# Mason Competitive Cyber

Intro to (CTF) Crypto



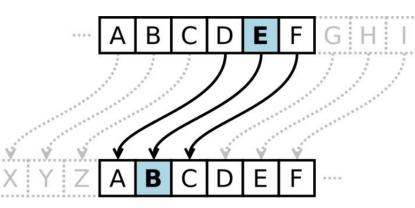
#### What is it?

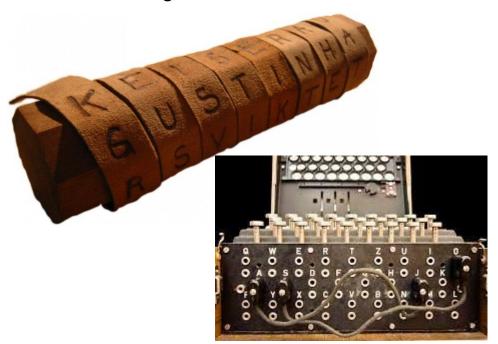


- Cryptography is the study of techniques used to secure data/communications
- History goes back a long way

• Greeks would tattoo slaves' heads and wait for their hair to grow back

- Invisible ink
- Julius Caesar's shift cipher
- Scytale
- o Enigma
- DES/AES
- o RSA





## Steganography (technically not crypto)



- Hiding information in images, videos, music, etc.
- Topic for a separate talk :)

# **Password Cracking**



- Generally found under Crypto categories in CTFs because it deals with hashes
- Topic for a separate talk :)

## Simple Ciphers



- Low point CTF challenges
- Caesar Cipher

Shift cipher: substitute each character by *shifting* it down the alphabet a certain amount (this amount is the

key)

- Key = 3
  - a->d
  - b->r
  - ..
  - Z-> C
- Plaintext: supersecretmessage
- Ciphertext: vxshuvhfuhwphvvdjh
- Atbash
  - o "A" becomes a "Z", "B" turns into "Y"
  - Basically just reversing the alphabet
  - supersecretmessage = hfkvihvxivgnvhhztv



## Simple Ciphers

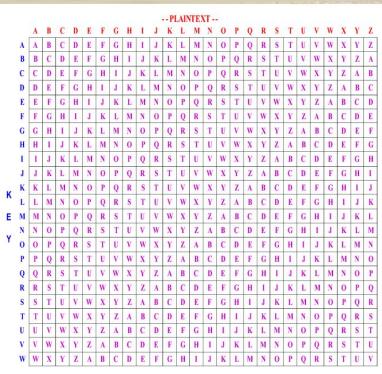


#### Vigenère

- Polyalphabetic substitution
- Match letter of message with letter from a secret key
- o Example:
  - Message: masoncciscool
  - Key: key

masoncciscool keykeykeykeyk -----wegyrammgmsmv

- One-time Pad (OTP)
  - Vigenère where key length = message length
  - Cannot be cracked\*



#### **XOR**



- Exclusive OR
- Used basically everywhere in cryptography
  - Does not leak information about original plaintext
    - AND operation leaks information with: 1 AND 1 = 1
      - Only one possible way the output can be 1
    - OR operation leaks information with: 0 OR 0 = 0
      - Only one possible way the output can be 0
  - Reversible (so if we know the key, we can get our plaintext back)
    - A XOR B = C
    - CXORB=A
    - C XOR A = B

1001(9)	1100(12)
0101(5)	0101(5)
XOR	XOR
1100(12)	1001(9)

Α	В	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

## XOR Challenge Walkthrough



• Ciphertext: 1a00070e050c360b101d0c3611061b

```
ciphertext = "la00070e050c360bl0ld0c36l106lb"

# convert every 2 hex digits into decimal and make an array out of it
dec_arr = [int(ciphertext[i:i+2], 16) for i in range(0, len(ciphertext), 2)]

plaintext = ""
for i in dec_arr:
    # xor decimal with key and convert back into ascii char
    plaintext += chr(i ^ 0x69)

print(plaintext)
```

single\_byte\_xor

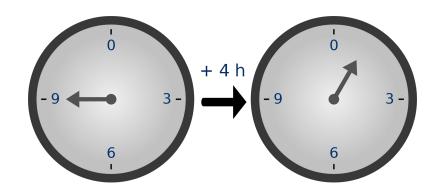
#### Modular Arithmetic (a prereq for RSA)



- Modulo symbol: %
- What it do?
  - Gives you the remainder when dividing one number by another
- Examples:

- Notation
  - Generally written as: 7 = 2 (mod 1)
- Think of a clock (modulo 12)

