en_US

Samsung Research

SSTF 2021 | Hacker's Playground

Tutorial Guide

RSA 101

Crypto

Do you remember?

	Symmetric Encryption	Asymmetric Encryption
Key	One shared key for encryption and decryption	Mathematically coupled public key and private key
Typical Key Size	128~256 bits	1024~3072 bits (for RSA)
Performance	High	Low, because it's a complex mathematical computation
Main Purpose	Data Encryption	Digital Signature/Certificate
Representative Algorithms	DES, AES, RC4	**RSA**, DSA, ECC

RSA(Rivest-Shamir-Adleman)

✔ The most widely used public-key cryptosystem.

- To establish secure channels such as HTTPS(SSL/TLS), email, VPNs, or so on.
- To ensure the integrity of the firmware in the devices.
- For the echo systems including payment, banking, contents protection, and so many things.

✔ Basically, RSA is a modular exponentiation operation.

$$\mathrm{RSA\_enc}((K_e, n), p) = p^{K_e} \bmod n$$

$$\mathrm{RSA\_dec}((K_d, n), c) = c^{K_d} \bmod n$$

- Modular exponentiation = exponentiation + modulo operation.

Asymmetric encryption algorithm is sometimes referred to as 'public-key cryptosystem' or 'public-key infrastructure(PKI)'.

Mathematics for RSA

Exponentiation

$$b^n = \underbrace{b \times b \times \cdots \times b}_{n \text{ times}}$$

- Repeated multiplication of the base.
- Usually expressed as a pow function; $\mathrm{pow}(b, n)$.
- It has some identities and properties.
  - $b^0 = 1$
  - $b^{m+n} = b^m \cdot b^n$
  - $(b^m)^n = b^{m \cdot n}$
  - $(b \cdot c)^n = b^n \cdot c^n$
  - $b^{-n} = \dfrac{1}{b^n}$
  - $b^{m^n} = b^{(m^n)}$

We only consider integer exponents and non-zero bases here.

Mathematics for RSA

Modulo operation

- Used to get the remainder, and usually expressed as a '%' or 'mod' operator.
- modulo n defines a number system consisting of the numbers 0 to $n-1$.
  In that system, we can say $12 \equiv 42$ when $n = 10$, and write it like this:
  $$12 \equiv 42 \pmod{10}$$
- It also has some identities and properties.
  - $(a \bmod n) \bmod n = a \bmod n$
  - $n^x \bmod n = 0$ for all positive integer x
  - $((-a \bmod n) + (a \bmod n)) = 0$
  - $(a+b) \bmod n = ((a \bmod n)+(b \bmod n)) \bmod n$
  - $ab \bmod n = ((a \bmod n)(b \bmod n)) \bmod n$
  - $((b \bmod n)(b^{-1} \bmod n)) \bmod n = 1$

Refer to modular arithmetic and Finite field.

$b^{-1} \bmod n$ denotes the modular multiplicative inverse, which is defined iff b and n are coprime.

Mathematics for RSA

Euler's totient function, $\varphi(n)$

- Number of integers k in the range $1 \leq k \leq n$, for which $\gcd(n, k) = 1$.
- For any prime numbers $p \neq q$, $\varphi(p) = p - 1$ and $\varphi(pq) = (p-1)(q-1)$.

Euler's theorem

- When n and a are coprime positive integers,
  $a^{\varphi(n)} \equiv 1 \pmod{n}$.

And some more things not covered here.

- gcd(greatest common divisor)
- prime number

RSA: Key generation

- **Step 1. Choose two distinct primes, p and q.**
  - Then $n = pq$ and $\varphi(n) = \varphi(pq) = (p-1)(q-1)$.
  - n is a huge number, so finding p or q from n by factorization is really difficult.

Refer to NP problem.

- **Step 2. Choose e where $1 < e < \varphi(n)$ and $\gcd(\varphi(n), e) = 1$.**
  - 65537 is commonly used for e.
  - And calculate $d \equiv e^{-1} \ (mod\ \varphi(n))$, by using the Extended Euclidean algorithm.
- **Step 3. Now you have a RSA Key pair.**
  - Public exponent e, for encryption, and modulus n will be opened to the public.
  - Private exponent d, for decryption, and parameters p and q should be kept secret.
    - If an attacker can find one of p, q, or d, he can decrypt any ciphertext.

