CRY 2024

Laboratoire #3

June 2024 - Nathan Rayburn

1 RSA OAEP

1.1 Introduction

This challenge concerns a poorly RSA-OAEP implementation. Our challenge is to crack a cipher which is an encrypted grade. The information that we are given is that the grades are between included 0 and 6 included with 0.1 intervals. We have access to the cipher and the public key (e, n).

1.2 Cracking RSA OAEP (poorly implemented)

The first thing I did was to compare the standard RSA-OAEP implementation to the one in the code. I printed the length of the `masked_db`, which was 254 bytes. Next, I checked the seed size, which we can calculate with the constant value of 1. The last byte missing is "00" to detect the standard. My initial intuition was to use brute force because the possible seed values were low.

Putting everything together, we got 256 possible seeds and 61 possible grades (0 is included).

To calculate the maximum number of attempts needed for the brute force in the worst case, we multiply the number of possible seeds by the number of possible grades:

$$256 \text{ seeds} \times 61 \text{ grades} = 15616 \text{ attempts}$$

Therefore, in the worst case, we would need to make a maximum of 15,616 attempts to brute force the decryption.
For the corresponding cipher text:

```
c = 37804545152788535125689063008... (way too long)
```

Values we need to be able to encrypt:

e = 65537
n = 971874506552385452773065988 4... (again, very long)

We are able to brute force and find the corresponding grade with the same cipher.

1.3 Code used for the attack

Here is the brute force attack code used, with comments explaining each line:

```python
# C : is the cipher of the grade to find
# E : is the public key
# N : is our domain we're working in
def bruteForceAttack(c, e, n):
    print(f"Searching for {c}")

    # parse each possible seed
    for i in range(256):
        # parse each possible grade
        for j in range(61):

             # Calculate grade to test
            grade_to_test = round(j * 0.1, 1)

            # Convert to string, then to bytes
            grade_to_test_bytes = str(grade_to_test).encode()

            print(f"Grade to test {grade_to_test}")
            try:
                # Encrypt the data
                find = reversed_RSA_AEP(grade_to_test_bytes, e, n, i)
                print(f"Current Grade: {c}")
                print(f"Current find : {find}")
                if c == find:  # Check both ciphers matches
                    print(f"Found grade: {grade_to_test}")

                    # return the matching grade
                    return grade_to_test
            except ValueError as e:
                print(f"Error encrypting grade {grade_to_test}: {e}")
```

Listing 1: Brute Force Attack Code

We were able to find the grade that was encrypted: 4.9

1.4 Patch the vulnerability

The vulnerabilities in the implementation are as follows:

- The seed is encoded randomly on one byte, providing only 256 possible seeds.

- The data we are trying to crack has a small search area, making it an easy search. The data can only be in 61 different states.

Suggested patch: To address this vulnerability, we should adhere to the RSA-OAEP standard. The seed size is not implemented properly and should be a lot bigger. According to the standard, it's recommended to make it the same size as the HASH function output, which in our case would be 256 bits or 32 bytes (SHA-256). This means we would have to make the message smaller, down to 190 bits in our current implementation. A reliable way to avoid such errors would be to use a cryptographic library such as PyCryptodome and following the current standards for the key sizes etc... This way we can reduce the chance of potential human error.

```
We can find the following required seed length below.
In RFC 80017 section 7.1.1 step 2.d:
```

> "d. Generate a random octet string seed of length hLen."

2 Signature RSA-CRT corrupted

2.1 Introduction

This challenge concerns a simulated implementation of a physical bug that occurs during the signature RSA-CRT (Chinese Remainder Theorem). (The signatures aren't valid at all with the implementation). To win the challenge, we need to find the private key that is used.
Hint : For x, x', if $x \mod p = x' \mod p$ but $x \mod q = x' \mod q$, what can be said about $x - x' \mod p$? And about $x - x' \mod q$? Deduce something about $x - x' \mod n$.