Modulo makes things loop



#### Modular Arithmetic (in Caesar Cipher)



Let's write the caesar cipher as a modular equation

$$o p + n = c \pmod{26}$$

- p = plaintext character
- $\blacksquare$  n = number to shift by
- c = ciphertext character
- a = 0, b = 1, ..., z = 25
- Example with n=3

$$0 + 3 = 3 \pmod{26}$$

$$0 23 + 3 = 0 \pmod{26}$$

$$0 24 + 3 = 1 \pmod{26}$$



#### Modular Arithmetic (in Vigenère)



- Let's write the Vigenère cipher as a modular equation
  - Encryption

$$P_i + K_i = C_i \pmod{26}$$

Decryption

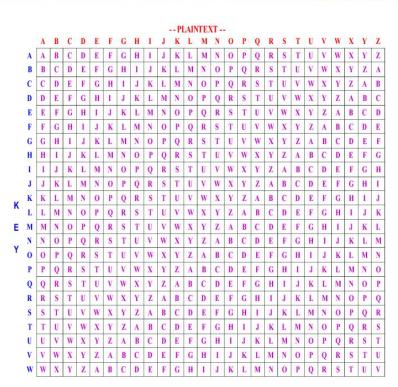
$$C_i - K_i + 26 = P_i \pmod{26}$$

Message: masoncciscool, Key: key

$$0 + 4 = 4 \pmod{26}$$

Decryption

$$\circ$$
 4 - 4 + 26 = 26 = 0 (mod 26)



### **Modular Exponentiation**



- Basis of RSA
- Ex:  $\circ$  2<sup>10</sup> mod 17 = 4
- Python
  - Bad way: 2\*\*10 % 17
     Good way: 2\*\*10 % 17
- pow() is a lot more efficient, if you're interested I'm sure google can tell you why

```
import time

start = time.time()
print((1234123**1123471) % 354456123)
print(f"Elapsed: {time.time() - start}")

start = time.time()
| print(pow(1234123,1123471,354456123))
print(f"Elapsed: {time.time() - start}")
```

```
# python3 time_pow.py
186435118
Elapsed: 3.8073582649230957
186435118
Elapsed: 1.4066696166992188e-05
```

#### GCD (a prereq for RSA)



- Greatest Common Divisor
  - Largest positive integer that divides each of the integers
  - For two integers a, b, the greatest common divisor of a and b is denoted by gcd(a,b)
- Example:
  - $\circ$  gcd(8, 12) = 4
- There are several ways to calculate the GCD, most popular being:
  - Euclidean algorithm
    - Algorithm replaces (a, b) by (b, a mod b) repeatedly until the pair is (d, 0), where d is the greatest common divisor.
    - $\blacksquare$  gcd(48,18) = 6

```
def gcd(a,b):
    if b == 0:
        return a
    return gcd(b, a%b)
```

- o Extended euclidean algorithm
  - $\blacksquare$  Solves: ax + by = gcd(a, b)
  - Will not discuss algorithm here
- Important property: If gcd (a, b) = 1, then a and b are said to be co-prime

#### Modular Multiplicative Inverse



• What is a normal multiplicative inverse?

$$\circ$$
 x \* y<sup>-1</sup> = 1

- *y* is our multiplicative inverse
- Multiply x by something to get 1
- What is a modular multiplicative inverse?
  - $\circ \quad ax \equiv 1 \pmod{m},$ 
    - Integer a such that ax is congruent to 1 with respect to the modulus m
- Example: x = 12 and m = 5
  - $0 12*a \equiv 1 (mod 5)$ 
    - a = 5
  - $0 = 1 \pmod{5}$
- Property: If a has a multiplicative inverse modulo m, this gcd must be 1 [ gcd (a, m) = 1 ]
- Find inverse with extended GCD

```
ax+my=\gcd(a,m)=1. Rewritten, this is ax-1=(-y)m, that is, ax\equiv 1\pmod m,
```

Use egcd() algorithm to find a and that's your modular multiplicative inverse!

```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m
```

## **RSA Encryption**



#### $C = M^e \mod N$

- N
- N is our modulus
- o It is the result of a product of two large primes, p and q (both should be kept secret)
  - $\blacksquare$  N = p\*q
- Part of the public key
- 6
- o e is our *public* exponent
- o Should be coprime with  $(p-1)^*(q-1)$  [also known as totient of N, phi(N),  $\Phi(N)$ ,  $\lambda(n)$ ]
  - $\blacksquare$  gcd(e, phi(N)) = 1
- 65537 is the most common value for e for math reasons
- Part of the public key
- M
  - This is our plaintext message
- (
- This is our resultant ciphertext

