Network Security Project

叶梓淳
520030910302

2023 年 6 月 7 日

1 环境配置

本次实验选择在 vscode 中搭建的 python 环境下进行，并使用了 Crypto 库，用于提供 AES 加解密服务，版本如下：

- python 版本：3.8.11
- Crypto 版本：1.4.1

运行命令请参考提交文件夹中的 README 文件。

2 实验过程

2.1 Textbook RSA

2.1.1 实现原理

RSA 加密算法是一种非对称算法，它依赖于大整数分解的困难性。它的公私钥生成过程如下：

- 生成大素数 p、q，计算 $N = p * q$
- 选择正整数 e，满足 $gcd(e, \phi(N)) = 1$ 且 $e < \phi(N)$，将（N，e）作为公钥对
- 计算 e 的乘法逆元 d，满足 $e * d \equiv 1(mod\ \phi(N))$，将（N，d）作为私钥对

生成完公私钥匙对后，加解密算法实现语法如下：

- $Enc(pk, m) \rightarrow c$: 输入公钥 $pk = (N, e)$ 和明文 $m \in Z_N^*$，输出密文 $c \equiv m^e (mod\ N)$
- $Dec(sk, c) \rightarrow m$: 输入私钥 $sk = (N, d)$ 和密文 $c \in Z_N^*$，输出明文 $m \equiv c^d (mod\ N)$

RSA 正确性验证过程：

$$\begin{aligned} Dec(sk, Enc(pk, m)) =& Dec(sk, m^e (mod\ N)) = (m^e\ mod\ N)^d\ (mod\ N) \\ &= m^{ed(mod\ \phi(N))}\ (mod\ N) = m (mod\ N) \end{aligned} \tag{1}$$

需要注意的是如果 m 大于 N，最终结果是 m 模 N 的余数。

2.1.2 代码分析

---素数生成

由于 N 的长度为 1024-bit，则将 p、q 的位数设置为 512-bit，循环直到两者相乘的结果为 1024-bit。使用 miller_rabin 素数检验算法生成 p、q。算法过程如下：

```
def miller_rabin(p, prime_size):
    if p % 2 == 0:
        return False
    if p in small_primes:
        return True
    for prime in small_primes:
        if p % prime == 0:
            return False

    target = p - 1
    r = 0
    while target % 2 == 0:
        target = target >> 1
        r += 1
    d = int(target)
    for i in range(prime_size):
        a = random.randrange(2, p - 1)
```

```
        v = pow(a, d, p)
        if v != 1:
            j = 0
            while v != p - 1:
                if j == r - 1:
                    return False
                else:
                    j += 1
                    v = (v ** 2) % p
    return True
```

其中，2-8 行利用素数集加快了判断速度，后面则是 miller_rabin 检验的具体过程，它利用了二次探测定理：

对于质数 p，若$x^2 \equiv 1(mod\ p)$，则小于 p 的解只有 1 和 p - 1

结合费马小定理，如果 $a^{\frac{p-1}{2}}$ 在模 p 的情况下解不是 1 或者 p - 1，则 p 不是素数。依此类推，由于 $p-1=d*2^r$(d 是奇数)，对 $a^d, a^{2d}, a^{4d}\cdots$ 一系列数进行检验，只有解全是 1 或者出现 p - 1 之后全是 1 才可能是素数。经实际检验，经过 p 的长度数次检验即可以几乎 1 的概率生成素数。

---生成公私钥

随机生成正整数 e，使用扩展欧几里得算法计算得到 e 与 $\phi(N)$ 的最大公因数，直到互素，同时保存 d 的值，从而生成公私钥对。扩展欧几里得算法如下：

```
def extended_euclidean(a, b):
    s0, s1 = 1, 0
    t0, t1 = 0, 1
    while b:
        q = a // b
        s1, s0 = s0 - q * s1, s1
        t1, t0 = t0 - q * t1, t1
        a, b = b, a % b
    return a, s0
```

返回的 a 代表 a、b 的最大公因数，s_0 代表 a 的乘数 (如果 a、b 互素则为乘法逆元)

---加解密过程

加密文本的输入为多行的字符串格式，对输入逐行处理，首先转换为字节序列，之后调用 b2a_hex 函数将字节序列转为十六进制序列，进而转换为 int 类型。此后进行加密过程，使用快速模指数运算得到结果，最后输出十六进制数。

```
def encrypt_plaintext(public_key, input_path, output_path):
    plaintext = []
    with open(input_path, 'r') as file:
        lines = file.readlines()
        for line in lines:
            plaintext.append(line.strip('\n'))
    file.close()

    ciphertext = []
    for i in range(len(plaintext)):
        currenttext = plaintext[i]
        currenttext = bytes(currenttext, encoding='utf-8')
        currenttext = int(binascii.b2a_hex(currenttext), 16)
        cipher = fast_mod(currenttext, public_key)
        ciphertext.append(hex(cipher))

    writer_file(ciphertext, output_path)
    print("--Encryption has completed\n")
```