2.2 Breakdown

To answer the hint:

$$(x - x') \mod (p \cdot q) = [((x - x') \mod p), ((x - x') \mod q)]$$

And

$$x \mod p = x' \mod p$$

This means that $x - x' \mod p$ must be equal to 0. Therefore, $x - x'$ equals to a factor $\lambda \cdot p$. Therefore, we can factorise and find the greatest common divider between $\lambda \cdot p$ and N which would give us p logically. Reminder $N = p \cdot q$

$$\gcd(x - x', p \cdot q) = p$$

This seems nice because we have every variable to solve the equation. $p \cdot q = N$, N is public. Next step is to find what x and x' correspond to.

Theory of signature validation : $m' = s^e \mod n$.

Since our signature isn't valide, this will represent us the x' from our previous equations.

We can define m which is supposed to be hashed with SHA-256.

Our equivalent to find p,

$$\gcd(\text{hash}(m) - m', n) = p$$

2.3 Cracking RSA-CRT Signature

Here is a quick walk-through by using the logic from the breakdown above:

I implemented the exploit to find p. This means from p, we are able to find the private key. We have $N = p \cdot q$, so $q = \frac{N}{p}$, which is useful to find $\phi(N)$.

$$\phi(N) = (p-1) \cdot (q-1)$$

We know that the private key is the exact inverse modulo of the public key. Therefore,

$$d = e^{-1} \mod \phi(N))$$

Finally, we can simply test by creating a working signature that isn't bugged and see if the signature is validated. I've double checked that my signature hack was working properly by generating my own keys and seeing if the function was able to crack the private keys.

```python
def hackSignature(e, n, m, s):

    # find corrupted m'
    mprime = power_mod(s, e, n)
    # message hash which corresponds to our x
    h = int.from_bytes(sha256(m).digest(), byteorder="big")

    # exploiting the vulnerability
    p = gcd(h - mprime, n)
    # Calculate Q with P from N
    q = n / p
    # Calculate the private key, phi(N) = (p - 1) * (q - 1)
    d = power_mod(e, -1, (p - 1) * (q - 1))

    # Create the real signature
    test_signature = signatureWorking(m, d, p, q, n)

    # Validate the signature
    if validateSignature(m, test_signature, e, n):
        print("Signature ok")
    else:
        print("Signature nok")
'''
This function is designed to validate a signature
'''
def validateSignature(m,s,e,n):
    # message hash
    h = int.from_bytes(sha256(m).digest(), byteorder = "big")
    # calculate message hash from signature
    mprime = power_mod(s,e,n)
    # check if the message hash corresponds with the signature
    if mprime == h:
        return true
    return false
```

Listing 2: Implemented hack

3 ECDSA

3.1 Introduction

This challenge is about ECDSA, which stands for
Elliptic Curves Digital Signature Algorithm.
The challenge implements a vulnerability which uses a random that isn't so random...
To win the challenge, we need to find the private key.

3.1.1 ECDSA parameters - reminder

- Choose a cryptographically secure elliptic curve and a point G of order n.
- Private key: $a \in \{1, \ldots, n-1\}$.
- Public key: $A = aG$.

3.1.2 ECDSA signature - reminder

To sign a message M:

1. Generate a secret number $k \in \{1, \ldots, n-1\}$ uniformly at random.
2. Calculate $(x_1, y_1) = kG$.
3. $r = x_1 \mod n$.
4. $s = \frac{H(M)+ar}{k} \mod n$.
5. The signature is (r, s) if $r \neq 0$ and $s \neq 0$. Otherwise, start over.

