←Home Posts Tags About

CSAW CTF 2019

Sep 16, 2019 · 1618 words · 8 minute read

I'm going to explain my writeup for some challenges that I have done in this year CSAW CTF.

Crypto

Fault Box

Below is the given chall

```python
import socketserver
import random
import signal
import time
import gmpy2
from Crypto.Util.number import inverse, bytes_to_long, long_to_bytes

FLAG = open('flag', 'r').read().strip()


def s2n(s):
    return bytes_to_long(bytearray(s, 'latin-1'))


def n2s(n):
    return long_to_bytes(n).decode('latin-1')


def gen_prime():
    base = random.getrandbits(1024)
    off = 0
    while True:
        if gmpy2.is_prime(base + off):
            break
        off += 1
    p = base + off

    return p, off


class RSA(object):
    def __init__(self):
        pass

    def generate(self, p, q, e=0x10001):
        self.p = p
        self.q = q
        self.N = p * q
        self.e = e
        phi = (p-1) * (q-1)
```

```
        self.d = inverse(e, phi)

    def encrypt(self, p):
        return pow(p, self.e, self.N)

    def decrypt(self, c):
        return pow(c, self.d, self.N)

    # ===== FUNCTIONS FOR PERSONAL TESTS, DON'T USE THEM =====
    def TEST_CRT_encrypt(self, p, fun=0):
        ep = inverse(self.d, self.p-1)
        eq = inverse(self.d, self.q-1)
        qinv = inverse(self.q, self.p)
        c1 = pow(p, ep, self.p)
        c2 = pow(p, eq, self.q) ^ fun
        h = (qinv * (c1 - c2)) % self.p
        c = c2 + h*self.q
        return c

    def TEST_CRT_decrypt(self, c, fun=0):
        dp = inverse(self.e, self.p-1)
        dq = inverse(self.e, self.q-1)
        qinv = inverse(self.q, self.p)
        m1 = pow(c, dp, self.p)
        m2 = pow(c, dq, self.q) ^ fun
        h = (qinv * (m1 - m2)) % self.p
        m = m2 + h*self.q
        return m


def go(req):
    r = RSA()
    p, x = gen_prime()
    q, y = gen_prime()

    r.generate(p, q)
    fake_flag = 'fake_flag{%s}' % (('%X' % y).rjust(32, '0'))

    def enc_flag():
        req.sendall(b'%X\n' % r.encrypt(s2n(FLAG)))

    def enc_fake_flag():
        req.sendall(b'%X\n' % r.encrypt(s2n(fake_flag)))

    def enc_fake_flag_TEST():
        req.sendall(b'%X\n' % r.TEST_CRT_encrypt(s2n(fake_flag), x))

    def enc_msg():
        req.sendall(b'input the data:')
        p = str(req.recv(4096).strip(), 'utf-8')
        req.sendall(b'%X\n' % r.encrypt(s2n(p)))

    menu = {
        '1': enc_flag,
        '2': enc_fake_flag,
        '3': enc_fake_flag_TEST,
        '4': enc_msg,
    }

    cnt = 2
    while cnt > 0:
        req.sendall(bytes(
            '====================================\n'
            '            fault box\n'
```

```
            '====================================\n'
            '1. print encrypted flag\n'
            '2. print encrypted fake flag\n'
            '3. print encrypted fake flag (TEST)\n'
            '4. encrypt\n'
            '====================================\n', 'utf-8'))

        choice = str(req.recv(2).strip(), 'utf-8')
        if choice not in menu:
            exit(1)

        menu[choice]()

        if choice == '4':
            continue

        cnt -= 1


class incoming(socketserver.BaseRequestHandler):
    def handle(self):
        signal.alarm(300)
        random.seed(time.time())

        req = self.request
        while True:
            go(req)


class ReusableTCPServer(socketserver.ForkingMixIn, socketserver.TCPServer):
    pass


socketserver.TCPServer.allow_reuse_address = True
server = ReusableTCPServer(("0.0.0.0", 23333), incoming)
server.serve_forever()
```