解密过程与上述流程相反，使用 a2b_hex 函数将十六进制序列转换为字节序列，最后编码成字符串即可。

2.2 CCA2 Attack

2.2.1 实现原理

参考论文 “When Textbook RSA is Used to Protect the Privacy of Hundreds of Millions of Users”，提供如下的交互模型：

- Client:

  (a) 生成 128-bit 的 AES 密钥作为会话密钥；

(b) 使用 1024-bit 的 RSA 公钥来加密会话密钥

(c) 使用 AES 会话密钥对 WUP 交互报文进行加密

(d) 将经过 RSA 加密后的会话密钥和经过 AES 加密后的 WUP 交互报文生成 message 发送给 server

- Server:

(a) 使用 RSA 私钥解密接收到的 AES 会话密钥

(b) 选择解密结果中有效的 128-bit 低位作为 AES 会话密钥

(c) 使用 AES 会话密钥和原始长度信息解密接收到的 WUP 交互报文

(d) 若 WUP 交互报文中的 request 合法，返回一个 AES 解密成功的响应。

通过上述模式分析，定义 WUP 类和 message 类如下：

```
class WUP:
    def __init__(self, request, response):
        self.request = request
        self.response = response

class message:
    def __init__(self, en_WUP, en_AES_key):
        self.en_WUP = en_WUP
        self.en_AES_key = en_AES_key
```

WUP 类包括成员 request 和 response，在本次实验中只需用到一条 request 字段即可实施 CCA2 攻击。攻击原理如下：

令 C 表示使用 RSA 公钥 (N, e) 加密后 128-bit 的 AES 密钥 k，则有

$$C \equiv k^e (mod\ N) \tag{2}$$

令 $k_b = 2^b k$ 表示 k 向左进行了 b 比特的移位，C_b 表示 RSA 加密后的 AES 密钥，则

$$C_b \equiv k_b^e (mod\ N) \tag{3}$$

因此，可以仅通过 C 和公钥 e 计算出 C_b

$$\begin{aligned} C_b \equiv k_b^e (mod\ N) &\equiv (k^e\ mod\ N) * (2^{eb}\ mod\ N)(mod\ N) \\ &\equiv C(2^{eb}\ mod\ N)(mod\ N) \end{aligned} \tag{4}$$

首先考虑 k_{127} 和 C_{127}，k_{127} 的最高位是 k 的最低位，其他位均为 0。猜测 k_{127} 的最高位是 1，并发送一条由 k_{127} 加密的 WUP 报文，与 C_{127} 组合成一条 message 发送给 server。如果 server 利用此条 message 解密的 WUP 请求合法，则说明 k 的最高位确实是 1，否则是 0，依此类推直到恢复 k 所有的信息，该过程只需迭代 128 次即可破解密钥，时间成本完全可以接受。

2.2.2 代码分析

---加密部分

定义 client 类和 server 类以及必要的成员函数，其中 client 拥有 Task 1 中生成的 RSA 公钥，server 拥有私钥。由于 CCA2 攻击只需用到一条 WUP 请求，这里不再模拟 server 的响应，只需要 client 的成员函数 encrypt_WUP 即可，对于长度不是 16 字节倍数的 WUP 请求，选择在末尾填充'0'，代码如下：

```
def encrypt_WUP(self, ori_WUP):
    AES_encryptor = AES.new(a2b_hex(hex(self.AES_key)[2:]), AES.MODE_ECB)
    request = ori_WUP.request[2:]
    req_bit = len(request) * 4
    while req_bit % 128 != 0:
        request += "0"
        req_bit += 4

    request = bytes.fromhex(request)

    en_request = int(b2a_hex(AES_encryptor.encrypt(request)), 16)
    req_len = int(req_bit / 4)
    en_request = "0x{:0{}x}".format(en_request, req_len)
    return WUP(en_request, "")
```

历史消息生成函数代码如下：

```
def generate_history_message(en_WUP, client_ori):
    en_AES_key = client_ori.encrypt_AES_key()
```

```
    Message = message(en_WUP, en_AES_key)
    return Message
```

