond that limit will be ignored in the resulting hash.

The design of integrity checks must consider the limitations of the hash functions in use. Failure to account for these pitfalls can open the door to attacks that compromise both the confidentiality and the integrity of the system.

Vulnerability

The vulnerability in the first challenge relies on the decryption oracle that the program performs, enabling for a Chosen Ciphertext Attack (CCA) using this oracle and RSA malleability. The second version is basically the same as the first, with the added integrity through hash that can be bypassed by exploiting bcrypt's limitations.

Solution

The exploit is executed in two main stages

Recovering RSA Modulus n By sending known plaintexts (e.g. single characters) to obtain the ciphertexts, the value $r_i = p^e - c_i$ can be computed, which is a multiple of `n`. Taking the *gcd* between multiple of these values obtained with different plaintexts, `n` can be recovered.

Multiplicative Attack Once `n` is known, a small multiplier is chosen and its encryption computed. Then, the ciphertext of the flag gets modified: $c^{'} = (c_{flag} * s^e) \ mod \ n$. When the server decrypts $c^{'}$, it yields $s * FLAG$, from which the flag can be obtained by dividing this result by s.

For the second version of the challenge, the integrity check is performed via a hash using bycrypt, that can be bypassed via bycrypt's truncation at 72 bytes. By providing a name of exactly 72 bytes during both the encryption (protect) and decryption (show) operations, only this exact name is hashed, regardless of any changes in the appended ciphertext. This allows the modified ciphertext to pass the integrity check and be successfully decrypted.

```python
from Crypto.Util.number import long_to_bytes
from math import gcd
from pwn import remote

def get_cipher(r: remote, message: bytes, name=None) -> int:
    r.sendlineafter(b'> ', b'1')
    if name: r.sendlineafter(b'> ', name)
    r.sendlineafter(b'> ', message)
    r.recvuntil(b': ')
    return int(r.recvline(keepends=False))

e = 65537
def compute_n(ciphers: dict[bytes, int]) -> int:
    # r_i = p^e - (p^e mod n) are multiples of n, so their gcd is n or
    # a multiple of it -> the more values the better
    # r_i is congruent to 0 mod n
    rs = [(pow(ord(p), e) - c) for p,c in ciphers.items()]
    return gcd(*rs)

def multiplicative_attack(r: remote, n: int, c_flag: int, name=None) -> int:
    s = 2                           # small multiplier
    s_e = pow(s, e, n)              # encryption of s
    c_prime = (c_flag * s_e) % n   # modified ciphertext: c' = (c_flag * s^e) mod n

    r.sendlineafter(b'> ', b'2')
    if name: r.sendlineafter(b'> ', b'A' * 72)
    r.sendlineafter(b'> ', str(c_prime).encode())
    r.recvuntil(b': ')

    m_prime = int(r.recvline(keepends=False)) # m' = s * FLAG
    return m_prime // s                        # s * FLAG < n -> simply divide

# ======== First version ========
r1 = remote('cyberchallenge.disi.unitn.it', 50102)
r1.recvuntil(b': ')
c_flag = int(r1.recvline(keepends=False))
ciphers = {}
for p in [b'A', b'B', b'C']:
    ciphers[p] = get_cipher(r1, p)

n = compute_n(ciphers)
int_flag = multiplicative_attack(r1, n, c_flag)
print(long_to_bytes(int_flag).decode())

# ======== Second version =======
r2 = remote('cyberchallenge.disi.unitn.it', 50103)
r2.recvuntil(b': ')
c_flag = int(r2.recvline(keepends=False))

# bcrypt only uses the first 72 bytes of the input to the hashing function
# if two integrity tokens share the same first 72 bytes, they will have the same hash
# integrity_token = name.encode() + long_to_bytes(encrypted)
name = b'A' * 72

ciphers = {}
for p in [b'A', b'B', b'C']:
    ciphers[p] = get_cipher(r2, p, name)

n = compute_n(ciphers)
int_flag = multiplicative_attack(r2, n, c_flag, name)
print(long_to_bytes(int_flag).decode())
```

Listing 1: Solution code

References

[1] GeeksForGeeks. Message Integrity. https://www.geeksforgeeks.org/message-integrity-in-cryptography/.

[2] Wikipedia. Bcrypt. https://en.wikipedia.org/wiki/Bcrypt.

[3] Wikipedia. GCM. https://en.wikipedia.org/wiki/Galois/Counter_Mode.

[4] Wikipedia. MAC. https://en.wikipedia.org/wiki/Message_authentication_code.

[5] Wikipedia. OAEP. https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding.

[6] Wikipedia. RSA Cryptosystem. https://en.wikipedia.org/wiki/RSA_cryptosystem.