In recent years, it is recommended to use Carmichael's totient function instead of Euler's totient function.

RSA: Encryption and decryption

- Encrypting message m with public key (n, e)
  - Ciphertext $c \equiv m^e \pmod{n}$
- Decrypting ciphertext c with private key (n, d)
  - $m \equiv c^d \pmod{n}$
  - $$\begin{aligned} c^d \equiv (m^e)^d \equiv m^{ed} & \pmod{n} & \\ \equiv m^{\varphi(n)\cdot k+1} & \pmod{n} & \because ed \equiv 1 \pmod{\varphi(n)} \\ \equiv m^{\varphi(n)\cdot k} \cdot m & \pmod{n} & \\ \equiv (m^{\varphi(n)})^k \cdot m & \pmod{n} & \\ \equiv 1^k \cdot m & \pmod{n} & \because \text{Euler's theorem} \\ \equiv m & \pmod{n} & \end{aligned}$$
- Therefore, anyone can encrypt a message, but only the owner of the private key can decrypt it.

RSA: Signing and verification

- **Generating digital signature s on document m with private key (n, d)**
  - Signature $s \equiv m^d \ (mod\ n)$
  - In most cases, message digest, $h(m)$, is used instead of m.

Refer to
cryptographic hash function.

- **Verifying s on m with public key (n, e)**
  - Signature s is valid iff $m \equiv s^e \pmod{n}$

- **Therefore, only the owner of the private key can generate a signature and anyone can verify it.**

Let's solve Crypto quiz!

Quiz #1
& solution

Quiz #1

```
n = 71429248760466541506031966887517802919937362412357335088258769184139035122083
e = 65537
p = 252306102913729311695741849559437257013
ct = 61617714183432990975533017724678764406865679264090036586430429254817365860213
```

Download the source code HERE.

- Simple python code with RSA parameters
- A secret parameter, p is given.
- Can you decrypt `ct`?

Solution for Quiz #1

- According to the RSA algorithm,
  we can recover all secret parameters.

```python
n = 71429248760466541506031966887517802919937362412357335088258769184139035122083
e = 65537
p = 252306102913729311695741849559437257013
ct = 61617714183432990975533017724678764406865679264090036586430429254817365860213

q = n // p                          # n = pq, so we can get q from n and p.

phi_n = (p - 1) * (q - 1)           # calculates φ(n) = (p − 1)(q − 1)

d = pow(e, -1, phi_n)               # calculates d ≡ e^-1 (mod φ(n))

m = pow(ct, d, n)                   # calculates m ≡ ct^d (mod n)

print(hex(m))                       # prints m in the hex representation

print(bytes.fromhex(hex(m)[2:]))    # prints m in the bytes representation
```

- Decryption success!

```
$ python3 ex.py
0x5468655f3173745f61747461636b5f306e5f525341
b'The_1st_attack_0n_RSA'
```

Quiz #2

& solution

Quiz #2

```python
from sympy import randprime, nextprime
from secret import pt

p = randprime(pow(2, 511), pow(2, 512))
q = nextprime(p)
n = p * q
e = 65537

ct = pow(pt, e, n)

print("n =", hex(n))
print("ct =", hex(ct))

'''
$ python3 challenge.py
n = 0xa28c55dd2df4f6845a1faf7755c080a... (omitted)
ct = 0x19cea145a7495409a0ab504261b80e... (omitted)
'''
```

Download the source code HERE.

- RSA encryption program.
- Can you get the plaintext, `pt`?
- Try it before you see the solution.
- HINT: You may need a little bit brute-forcing.

Solution for Quiz #2

```python
from sympy import randprime, nextprime
from secret import pt

p = randprime(pow(2, 511), pow(2, 512))
q = nextprime(p)
n = p * q
e = 65537

ct = pow(pt, e, n)

print("n =", hex(n))
print("ct =", hex(ct))

'''
$ python3 challenge.py
n = 0xa28c55dd2df4f6845a1faf7755c080a... (omitted)
ct = 0x19cea145a7495409a0ab504261b80e... (omitted)
'''
```

- **In the RSA key generation,**
  - p and q should be independently generated.
- **But here, q is the smallest prime greater than p.**
  - So $p \neq q$, but they'll be very close.
  - We can attack this point.