Public Key = (N, e)

This is all the information anybody needs to encrypt a message you can decrypt

### **RSA Encryption Example**



So what does this actually look like?

```
from Crypto.Util.number import *
D = 298174781055212569377693674289636639973
q = 18083543398483946245485155451388769483
phi = (p-1)*(q-1)
print(f"phi: {phi}")
n = p*q
print(f"n: {n}")
e = 65537
# make sure e is coprime with phi(n)
if GCD(e, phi) != 1:
    quit()
# convert to byte string to long integer so we can do math
m = bytes to long(m)
print(f"m: {m}")
c = pow(m, e, n)
print(f"\nc: {c}")
```

```
python3 rsa_encrypt.py
phi: 5392056593545385298246637793289447006190888367071557105939999041862134934504
n: 5392056593545385298246637793289447006507146691525253621563177871603160343959
m: 659152916929540644587747821302251083870823868261
c: 4901920022690339146558196061993969247960931292119760747196818899689477731006
```

## **RSA** Decryption



#### $M = C^d \mod N$

- N
  - o N is our modulus
  - Part of the private key
- d
- o d is our *private* exponent
- Modular multiplicative inverse of e mod phi(N)
  - $d = e^{-1} \pmod{phi(n)}$
- Part of the private key
- (
- This is our resultant ciphertext
- M
  - This is our plaintext message

Private Key = (N, d)

This should be kept secret as it will decrypt any message encrypted with the public key

### **RSA** Decryption Example



So what does this actually look like?

```
from Crypto.Util.number import *

p = 298174781055212569377693674289636639973
q = 18083543398483946245485155451388769483
n = p*q
e = 65537

c = 4901920022690339146558196061993969247960931292119760747196818899689477731006

phi = (p-1)*(q-1)
d = inverse(e, phi)
print(f"d: {d}")

m = pow(c,d,n)
print(f"\nm: {long to bytes(m)}")
```

```
# python3 rsa_decrypt.py
d: 3768359493073618527401200912587442998268994140540632903035900882179066851857
m: b'super_secret_message'
```

### RSA: Why is it secure?



- Suppose all we know is the public key (N, e)
- To decrypt a message, we need the private exponent, d

```
o d = e^{-1} \pmod{phi(n)}
o phi(n) = (p-1)*(q-1)
```

- Basically, all we need to do is figure out what p and q are, but how?
  - Prime factorization
- We need to factor N into its prime factors, which are p and q
- This is where RSA security lies
  - Computers don't like integer factorization; it takes forever, especially with super large numbers
    - https://en.wikipedia.org/wiki/Integer factorization
  - If the chosen p and q were large enough (and thus N is large), it is computationally infeasible to factor
     N back into those primes

## RSA: Cracking a Weak Key



- N from previous example:
   5392056593545385298246637793289447006507146691525253621563177871603160343959
- Maybe <a href="http://factordb.com/">http://factordb.com/</a> already knows this one
  - O 18083543398483946245485155451388769483<br/><38> · 298174781055212569377693674289636639973<br/><39>
- This calculator is pretty good: <a href="https://www.alpertron.com.ar/ECM.HTM">https://www.alpertron.com.ar/ECM.HTM</a>
  - After about a minute
    - 18 083543 398483 946245 485155 451388 769483 (38 digits) × 298 174781 055212 569377 693674 289636 639973 (39 digits)
- RsaCtfTool
  - https://github.com/Ganapati/RsaCtfTool

```
# /opt/tools/RsaCtfTool/RsaCtfTool.py -n 5392056593545385298246637793289447006507146691525253621563177871603160343959 -e 65537 --attack all
[*] Testing key /tmp/tmp7 x41bdq.
[*] Performing fibonacci_gcd attack on /tmp/tmp7_x41bdq
                                                                                                                                                                              9999/9999 [00:00<00:00, 126551.68it/s]
                                                                                                                                                                                 113/113 [00:00<00:00, 204998.42it/s]
[*] Performing mersenne primes attack on /tmp/tmp7 x41bdq.
                                                                                                                                                                                 | 12/51 [00:00<00:00, 152983.73it/s]
[*] Performing smallq attack on /tmp/tmp7 x41bdq.
   Performing factordb attack on /tmp/tmp7 x41bdq.
[*] Attack success with factordb method
Results for /tmp/tmp7 x41bdq:
Private key :
 ----BEGIN RSA PRIVATE KEY-----
MIGnAgEAAiAL68u+0Psjtl6v5y6qNyuG7pTr8zP3qDq2kKnMqgxllwIDAQABAiAI
VNDeGNuLh+cA9IbOeyQ10LtyLVqBYm1djcrjnM5yEQIQDZrDhqOPNeJzaPezFdM0
ywIRAOBSX7soG6URoArXZReBBOUCEASkkgH6Y7+B/t0N28JRCC0CEEdDZA49C6tB
EirfisoQvtECEAG48EH1fcXj7pLqlv/kfyc=
 ----END RSA PRIVATE KEY-----
  5392056593545385298246637793289447006507146691525253621563177871603160343959
  3768359493073618527401200912587442998268994140540632903035900882179066851857
  18083543398483946245485155451388769483
  298174781055212569377693674289636639973
Public key details for /tmp/tmp7 x41bdq
 : 5392056593545385298246637793289447006507146691525253621563177871603160343959
```