3.2 Break down

What we are interested is how the signature is made and how the parameters are defined like above. Below we can see in the implementation that *ctr* is suppose to represent our secret number k which isn't at all random like it should. Therefore, we can mathematically find the private key since k is deterministic if we have two signatures that are done with the same private key.

```
1 def sign(G, n, a, ctr, m):
2     ctr = alpha*ctr+beta # needs to be random... Alpha and Beta are
       known
3     rest of the code...
```

Listing 3: Implemented signature algorithm

3.3 Cracking ECDSA

3.3.1 Defined constants

$$
\begin{aligned}
m1 &= \text{b"Welcome to the CRY class"} \\
m2 &= \text{b"We will do maths, maths, and maths!"} \\
\alpha &= 13 \\
\beta &= 17 \\
(r1, s1, ctr) &= \text{sign}(G, n, a, ctr, m1) \\
(r2, s2, ctr) &= \text{sign}(G, n, a, ctr, m2)
\end{aligned}
$$

3.3.2 Prepare our equations

To isolate k in the equation $s1 = \frac{h(m1)+a \cdot r1}{k}$, follow these steps:
1) Multiply both sides of the equation by k to eliminate the denominator:

$$s1 \cdot k = h(m1) + a \cdot r1$$

2) Then, divide both sides of the equation by $s1$ to isolate k:

$$k = \frac{h(m1) + a \cdot r1}{s1}$$

To isolate k in the equation $s2 = \frac{h(m2)+a \cdot r2}{\alpha \cdot k+\beta}$, follow these steps:
1) Multiply both sides of the equation by $\alpha \cdot k + \beta$ to eliminate the denominator:

$$s2 \cdot (\alpha \cdot k + \beta) = h(m2) + a \cdot r2$$

2) Expand the left side of the equation:

$$s2 \cdot \alpha \cdot k + s2 \cdot \beta = h(m2) + a \cdot r2$$

3) Isolate the term containing k by subtracting $s2 \cdot \beta$ from both sides of the equation:

$$s2 \cdot \alpha \cdot k = h(m2) + a \cdot r2 - s2 \cdot \beta$$

4) Divide both sides of the equation by $s2 \cdot \alpha$ to isolate k:

$$k = \frac{h(m2) + a \cdot r2 - s2 \cdot \beta}{s2 \cdot \alpha}$$

3.3.3 Putting together the equations

1) Our equations we have calculated with k isolated :

$$k = \frac{h(m1) + a \cdot r1}{s1}$$

$$k = \frac{h(m2) + a \cdot r2 - s2 \cdot \beta}{s2 \cdot \alpha}$$

2) Set the two expressions for k equal to each other:

$$\frac{h(m1) + a \cdot r1}{s1} = \frac{h(m2) + a \cdot r2 - s2 \cdot \beta}{s2 \cdot \alpha}$$

3) Cross-multiply to eliminate the denominators:

$$(h(m1) + a \cdot r1) \cdot (s2 \cdot \alpha) = (h(m2) + a \cdot r2 - s2 \cdot \beta) \cdot s1$$

4) Distribute on both sides:

$$s2 \cdot \alpha \cdot h(m1) + s2 \cdot \alpha \cdot a \cdot r1 = s1 \cdot h(m2) + s1 \cdot a \cdot r2 - s1 \cdot s2 \cdot \beta$$

5) Collect all terms involving a on one side of the equation and the remaining terms on the other side:

$$s2 \cdot \alpha \cdot a \cdot r1 - s1 \cdot a \cdot r2 = s1 \cdot h(m2) - s2 \cdot \alpha \cdot h(m1) - s1 \cdot s2 \cdot \beta$$

6) Factor out a on the left side:

$$a \cdot (s2 \cdot \alpha \cdot r1 - s1 \cdot r2) = s1 \cdot h(m2) - s2 \cdot \alpha \cdot h(m1) - s1 \cdot s2 \cdot \beta$$

7) Finally, solve for a:

$$a = \frac{s1 \cdot h(m2) - s2 \cdot \alpha \cdot h(m1) - s1 \cdot s2 \cdot \beta}{s2 \cdot \alpha \cdot r1 - s1 \cdot r2}$$

3.4 Implemented Code to Crack the Challenge

The following code demonstrates the implementation to crack the challenge:

```
a = ((s1 * h(m2) - s2 * alpha * h(m1) - s1 * s2 * beta) / (s2 *
    alpha * r1 - s1 * r2 )) % n

(r3, s3, ctr) = sign(G, n, a, ctr, mchall)

print(verify(mchall, A, r3, s3, G, n))  # Outputs: True
```