When I try to connect to the service, I was greeted by this message.

```
====================================
            fault box
====================================
1. print encrypted flag
2. print encrypted fake flag
3. print encrypted fake flag (TEST)
4. encrypt
====================================
```

So basically, we need to enter our chosen menu, and the service will return the encrypted message. For menu 1, the service will return the encrypted flag (the one that we need to decrypt), menu 2 will return the encrypted fake_flag, menu 3 will return the encrypted fake_flag also, but with different method (CRT), menu 4 will ask us for an input, and they will return the encrypted message. Let’s check the problem challenge code.

```
cnt = 2
while cnt > 0:
    req.sendall(bytes(
        '====================================\n'
        '            fault box\n'
```

```
            '====================================\n'
            '1. print encrypted flag\n'
            '2. print encrypted fake flag\n'
            '3. print encrypted fake flag (TEST)\n'
            '4. encrypt\n'
            '====================================\n', 'utf-8'))

        choice = str(req.recv(2).strip(), 'utf-8')
        if choice not in menu:
            exit(1)

        menu[choice]()

        if choice == '4':
            continue

        cnt -= 1
```

After examining for a while, this challenge only give us the e value, which is $0x10001$. Not only that, the challenge give us chance to encrypt as many message as we can using the fourth menu without changing the N value, but only give us chance to select two of the first three menu.

So, our first mission is we need to retrieve the N value first. Using the fourth menu and a simple math, we could retrieve the N value based on the definition of congruence itself. See below equation:

$$
\begin{aligned}
c &\equiv m^e \mod n && (1)\\
c &= n.k + m^e && (2)\\
c - m^e &= n.k && (3)
\end{aligned}
$$

Let say we have m_1 and m_2, then:

$$
\begin{aligned}
c_1 &\equiv m_1^e \mod n && (4)\\
c_1 &= n.k_1 + m_1^e && (5)\\
c_1 - m_1^e &= n.k_1 && (6)
\end{aligned}
$$

$$
\begin{aligned}
c_2 &\equiv m_2^e \mod n && (7)\\
c_2 &= n.k_2 + m_2^e && (8)\\
c_2 - m_2^e &= n.k_2 && (9)
\end{aligned}
$$

Using both equations, we can conclude that

$$n = GCD(c_1 - m_1^e, c_2 - m_2^e)$$

If the result is wrong, maybe what we got from the GCD is $n * GCD(k_1, k_2)$, and we just need to repeat the above equation and GCD it again.

After we got the n, we still can't factor the n because it's big. Examine the third option code

```
    # ===== FUNCTIONS FOR PERSONAL TESTS, DON'T USE THEM =====
    def TEST_CRT_encrypt(self, p, fun=0):
        ep = inverse(self.d, self.p-1)
        eq = inverse(self.d, self.q-1)
```

```
        eq = inverse(self.d, self.q-1)
        qinv = inverse(self.q, self.p)
        c1 = pow(p, ep, self.p)
        c2 = pow(p, eq, self.q) ^ fun
        h = (qinv * (c1 - c2)) % self.p
        c = c2 + h*self.q
        return c
```

So basically, the function want to encrypt the message using CRT, where ep and eq is equals to e actually. There is a bug on the encrypt where it xor the $pow(m, eq, q)$ with fun variable. We could do derivation from this faulty equation, which eventually will lead to the factor of N. See below equation:

$$\begin{cases} c_1 & \equiv m^e \mod p \\ c_2 & \equiv m^e \mod q \end{cases} \Rightarrow \begin{cases} c_1 - m^e & \equiv 0 \mod p \\ c_2 - m^e & \equiv 0 \mod q \end{cases} \Rightarrow \begin{cases} c_1 - m^e & = k_1.p \\ c_2 - m^e & = k_2.q \end{cases} \Rightarrow c - m^e = k_3.p.q$$

This is how to encrypt a message with RSA via CRT. Merge the c_1 and c_2 with CRT, you will get the c value. However, due to the xor (which caused Faulty RSA), what actually happens is like below:

$$\begin{cases} c_1 & \equiv m^e \mod p \\ c_2 & \not\equiv m^e \mod q \end{cases} \Rightarrow \begin{cases} c_1 - m^e & \equiv 0 \mod p \\ c_2 - m^e & \not\equiv 0 \mod q \end{cases} \Rightarrow \begin{cases} c_1 - m^e & = k_1.p \\ c_2 - m^e & \neq k_2.q \end{cases} \Rightarrow c - m^e = k_3.p$$

Based on above equations, the result of faulty encryption isn't divisible by q, so if we try to do $GCD(n, c_{faulty} - m^e)$, we will got p.