## **RSA** Keyfiles



```
-# /opt/tools/RsaCtfTool/RsaCtfTool.py -n 5392056593545385298246637793289447006507146691525253621563177871603160343959 -e 65537 --attack all --dumpkey --private
[*] Testing key /tmp/tmp7 x41bdq.
[*] Performing fibonacci_gcd attack on /tmp/tmp7_x41bdq,
                                                                                                                                                                            | 9999/9999 [00:00<00:00, 126551.68it/s]
                                                                                                                                                                               | 113/113 [00:00<00:00, 204998.42it/s]
[*] Performing mersenne primes attack on /tmp/tmp7 x41bdq.
                                                                                                                                                                                | 12/51 [00:00<00:00, 152983.73it/s]
[*] Performing smallq attack on /tmp/tmp7 x41bdq.
[*] Performing factordb attack on /tmp/tmp7 x41bdq.
[*] Attack success with factordb method !
Results for /tmp/tmp7 x41bdq:
Private key :
 ----BEGIN RSA PRIVATE KEY-----
MIGnAgEAAiAL68u+0Psjtl6v5y6qNyuG7pTr8zP3qDq2kKnMqgxllwIDAQABAiAI
VNDeGNuLh+cA9IbOeyQ10LtyLVqBYm1djcrjnM5yEQIQDZrDhqOPNeJzaPezFdM0
ywIRAOBSX7soG6URoArXZReBBOUCEASkkqH6Y7+B/t0N28JRCC0CEEdDZA49C6tB
EirfisoQvtECEAG48EH1fcXj7pLqlv/kfyc=
 ----END RSA PRIVATE KEY----
n: 5392056593545385298246637793289447006507146691525253621563177871603160343959
d: 3768359493073618527401200912587442998268994140540632903035900882179066851857
p: 18083543398483946245485155451388769483
q: 298174781055212569377693674289636639973
Public key details for /tmp/tmp7 x41bdq
n: 5392056593545385298246637793289447006507146691525253621563177871603160343959
```

- This is in PEM format
  - There are others, but you'll probably see PEM more in CTFs