再将生成的 message 写入文档中即可。

---攻击部分

为了实施 CCA2 攻击，需要利用 server 提供的解密服务，成员函数如下：

```
def decrypt_WUP(self, en_WUP, ori_len):
    AES_key_string = ""
    for i in hex(self.AES_key)[2:]:
        AES_key_string += i
    while len(AES_key_string) < 32:
        AES_key_string = "0" + AES_key_string
    AES_decryptor = AES.new(a2b_hex(AES_key_string), AES.MODE_ECB)
    en_request = en_WUP.request[2:]

    de_request = AES_decryptor.decrypt(a2b_hex(en_request))
    de_request = int(b2a_hex(de_request)[0 : ori_len], 16)
    de_request = "0x{:0{}x}".format(de_request, ori_len)

    return WUP(de_request, "")
```

其过程与加密函数大致相反。要注意的是此时 AES 密钥转化为十六进制后必须扩充到 128 比特，以便作为参数传入 AES.new 函数。最后需要截取原始长度的部分以去掉填充。实现了解密函数后，攻击函数的主体部分如下：

```
for i in range(128, 0 , -1):
    ki = int(AES_key >> 1) + (1 << 127)
    req_len = len(attack_WUP.request) - 2
    attacker = client()
    attacker.reset_AES_key(ki)
    en_WUP = attacker.encrypt_WUP(attack_WUP)
    fac = fast_mod(2, key[0], (i - 1) * key[1])
    Ci = fast_mod(fac * en_AES_key, key[0], 1)
    Ci = bin(server_ori.decrypt_AES_key(Ci))[-128:]
    Ci = int(Ci, 2)

    target = server(Ci)
```

```
    de_WUP = target.decrypt_WUP(en_WUP, req_len)

    if de_WUP.request == attack_WUP.request:
        AES_key = ki
    else:
        AES_key = int(AES_key >> 1)
```

每一轮将已经获得的位数右移一位，再把最高位置 1，作为下一轮的猜测密钥。获取 AES 密钥后，再次对加密后的 request 调用解密函数即可输出结果。

2.3 OAEP

2.3.1 实现原理

OAEP（Optimal Asymmetric Encryption Padding）是一种对消息进行随机填充的策略，借助哈希函数 G、H，输入明文 m 和固定比特长度的 r，填充过程如下：

$$OAEP(m,r) = X \mid\mid Y = (m||0^{k_1}) \oplus G(r) \mid\mid r \oplus H((m||0^{k_1}) \oplus G(r)) \quad (5)$$

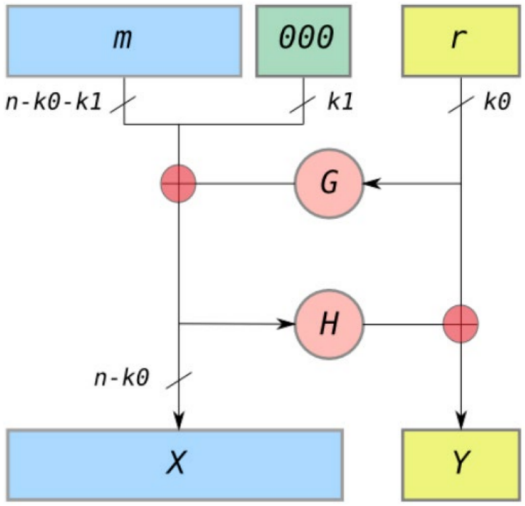

图 1: OAEP

相比于原始的 RSA 加密算法，OAEP 通过引入随机性和哈希函数，提供了语义安全。语义安全意味着攻击者无法通过分析密文获得关于明文的任何信息。即使攻击者拥有多个明文的加密结果，他们也无法确定这些明文的内容。同时，OAEP 的计算性能和空间利用率在实际应用中都非常有效。

2.3.2 代码分析

本次实验选用 $n = 1024$, $k_0 = 512$, $G = H = sha512()$。加密过程的主体部分如下：

```
for i in range(len(plaintext)):
    currenttext = plaintext[i]
    currenttext = binascii.b2a_hex(bytes(currenttext, encoding='utf-8'))
    currenttext = bin(int(currenttext, 16))[2:]

    if len(currenttext) > 512:
        print("The message of this line is longer than 512 bits")
        return
    k1 = 512 - len(currenttext)
    currenttext = currenttext + ('0' * k1)

    hash_G.update(r.encode('utf-8'))
    x = "{:0512b}".format(int(currenttext, 2) ^ int(hash_G.hexdigest(),
        16))
    hash_H.update(x.encode('utf-8'))
    y = "{:0512b}".format(int(r, 2) ^ int(hash_H.hexdigest(), 16))
    padded = int(x + y, 2)
    paddedtext.append(hex(padded))

    cipher_x = hex(fast_mod(int(x, 2), public_key))[2:]
    cipher_y = hex(fast_mod(int(y, 2), public_key))[2:]
    while len(cipher_x) != 256:
        cipher_x = '0' + cipher_x
    while len(cipher_y) != 256:
        cipher_y = '0' + cipher_y
    cipher = '0x' + cipher_x + cipher_y
    ciphertext.append(cipher)
```