Solution for Quiz #2

- Let's generate sample RSA parameters in the same way.

```python
from sympy import randprime, nextprime

p = randprime(pow(2, 511), pow(2, 512))
q = nextprime(p)

print(hex(p))
print(hex(q))
```

```
$ python3 test.py
0xaf90d12bbc75c45b9d4653f26942931d2742bb3205517f6c2c253e012f0c5ca50644
75cb0dd91bedccca046dac43f28b421f07a3eca233a5cff0e277c686e627
0xaf90d12bbc75c45b9d4653f26942931d2742bb3205517f6c2c253e012f0c5ca50644
75cb0dd91bedccca046dac43f28b421f07a3eca233a5cff0e277c686e661
$ python3 test.py
0xcfa08b2bb696a1172f54d6cb9d72d408f86ed1830d14c568a0b6929260f49a8ca077
02cf5cb0cca6753c12b50ecf806cc3b9f614c0f9c698698df477a4320727
0xcfa08b2bb696a1172f54d6cb9d72d408f86ed1830d14c568a0b6929260f49a8ca077
02cf5cb0cca6753c12b50ecf806cc3b9f614c0f9c698698df477a4320795
$ python3 test.py
0xc84708c931b2024b502918dd9443477efe52d85b708892ea7666aac072ff17330235
4f40f72f7db45392cc533101ee248c03a6d1c51f9e8b13fdaf9a0b06b2a5
0xc84708c931b2024b502918dd9443477efe52d85b708892ea7666aac072ff17330235
4f40f72f7db45392cc533101ee248c03a6d1c51f9e8b13fdaf9a0b06b4e1
```

- We can see that p and q are the **same** except for the **last few bits**.

Solution for Quiz #2

- So we can say $p \approx q$.
- $n = pq \approx p^2$,
  therefore $\sqrt{n} \approx p$ and $p - \sqrt{n} \approx 0$.
- We can calculate $\sqrt{n}$,
  so we can easily find p near $\sqrt{n}$
  by brute-force attack.
- Decryption success!

```
$ python3 ex.py
0x5468335f326e645f61747434636b5f4f6e5f525341
b'Th3_2nd_att4ck_On_RSA'
```

```
n = 0xa28c55dd2df4f6845a1faf7755c0... (omitted)
ct = 0x19cea145a7495409a0ab504261b... (omitted)

# gmpy2 is an arithmetic library for python
from gmpy2 import isqrt
p = isqrt(n)      # integer square root of n

e = 65537

# simple brute-force
while n % p != 0:
    p += 1

# same as Quiz #1
q = n // p
phi_n = (p - 1) * (q - 1)
d = pow(e, -1, phi_n)
m = pow(ct, d, n)

print(hex(m))
print(bytes.fromhex(hex(m)[2:]))
```

There's a mathematical method that simply finds $\varphi(n)$ when $p \approx q$, but it is not covered here.

Let's practice

Solve the tutorial challenge

Challenge Definition

```
$ nc rsa101.sstf.site 1104
[RSA parameters]
n = 0x9aabdceb4c9e3a3820863f7a949584c05db75aa4946fd8a94375b93a22ead9fd
ccb1741dcc39c668081ca3ba488f3708d1c41ee3f673a1d720864a115d730347c19df2
02e5e0a79ae7643e73acbac2099e5576aa68be3c1932f3f5f457c547a249d5f5adf43c
561fa0d54318502cfe3d85c2c5fbb94c48292c219c818aa0e29f
e = 0x10001

Welcome to command signer/executor.
Menu : 1. Verify and run the signed command
       2. Generate a signed command
       3. Base64 encoder
       4. Exit
 >
```

```
 > 2
Base64 encoded command to sign:
```

- A command signer/executor
  - based on RSA
  - RSA public parameters are given.
- It seems that
  - we can generate a signed command and execute it.
- The signer takes a base64 encoded command.