### **Generate RSA Keyfiles**



#### **OpenSSL**

- Generate an RSA private key, of size 2048
  - o \$ openssl genrsa -out key.pem 2048
- Extract the public key from the key pair
  - o \$ openssl rsa -in key.pem -outform PEM -pubout -out public.pem

```
# openssl genrsa -out privkey.pem 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
       t@kali) - [~/Documents/masoncc/crypto]
 ----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAuNEUmrG1qcS5TPOiwaXW8Po6cR5EQ/IG5jhrmFt5bBCq8oBM
4ZAUz1L@cX9ymR5As3KRRv89ZsJ3kQuy/7qoql5DdULe3g/69/B4Kb@R5cdKbvdm
IGxHzYIjsxS3ujmhvMEMVfV8exMtpN8yUsjiv401Y6Vt8R82pR0jWEc5ml8FcVTc
rsGIjRK83ZryWcfpZ8qBfolp0WBE/itsXjsF26gQg3WB01/UnE5TdnoFwwZWXQC/
POMEP6uaKuaz3W+9M3HCjyjOpG10DuEG6iJIz9yLVePKxuuouwYW8UQBAz+D/2kU
fDhj/45U2YTBFHR7IauIrruA/rXtiaIC7/3oGQIDAQABAoIBAQCeP8jYvJnEiAHy
dhtCnPHy3Q/1k+j4F0D7eAwJJSp08eauRlWOPoTTEUDlfi9uFI1qGgtzM1IsDw+6
Tr4yYm3Rk21Hlr6mfiBy9KCvfIk6qu59HbGAybahrXFMAc1f08E9kj8xy0WPTNik
d8jrQb+GLS8t55TABe4Qa2a8TCByoL2kjAYGdv4sGMgW45AxKEBbbavikj9eKQaI
KaQMCj2QWWCtn6iUXxQNH7jLR112VsxIOQTd8dI2HkhkGCQPfMFEEi06FJ2ONymo
t8iaBY1N6X4YUecUG8iLggFDHKgjAS/IVvVL0X1oh/o56KaeufxB+klJogPlleZe
gOsS0rdRAoGBAOB+B8W7BUuVu9U27076cpVfOgs4wN2iPVMpvCP7YdlW85TVi9i6
zBLaBZ2JwFtJ/tHOT8S01VuKABSbnsU+TnKlY0ovEYcZu/Ir8Hl0WYLEdQdqGUjb
HS+oqhdSlNB5/BvpkGMIgyJ3D0mGQMIgmUs/5q5W1Wx11Z4sUpBSrViLAoGBANLB
g2I7TIotYFaU5gkjQlfwoSz3otaqleKWX2yjFrwsImwQSc+bfpGxZ96Mk4Y+Lbw3
zpu4UzgoIw1nkjNO9pZxuFWJ9iSVh179OnNbeqz0w6F5cxpM31S+TD8+yGS1YXdo
GHZQ6jAGMygl+rpl4nj7tLT0KKjttLv0jWQ31HJrAoGAc3UeNjnLrWYjJrOMPMhe
KijCGdGkj9km7yCl3LOrTYaE3GVCCCt6Ta/H95AjWFAkOG+tYvPGT8AX7oFjFPrI
ifo5qKuwjhVULmce1LhcWP8FWXzd6DSZPCu43ynf06EJYqmrUuPL3evx9tSPdcgi
TytcTwnl1lR80098XzHGaaUCgYB8r/pAAp883K2za7JQ06hriLSCexu+7vtwK9DW
4AISX/YZguV4SvFtpsvx809Sa7S5NzcoD7xEbBKY2p63dk8TIMKS80VMLZ8CRsTu
/L8uVjzraoP7mmFCrLdFb6p1Uo4Qa+iVDzg0I4zeguG8p4x+UURPDmiD00ZtkLia
AyHQcwKBgDh85udhEye/hAQfMHnsJRz/rUtyZU5DAu3SA11aZA5W0CoLCOnILIAN
Uhk/Tt7upl0I0KlN0PWqykbwjpmkFwcNcZqVMJKMzoTe2uCAaRAQ+aZ+lmsv+cvT
P7WACE1FiLjrgZzymg6SbbSUGRhNr0EpDfkZme725/0h8goXNrX0
  ----END RSA PRIVATE KEY-----
```

```
## openssl rsa -in privkey.pem -outform PEM -pubout -out pubkey.pem writing RSA key

(root ⊗ kali)-[~/Documents/masoncc/crypto]

# cat pubkey.pem
-----BEGIN PUBLIC KEY-----
MIIBIJANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAUNEUmrG1qcS5TPOiwaXW
8Po6cR5EQ/IG5jhrmFt5bBCq8oBM4ZAUz1L0cX9ymR5As3KRRv89ZsJ3kQuy/7qo
ql5DdULe3g/69/B4Kb0R5cdKbvdmIGxHzYIJsxS3ujmhvMEMVfV8exMtpN8yUsji
v401Y6Vt8R82pR0jWEc5ml8FcVTcrsG1jRK83ZryWcfpZ8qBfolp0WBE/itsXjsF
Z6gQg3WBO1/UnE5TdnoFwwZWXQC/POMEP6uaKuaz3W+9M3HCjyjOpG10DuE66iJI
z9yLVePKxuuouwYW8UQBAz+D/2kUfDhj/45U2YTBFHR7IauIrruA/rXtiaIC7/3o
GQIDAQAB
-----END PUBLIC KEY-----
```

#### Read RSA Keyfiles

- Each RSA format is different, we will focus on PEM
- Get (N,e) from public PEM key
  - \$ openssl rsa -pubin -inform PEM -text -noout < pubkey.pem</pre>
- Get (N,e,d,p,q) from private PEM key
  - \$ openssl rsa -inform PEM -text -noout < privkey.pem</pre>
  - \$ openssl asn1parse -in privkey.pem

```
0:d=0 hl=4 l=1187 cons: SEQUENCE
                                              :00
                                               :B8D1149AB1B5A9C4B94CF3A2C1A5D6F0FA3A711E4443F206E
           D8223B314B7BA39A1BCC10C55F57C7B132DA4DF3252C8E2BF8D3563A56DF11F36A51D235847399A5F0571
67E966B2FF9CBD33FB580084D4588B8EB819CF29A0E926DB49419184DAF41290DF91999EEF6E7FD21F20A1736B5CE
```

```
>>>from Crypto.PublicKey import RSA
>>>f = open("alicepublic.pem", "r")
>>>key = RSA.importKey(f.read())
>>>print key.n #displays n
>>>print key.e #displays e
```