为方便起见，这里限制了明文的长度不大于 512-bit。哈希函数由 hashlib 库

提供。为了防止进制转换过程中高位 0 被省略，需要使用类似 {:0512b} 的语法指定生成的字符串形式。选择分别加密 x 和 y，再将它们拼接到一起，这样做是避免先拼接后再加密导致明文值大于 N，解密阶段出现问题。因此，Encrypted_Message.txt 文件中被加密的十六进制字符串长度是 512。

解密过程与上述过程相反。取密文前半部分作为 x 加密后的密文，后半部分作为 y 加密后的密文，再按 OAEP 的逆序进行解密，最终去掉末尾填充的 0 即可。

3 实验结果

3.1 Textbook RSA

运行各个命令，终端输出如下：

```
PS C:\Users\86159\Downloads\Project_520030910302\Task 1> python Textbook_RSA.py --generate_keys --n_bits 1024
--Generation has completed

PS C:\Users\86159\Downloads\Project_520030910302\Task 1> python Textbook_RSA.py --encrypt_plaintext ./Encryption/Raw_Message.txt
--Encryption has completed

PS C:\Users\86159\Downloads\Project_520030910302\Task 1> python Textbook_RSA.py --decrypt_ciphertext ./Encryption/Encrypted_Message.txt
--Decryption has completed
```

图 2: Textbook_RSA 运行结果

运行密钥生成函数完毕后，生成的参数 (N, p, q) 会存放到 parameters 文件下；公私钥对会存放到 key 文件下，且第一行为 N，第二行为 e/d；加解密函数结果存放在 Encryption 文件下，最终可验证 Raw_Message.txt 与 Decrypted_Message.txt 完全相同。

3.2 CCA2 Attack

运行命令，得到输出结果：

```
PS C:\Users\86159\Downloads\Project_520030910302\Task 2> python CCA_Attack.py
--AES_Key has been generated

--Encrypt WUP_Request
--Finished

--Generate history message
--Finished

--Start attack
--The AES key is 0xf886610fe02893a661db8d67659e54d9
--Finished, please refer to Logs.txt for more information

--Decrypt AES_Encrypted_WUP
The decrypted request is 0x123456789abcdefaadd23
--Finished
PS C:\Users\86159\Downloads\Project_520030910302\Task 2>
```

图 3: CCA_Attack 运行结果

其中历史消息生成函数产生的文档 History_Message.txt 第一行为加密后的 WUP 请求，第二行为加密后的 AES 会话密钥。攻击过程生成的文档 Logs.txt 是每轮测试时输出的结果，截取部分如下图所示：

```
This turn I will try AES key =
0b10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000

The 128-th bit of AES key is 1
This turn I will try AES key =
0b11000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000

The 127-th bit of AES key is 0
This turn I will try AES key =
0b10100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000
```

图 4: Logs.txt 部分内容

最终用攻击获取的 AES 密钥解密加密后的 WUP 请求，得到结果与 WUP_Request.txt 文件内容相同。

3.3 OAEP

运行命令, 终端输出如下：

图 5: OAEP_RSA 运行结果

运行加密函数完毕后,生成 Random_Number.txt、Message_After_Padding.txt 和 Encrypted_Message.txt 文件，运行解密函数后，最终可验证 Raw_Message.txt 与 Decrypted_Message.txt 完全相同。

4 总结

在本次实验的过程中，我用代码实现的方式体验了 RSA 加密技术的使用，了解了 CCA2 攻击的具体原理和 OAEP-RSA 的具体实现，这使得我对 RSA 的底层原理以及实际应用有了更深的认识。实验过程中，对我来说最大的难点是在于各种进制间的转换，字符串与比特流的转换，以及填充与反填充，需要注意很多细节才能保证代码的正常运行。特别是在实现 OAEP 的过程中，忽略了 m 可能大于 N 时的情况，导致有时代码能跑，有时报错。

感谢刘振、朱浩瑾老师、张乐助教在课堂及课后答疑上的教学和指导。