Cyber Security Challenge 2017

Week 5

Monash University Tutorial on RSA Attack

Prepared by: Joseph Liu

Setup Requirements:

1. Install OpenSSL, Python, gcc (or other C compiler), if you don’t have these tools in your computer.
2. After the installation, don’t forget to set these tools to the path, so that you can run it at any directory (from the console).

Step-by-Step Attack:

1. Download .pem (public key) files, and the ciphertext files (encrypted using these public keys). Suppose there are some public keys (n) that share a common factor. Our goal is to decrypt the ciphertext using the information of public key only!

2. Extract the n value from the .pem file (you can use Python to do that):

```
>>> from Crypto.PublicKey import RSA
>>> pem1 = open("public1.pem").read()
>>> k1 = RSA.importKey(pem1)
>>> k1.n
15852443160399830162706305179471887570348682849317352
20447180482118233105776697627128257981224148674034647
50160902730286113975680954506122323962740369060552422
61894361607421267912379277283491056459774023869340580
81011132575569223986796811601920662271509859125225566
41505035902589310014574430872209669989818457
```

(suppose `public1.pem` is the public key file)

3. Repeat Step 2 to get another public key and extract its n value.

4. Find GCD(n1, n2). If it is not equal to 1, it is a common factor for n1 and n2:
    - Compile gcd.c using the following gcc command:
      `gcc -I/openssl -o gcd_exe_file gcc.c -lcrypto`

      (assume `/openssl` is the directory for the **openSSL library** – that is, the directory storing those .h files)
    - Usage: gcd n1 n2

      You can copy and paste the n1 and n2 values from your screen that you have obtained in Step 2.
    - Output: gcd(n1, n2)

      If the output is not equal to 1, that means n1 and n2 share a common non-trivial factor!

5. Suppose you have found a non-trivial factor for n. Now you need to find the other non-trivial factor:
    - Compile another_fac.c using the following gcc command:
      `gcc -I/openssl -o anofac_exe_file another_fac.c -lcrypto`
      (assume `/openssl` is the directory for the **openSSL library** – that is, the directory storing those .h files)
    - Usage: another_fac n p

      You can copy and paste the n and p values from your screen that you have obtained in Step 2 and Step 4, respectively.
    - Output: Another factor for n if p is one factor.

6. Once you've got p, q (2 factors of n), compute the corresponding d (and thus the private key):
    - Compile private-from-pq.c using the following gcc command:
      `gcc -I/openssl -o pri_exe_file private-from-pq.c -lcrypto`
      (assume `/openssl` is the directory for the **openSSL library** – that is, the directory storing those .h files)

- Usage: private-from-pq p q
  You can copy and paste the p and q values from your screen that you have obtained in Step 4 and Step 5, respectively.

- Output: A RSA private key file (in PEM format) containing p, q, d (on screen)

  You can copy and paste from the screen and store it as a file, e.g. private_key.pem

```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDBeNS5jaWzj5LpSVdw/yYoNvFerv0GOiCPIHd1z/jKdgR9URQG
B9upbNJNbYMum2bFCU/ZR0mVrW+gcmxp4HhggmGHWs4HCwUhDQDElyZnW434hJpY
2O1UuBKcQvdXmwTFM2WRSGYK955MaPqeZ0u07vKfCxeHjpEzsBkvhUFybQIDAQAB
AoGAbbrMW9kcq/S4TBvOvkXYLGLNDviMhWWeZc7yMh2ca6f37+N2Sd1XFoaj2EpO
Zra42PPpF8C42W2erA95sasgWg5d38K3UNdaxUV1mKh9X1JCC8fpBCCuUeDPGzHn
Ypu/pKFLfPBYXKYNn1uNZJj+Yzwn7wzYKGrfb7YANXAatIECQQD4dFJsh8V1aStA
164zZrbeO8ofqdwcoWjSnfIn1+cEfzQOS70sK+xZpdL8GGW+jm5nWa0ZvuLq9/3X
jaDvdBnxAkEAx1kIdmMad6x8PEsKeVxgT3hLjilAfNiXemS1B6tN1DvCpOoD9tV9
WwZAv61lF9oTlnA4HmT9fdfr/ZlnfaeEPQJBANGP283rHxaQhIkm4qo8YtZD/BKF
BUGo629rBcuRkiv61v+P5roROkPLWJCGS5tVK85El3r1xRSHpDZiIXKXrSECQDKh
TUHsN7uvZjpWAMoECT4F2oK3vXY3+HkQeM2y11hPuUbzXKBjZpwowWctXeCp9ZGe
3NHzamJ85aYaQSur0S0CQQDl7Y1mEPlpoctq/qJPdlOuebUxnc/ugct+U/Q/WLmj
pkalxxf0nWbZvxNKBWOeInbYn7gRXPDYWMPrAgByr6jI
-----END RSA PRIVATE KEY-----
```

  Or you can modify the `private-from-pq.c` file to make it output the PEM format private1.pem directly.

7. Decrypt the ciphertext using the private key (you can use openSSL), using the following command:

   ```
   openssl rsautl -decrypt -inkey private_key.pem
   -in ciphertext.dat -out plaintext.txt
   ```

Note: This is just a guideline. You can write your own code, or to combine these steps into one single program.

Exercise:

1. Try the small challenge (containing only 3 public keys and 3 ciphertexts).
2. Try the big challenge (containing 100 public keys and 100 ciphertexts).

Interesting Fact:

1. How often does it occur to have two RSA common-factor moduli in the Internet?

   - Reference page: http://www.loyalty.org/~schoen/rsa/
   - Under the Section: Common factors in practice
   - They mention that around 0.28% 512-bit integers are prime. this suggests that there are somewhere between 2^{503} and 2^{504} 512-bit primes.
   - How about in the real world?
   - http://eprint.iacr.org/2012/064.pdf

among the 4.7 million distinct 1024-bit RSA moduli that we had originally collected, 12720 have a single large prime factor in common. That this happens may be crypto-folklore, but it was new to us (but see [14]), and it does not seem to be a disappearing trend: in our current collection[3] of 11.4 million RSA moduli 26965 are vulnerable, including ten 2048-bit ones. When exploited, it could affect the expectation of security that the public key infrastructure is intended to achieve.

2. How to find the GCD(n1, n2)?

   - Using Euclidean algorithm: https://en.wikipedia.org/wiki/Euclidean_algorithm
   - E.g. GCD(1071, 462)

     1071 = 462 X 2 + 147 [147 is the reminder]

     462 = 147 X 3 + 21 [21 is the reminder]

     147 = 21 X 7 + 0 [0 is the reminder]

     The last non-zero reminder is the GCD of (1071, 462)