Our plan is clear now. What we need to do is:

- Get the fake_flag
- Get the faulty encryption of fake_flag
- GCD it with n
- Normally decrypt the real flag

However, we have a constraint:

- The service only limit us to choose two of the first three menu.

We definitely need to use it on option 1 to retrieve the c_{real}, and option 2 to retrieve the c_{faulty}. However, to use the above equations, we need to know the fake_flag (to generate $m^e_{FakeFlag}$). Luckily, notice that the space of the fake_flag is very small, so it is bruteforce-able. We can simply bruteforce it and validate it by doing $GCD(c_{faulty} - c_{BruteforceFakeFlag}, N)$. If the result is big and prime, we found the correct fake_flag value, because that result of the GCD is p. After that, we can simply decrypt the real flag. Below is my solver:

```
import socketserver
import random
import signal
import time
import gmpy2
from pwn import *
from Crypto.Util.number import *
```

```python
from itertools import product

context.log_level = 'error'

def s2n(s):
    return bytes_to_long(bytearray(s, 'latin-1'))

def n2s(n):
    return long_to_bytes(n).decode('latin-1')

def gen_fake(r, e, n):
    arr = []
    for i in range(1500):
        fake_flag = 'fake_flag{%s}' % (('%X' % i).rjust(32, '0'))
        enc_fake_flag = pow(s2n(fake_flag), e, n)
        arr.append(enc_fake_flag)
    return arr

e = 0x10001
r = remote("crypto.chal.csaw.io", 1001)

# Retrieving N value
r.recvuntil("encrypt\n====================================\n")
r.sendline('4')
r.recvuntil("data:")
r.sendline(n2s(2))
enc1 = int(r.recvuntil("\n"), 16)

r.recvuntil("encrypt\n====================================\n")
r.sendline('4')
r.recvuntil("data:")
r.sendline(n2s(3))
enc2 = int(r.recvuntil("\n"), 16)

r.recvuntil("encrypt\n====================================\n")
r.sendline('4')
r.recvuntil("data:")
r.sendline(n2s(5))
enc3 = int(r.recvuntil("\n"), 16)

r.recvuntil("encrypt\n====================================\n")
r.sendline('4')
r.recvuntil("data:")
r.sendline(n2s(7))
enc4 = int(r.recvuntil("\n"), 16)

# To prevent if the gcd result is not n but n*gcd(k1, k2)
n = gmpy2.gcd(gmpy2.gcd(2**e - enc1, 3**e - enc2), gmpy2.gcd(5**e - enc3, 7**e - enc4))
print "[+] n:", n

# get the encrypted real flag
r.recvuntil("encrypt\n====================================\n")
r.sendline('1')
c = int(r.recvuntil("\n"), 16)
print "[+] c:", c

# Retrieving the factor of n
r.recvuntil("encrypt\n====================================\n")
r.sendline('3')
enc_fake_flag_test = int(r.recvuntil("\n"), 16)

fake_flags = gen_fake(r, e, n)
for i, enc_fake_flag in enumerate(fake_flags):
    p = gmpy2.gcd(enc_fake_flag_test - enc_fake_flag, n)
```

```
    if (p > 1):
        print "[+] p:", p
        q = n/p
        phi = (p-1) * (q-1)
        d = inverse(e, phi)
        print "[+] Flag:", n2s(pow(c, d, n))
        break
```

> Flag: flag{ooo000_f4ul7y_4nd_pr3d1c74bl3_000ooo}

Writeup CSAW CTF crypto rsa 2019