Doing this in the interpreter will display the components.

```
to install Crypto use this: pip install pycryptodome
```

```
opensal rsa -inform
RSA Public-Key: (2048 bit)
                                                                RSA Private-Key: (2048 bit, 2 primes)
    00:b8:d1:14:9a:b1:b5:a9:c4:b9:4c:f3:a2:c1:a5:
    d6:f0:fa:3a:71:1e:44:43:f2:06:e6:38:6b:98:5b:
                                                                    00:b8:d1:14:9a:b1:b5:a9:c4:b9:4c:f3:a2:c1:a5:
    79:6c:10:aa:f2:80:4c:e1:90:14:cf:52:f4:71:7f:
                                                                    d6:f0:fa:3a:71:1e:44:43:f2:06:e6:38:6b:98:5b:
    72:99:1e:40:b3:72:91:46:ff:3d:66:c2:77:91:0b:
                                                                    79:6c:10:aa:f2:80:4c:e1:90:14:cf:52:f4:71:7f:
    b2:ff:ba:a8:aa:5e:43:75:42:de:de:0f:fa:f7:f0:
                                                                    72:99:1e:48:b3:72:91:46:ff:3d:66:c2:77:91:0b:
    78:29:bd:11:e5:c7:4a:6e:f7:66:20:6c:47:cd:82:
                                                                    b2:ff:ba:a8:aa:5e:43:75:42:de:de:@f:fa:f7:f0:
    23:b3:14:b7:ba:39:a1:bc:c1:0c:55:f5:7c:7b:13:
                                                                    78:29:bd:11:e5:c7:4a:6e:f7:66:20:6c:47:cd:82:
    2d:a4:df:32:52:c8:e2:bf:8d:35:63:a5:6d:f1:1f:
                                                                    23:b3:14:b7:ba:39:a1:bc:c1:0c:55:f5:7c:7b:13:
    36:a5:1d:23:58:47:39:9a:5f:05:71:54:dc:ae:c1:
                                                                    2d:a4:df:32:52:c8:e2:bf:8d:35:63:a5:6d:f1:1f:
    88:8d:12:bc:dd:9a:f2:59:c7:e9:67:ca:81:7e:89:
                                                                    36:a5:1d:23:58:47:39:9a:5f:05:71:54:dc:ae:c1:
    69:d1:60:44:fe:2b:6c:5e:3b:05:db:a8:10:83:75:
    81:3b:5f:d4:9c:4e:53:76:7a:05:c3:06:56:5d:00:
                                                                    88:8d:12:bc:dd:9a:f2:59:c7:e9:67:ca:81:7e:89:
    bf:3c:e3:04:3f:ab:9a:2a:e6:b3:dd:6f:bd:33:71:
                                                                    69:d1:60:44:fe:2b:6c:5e:3b:05:db:a8:10:83:75:
    c2:8f:28:ce:a4:6d:4e:0e:e1:06:ea:22:48:cf:dc:
                                                                    81:3b:5f:d4:9c:4e:53:76:7a:05:c3:06:56:5d:00:
    8b:55:e3:ca:c6:eb:a8:bb:06:16:f1:44:01:03:3f:
                                                                    bf:3c:e3:04:3f:ab:9a:2a:e6:b3:dd:6f:bd:33:71:
    83:ff:69:14:7c:38:63:ff:8e:54:d9:84:c1:14:74:
                                                                    c2:8f:28:ce:e4:6d:4e:0e:e1:06:ee:22:48:cf:dc:
    7b:21:ab:88:ae:bb:80:fe:b5:ed:89:a2:02:ef:fd:
                                                                    8b:55:e3:ca:c6:eb:a8:bb:06:16:f1:44:01:03:3f:
                                                                    83:ff:69:14:7c:38:63:ff:8e:54:d9:84:c1:14:74:
 xponent: 65537 (0x10001)
                                                                    7b:21:ab:88:ae:bb:80:fe:b5:ed:89:a2:02:ef:fd:
                                                                 rivateExponent:
                                                                    00:9e:3f:c8:d8:bc:99:c4:88:01:f2:76:1b:42:9c:
```

```
Sams-MacBook-Pro-3:~ sambowne$ openssl genrsa -out key.pem
Generating RSA private key, 512 bit long modulus
......
...++++++++++
e is 65537 (0x10001)
Sams-MacBook-Pro-3:~ sambowne$ cat key.pem
 ----BEGIN RSA PRIVATE KEY-----
MIIBOgIBAAJBAOFOVAZF1ridu0Q3vkc6g5MUoD7vgNtDZ9U0Km4aqRHnxtu0w5/v
ecGMLmkv4NBj3SakVvEaFq8iDpkfnS3i810CAwEAAQJAPj7fo+QDkHmzVQN5hEA8
PZRDOV/935Xdx99ioYuoDhPPwu3af0afP0045gPedL/eiZpEJ00WKLNaZT9guTpr
AQIhAP7cfT3Ihn7sdJq032tePEvx9KbyIvFjZvHDAe0cPbkhAiEA4lAImGEbm5GW
Ezri+r1BkJAY8Pw4FHWW0QsG8+Yqhr0CIQCGr2sYXYKGTNuJCEMV8KUY1XMfTtMc
khVzKkU4UzZjQQIgXhtRt6uJw2MCuPRftkxEm0yQUoV0/JL5J+wh9AHd5TECICMb
k80ogKZrqIuQpo4LGOAcXt4KOmDcOXQ+IdtSSm3B
 ----END RSA PRIVATE KEY----
|Sams-MacBook-Pro-3:~ sambowne$ openssl asn1parse -in key.pem
    0:d=0 hl=4 l= 314 cons: SEQUENCE
    4:d=1 hl=2 l= 1 prim: INTEGER
    7:d=1 hl=2 l= 65 prim: INTEGER
                                            n: E14E540645D6B89DBB4437BE473A839314A03EEF80DB4367D5342A6E1AA911E7C6DB8
EC39FEF79C18C2E692FE0D063DD26A456F11A16AF220E991F9D2DE2F35D
   74:d=1 hl=2 l= 3 prim: INTEGER
                                            e :010001
   79:d=1 hl=2 l= 64 prim: INTEGER
                                            d :3E3EDFA3E4039079B355037984403C3D9443395FFDDF95DDC7DF63A18BA80E13CFC2E
DDA7D069F3D0D38E6A3DE74BFDE899A4427441628B35A653F60B93A6B01
 145:d=1 hl=2 l= 33 prim: INTEGER
                                            :FEDC7D3DC8867EEC74980EDF6B5E3C4BF1F4A6F222F16366F1C301ED1C3DB921
  180:d=1 hl=2 l= 33 prim: INTEGER
                                            E2500898611B9B9196133AE2FABD41909018F0FC38147596D10B06F3E62A86BD
 215:d=1 hl=2 l= 33 prim: INTEGER
                                              :86AF6B185D82864CDB89084315F0A518D5731F4ED31C9215732A453853366341
 250:d=1 hl=2 l= 32 prim: INTEGER
                                              :5E1B51B7AB89C36302B8F45FB64C449B4C9052854EFC92F927EC21F401DDE531
 284:d=1 hl=2 l= 32 prim: INTEGER
                                              :231B93CD28A8A66B808B90A68E0B18E01C5EDE0A3A60DC39743E21DB524A6DC1
```

```
publicExponent: 65537 (0x10001)
```

```
f1:f2:dd:0f:f5:93:e8:f8:14:e0:fb:78:0c:09:25:
Za:4e:f1:e0:ae:40:55:8e:3e:84:d3:11:40:e5:7e:
2f:6e:14:8d:6a:1a:0b:73:33:52:2c:0f:0f:ba:4e:
be:32:62:6d:d1:93:6d:47:96:be:a6:7e:20:72:f4:
a0:af:7c:89:3a:aa:ee:7d:1d:b1:80:c9:b5:a1:ad:
71:4c:01:cd:5f:d3:c1:3d:92:3f:31:cb:45:8f:4c:
d8:a4:77:c8:eb:41:bf:80:2d:2f:2d:e7:94:c0:05:
ee:10:6b:66:bc:4c:20:72:a0:bd:a4:8c:06:06:76:
fe:2c:18:c8:16:e3:90:31:28:40:5b:6d:ab:e2:92:
3f:5e:29:06:88:29:04:0c:0a:3d:90:59:60:ad:9f:
04:dd:f1:d2:36:le:48:64:18:24:0f:7c:c1:44:12:
2d:3a:14:9d:8e:37:29:a8:b7:c8:9a:05:8d:4d:e9:
7e:18:51:e7:14:1b:c8:8b:82:01:43:1c:a8:23:01:
2f:c8:50:f5:4b:d1:7d:68:87:fa:39:e8:a0:9e:b9:
fc:41:fa:49:49:a2:03:e5:95:e0:5e:a8:eb:12:d2:
```

```
primel:
   00:e0:7e:07:c5:bb:05:4b:05:bb:d5:36:ef:4e:fa:
   72:95:5f:3a:ab:38:c0:dd:a3:3d:53:29:bc:23:fb:
   61:d9:56:f3:94:d5:8f:d8:ba:cc:12:da:05:9d:89:
   c0:5b:49:fe:d1:ce:4f:c4:b4:d5:5b:8a:00:14:9b:
   9e:c5:3e:4e:72:a5:63:4a:2f:11:87:19:bb:f2:2b:
   f0:79:74:59:82:c4:75:07:0a:19:48:db:1d:2f:a8:
   aa:17:52:94:d0:79:fc:1b:e9:90:03:08:83:22:77:
   0f:49:85:40:c2:20:99:4b:3f:e5:ae:55:d5:6c:75:
```

```
d5:9e:2c:52:98:52:ad:58:8b
```