Challenge Definition

```
 > 3
String to encode: whoami
Base64 encoded string: d2hvYW1p

Welcome to command signer/executor.
Menu : 1. Verify and run the signed command
       2. Generate a signed command
       3. Base64 encoder
       4. Exit
 > 2
Base64 encoded command to sign: d2hvYW1p
Signed command: Pax0ifjBvePTSedak2WtjI50/Jkim31Q7Tkq9bUVuk8vLTq2p4UxHs
7/wI8WUqsRZReHHU3Rp7TcVExMNQLqywl9h+zXUYeTNwogXCZD31fsqJLPZNM4eUUsxjWm
bAnPCrUleh8CKok0xIBHjRlFVjP74c3s8daXdKemuOTELsE=

Welcome to command signer/executor.
Menu : 1. Verify and run the signed command
       2. Generate a signed command
       3. Base64 encoder
       4. Exit
 > 1
Signed command: Pax0ifjBvePTSedak2WtjI50/Jkim31Q7Tkq9bUVuk8vLTq2p4UxHs
7/wI8WUqsRZReHHU3Rp7TcVExMNQLqywl9h+zXUYeTNwogXCZD31fsqJLPZNM4eUUsxjWm
bAnPCrUleh8CKok0xIBHjRlFVjP74c3s8daXdKemuOTELsE=
Your command is not in the white list.
Possible commands:  ['ls -l', 'pwd', 'id', 'cat flag']
```

- Testing a command, `whoami`.
  - Signature generation success

    $$s \equiv m^d \pmod{n}$$

    ```python
    def sign(msg):
        m = bytes_to_long(msg)
        s = pow(m, d, n)
        return long_to_bytes(s)
    ```

  - but execution failed.
- Only commands in the white list can be executed.
  - `cat flag` is what we need.

Challenge Definition

```python
elif sel == "2":
    cmd = input("Base64 encoded command to sign: ")
    cmd = b64decode(cmd)
    if cmd == b"cat flag":
        print("It's forbidden.")
    else:
        print("Signed command:", b64encode(sign(cmd)).decode())
```

```
 > 3
String to encode: cat flag
Base64 encoded string: Y2F0IGZsYWc=

Welcome to command signer/executor.
Menu : 1. Verify and run the signed command
       2. Generate a signed command
       3. Base64 encoder
       4. Exit
 > 2
Base64 encoded command to sign: Y2F0IGZsYWc=
It's forbidden.
```

- ✔ We can sign any command
  - except for `cat flag`.
- ✔ So we should
  - make a signature of `cat flag` from other commands and signatures,
  - without RSA private key.

Decomposition of the command

- Let $c = \text{integer}(\texttt{"cat flag"})$.
- We can generate $s \equiv m^d \pmod{n}$ for all $m \neq c$.
- Can you find m_1, m_2 such that $m_1 \neq c \neq m_2$ and $m_1 \cdot m_2 = c$?
  - The simplest way is using factorization.

```
>>> c = int(b"cat flag".hex(), 16)
>>> for i in range(2, c // 2):
...     if c % i == 0:
...         print(i)
...         break
...
103
>>> c // 103
69525558883514113
```

  - We got $m_1 = 103,\ m_2 = 69525558883514113$.