00:d2:c1:83:62:3b:4c:8a:2d:60:56:94:e6:09:23: 42:57:f0:a1:2c:f7:a2:d0:aa:95:e2:96:5f:6c:a3: 16:bc:2c:22:6c:10:49:cf:9b:7e:91:b1:67:de:8c: 93:86:3e:2d:bc:37:ce:9b:b8:53:38:28:23:0d:07: 92:33:4e:f6:96:71:b8:55:89:f6:24:95:87:5e:fd: 3a:73:5b:7a:ac:f4:c3:a1:79:73:1a:4c:df:54:be: 4c:3f:3e:c8:64:b5:61:77:68:18:76:50:ea:30:06: 33:28:25:fa:ba:65:e2:78:fb:b4:b4:f4:28:a8:ed: b4:bb:f4:8d:64:37:d4:72:6b

#### **RSA Attacks**



- Weak public key factorization
- Wiener's attack
- Hastad's attack (Small public exponent attack)
- Small q (q < 100,000)</li>
- Common factor between ciphertext and modulus attack
- Fermat's factorisation for close p and q
- Gimmicky Primes method
  - Past CTF Primes method
- Self-Initializing Quadratic Sieve (SIQS) using Yafu (https://github.com/DarkenCode/yafu.git)
- Common factor attacks across multiple keys
- Small fractions method when p/q is close to a small fraction
- Boneh Durfee Method when the private exponent d is too small compared to the modulus (i.e d < n^0.292)</li>
  - Elliptic Curve Method
- Pollards p-1 for relatively smooth numbers
- Mersenne primes factorization
- Factordb
- Londahl
- Noveltyprimes
- Partial q
- Primefac
- Oicheng
- Same n, huge e
- binary polynomial factoring
- Euler method
- \_\_\_\_\_
- Pollard Rho
- Wolfram alpha
- cm-factor
- z3 theorem prover
- Primorial pm1 gcd
- Mersenne pm1 gcd
- Fermat Numbers gcd
- Fibonacci gcd
- System primes gcd
- Small crt exponent
- Shanks's square forms factorization (SQUFOF)
- Return of Coppersmith's attack (ROCA) with NECA variant
- Dixon
- brent (Pollard rho variant)
- Pisano Period
- NSIF Vulnerability, Power Modular Factorization, Near Power Factors

#### Resources



- https://cryptohack.org/
- https://www.youtube.com/channel/UC1usFRN4LCMcflV7UjHNuQg/videos
- https://ctf101.org/cryptography/overview/
- <u>https://www.geeksforgeeks.org/euclidean-algorithms-basic-and-extended/</u>
- https://github.com/Ganapati/RsaCtfTool
- http://factordb.com/
- https://www.alpertron.com.ar/ECM.HTM

#### What I Left Out



- Stream Ciphers
- Block Ciphers

These are important, but I don't have the bandwidth to teach myself them at the moment :) Topic for another talk anybody?

# **Challenge Time**



- <a href="https://tctf.competitivecyber.club/challenges">https://tctf.competitivecyber.club/challenges</a>
- Classical Cryptography
  - Prisoner
  - All Around The World
- RSA Corner
  - Common Modulo

# **Proud Sponsors**