Decomposition of the command

- ✔ Now we can get $s_1 \equiv m_1^d, s_2 \equiv m_2^d \pmod{n}$, respectively.

```
>>> from Crypto.Util.number import long_to_bytes
>>> from base64 import b64encode
>>> b64encode(long_to_bytes(103))
b'Zw=='
>>> b64encode(long_to_bytes(695255588835141l3))
b'9wEgoA/3AQ=='
```

```
>>> from base64 import b64decode
>>> from Crypto.Util.number import bytes_to_long
>>> s1 = bytes_to_long(b64decode("g9617XEI4XLsIf95
f5dfJnPX1l7Vd+V0MDH/Re3ORUD56UOJSQZ78MXAYAmclILVtQ
h5mtUvrYf9P+zA0pr45tWBYdnhudF23P62RyHLUcJIfVubhb9b
CKjhEOfG6gvtV6IkF2johU2bq2kAF1L2h1SHnPv08ozZRAkhx4
MIDS4="))
>>> s2 = bytes_to_long(b64decode("ODzQCjnEV79m68CX
Aec1SNgUWyPGkd5Epll3VQunl6OgxbSg6nsB3LJtKjsCWeyDAj
1zBgOBN+x0s3Sa6k8O9aTEiRWtzl/I1Bd1Q1kySqKLkweTkzPq
KtpsCbozQbDTwNAxVG8sG892Fviv+EQoMUaT4w3J937KVjjtgr
eX4MA="))
```

```
 > 2
Base64 encoded command to sign: Zw==
Signed command: g9617XEI4XLsIf95f5dfJnPX1l7Vd+V0MD
H/Re3ORUD56UOJSQZ78MXAYAmclILVtQh5mtUvrYf9P+zA0pr4
5tWBYdnhudF23P62RyHLUcJIfVubhb9bCKjhEOfG6gvtV6IkF2
johU2bq2kAF1L2h1SHnPv08ozZRAkhx4MIDS4=

Welcome to command signer/executor.
Menu : 1. Verify and run the signed command
       2. Generate a signed command
       3. Base64 encoder
       4. Exit
 > 2
Base64 encoded command to sign: 9wEgoA/3AQ==
Signed command: ODzQCjnEV79m68CXAec1SNgUWyPGkd5Epl
l3VQunl6OgxbSg6nsB3LJtKjsCWeyDAj1zBgOBN+x0s3Sa6k8O
9aTEiRWtzl/I1Bd1Q1kySqKLkweTkzPqKtpsCbozQbDTwNAxVG
8sG892Fviv+EQoMUaT4w3J937KVjjtgreX4MA=
```

Multiplication Properties of Exponents

- What happens to $s_1 \cdot s_2 \pmod{n}$?

$$s_1 \cdot s_2 \equiv m_1^d \cdot m_2^d \equiv (m_1 \cdot m_2)^d \equiv c^d \pmod{n}$$

- We got $c^d \bmod n$, without private key!
  - Let's make sure it works properly.

```
>>> n = 0x9aabdceb4c9e3a3820863f7a949584c05db75aa4
946fd8a94375b93a22ead9fdccb1741dcc39c668081ca3ba48
8f3708d1c41ee3f673a1d720864a115d730347c19df202e5e0
a79ae7643e73acbac2099e5576aa68be3c1932f3f5f457c547
a249d5f5adf43c561fa0d54318502cfe3d85c2c5fbb94c4829
2c219c818aa0e29f
>>>
>>> b64encode(long_to_bytes((s1*s2)%n))
b'iwY6mKS++NM7Ux9hoa9UnxB1gQsWqk6HI6mi1/lm0m6+khdZ
ED+gWSbzgQplWSnEVOcIwjQW3Jfqa9Wmesnysuz0Ha+ybXmDw/
vyX9/5hj1BiziU+sfZYvLDHPc4U3O35tPLZ+JRrZftBXrN1voE
gkfVq7AvFTKTxbeh+RQfSSg='
```

```
 > 1
Signed command: iwY6mKS++NM7Ux9hoa9UnxB1gQsWqk6HI6
mi1/lm0m6+khdZED+gWSbzgQplWSnEVOcIwjQW3Jfqa9Wmesny
suz0Ha+ybXmDw/vyX9/5hj1BiziU+sfZYvLDHPc4U3O35tPLZ+
JRrZftBXrN1voEgkfVq7AvFTKTxbeh+RQfSSg=
SCTF{[redacted]}
```

Try it yourself!

What if you can't factorize c?

- You can still use the same method.
  - Choose an integer $a > 1$.
  - Get $s_a \equiv (a \cdot c)^d \pmod{n}$.
  - Calculate $b \equiv a^{-1} \pmod{n}$ by using extended Euclidean algorithm.
  - Get $s_b \equiv b^d \pmod{n}$.
  - Now, $s_a \cdot s_b \equiv (a \cdot c)^d \cdot b^d \equiv (a \cdot c \cdot b)^d \equiv (a \cdot c \cdot a^{-1})^d \equiv c^d \pmod{n}$.

! You should adopt the appropriate padding algorithm to prevent such attacks.