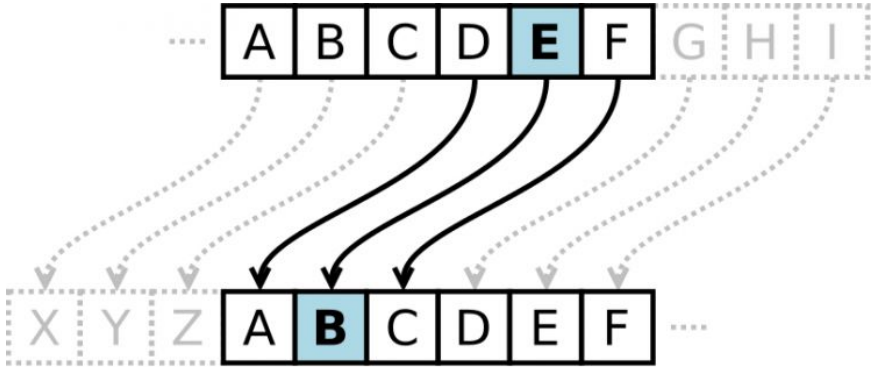
Mason Competitive Cyber

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
  - Enigma
  - DES/AES
  - 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

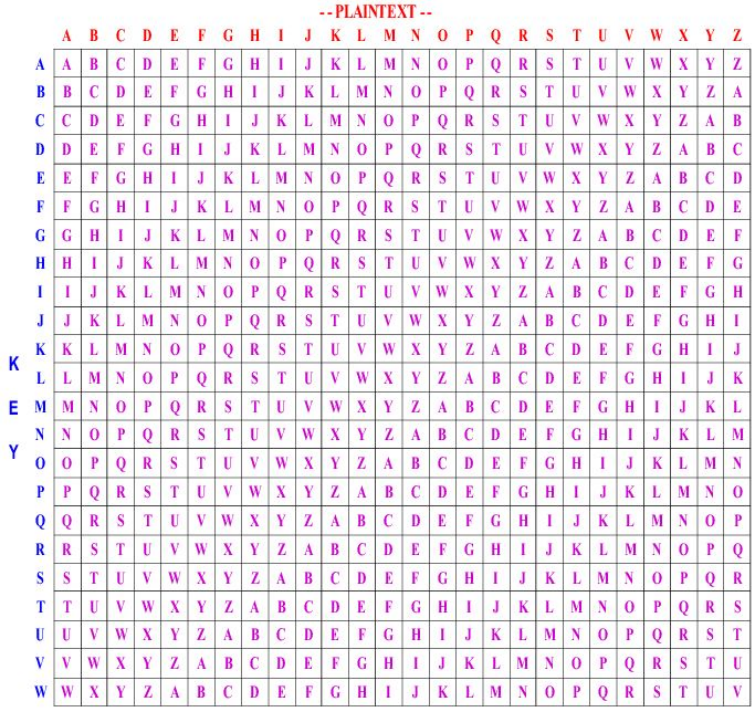
- Low point CTF challenges
- Caesar Cipher
  - Shift cipher: substitute each character by *shifting* it down the alphabet a certain amount (this amount is the key)
  - Key = 3
    - a -> d
    - b -> r
    - ...
    - z -> c
  - Plaintext: *supersecretmessage*
  - Ciphertext: *vxshuvhfuhwphvvdjh*
- Atbash
  - "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
  - Example:
    - Message: *masoncciscool*
    - Key: *key*

```
masoncciscool
keykeykeykeyk
-------------
weqyrammqmsmv
```

- 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
    - C XOR B = A
    - C XOR A = B

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

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

XOR Challenge Walkthrough

- Ciphertext: `1a00070e050c360b101d0c3611061b`

```python
from binascii import hexlify, unhexlify

plaintext = "[redacted]"

arr = []
for c in plaintext:
    arr.append(ord(c) ^ 0x69)

ciphertext = hexlify(bytearray(arr)).decode("UTF-8")
print(ciphertext)
```

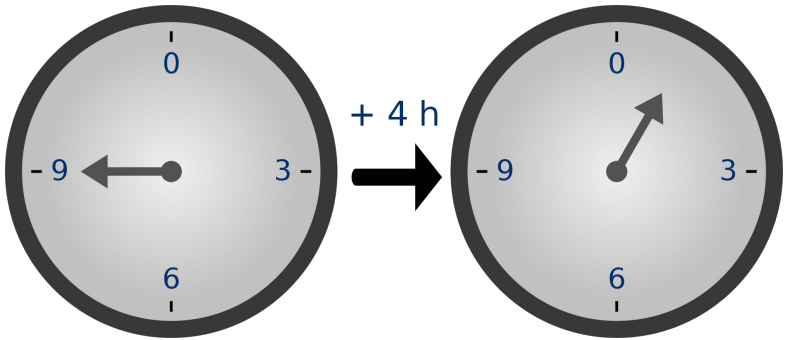
```python
ciphertext = "1a00070e050c360b101d0c3611061b"

# 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)
```

```
[26, 0, 7, 14, 5, 12, 54, 11, 16, 29, 12, 54, 17, 6, 27]
```

```
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:
  - 7 % 2 = 1
  - 9 % 3 = 0
  - 14 % 5 = 4
  - 4 % 9 = 4
  - -4 % 9 = 5
- Notation
  - Generally written as: 7 = 2 (mod 1)
- Think of a clock (modulo 12)
  - 9 + 4 = 13
  - 13 mod 12 = 1
- Modulo makes things loop

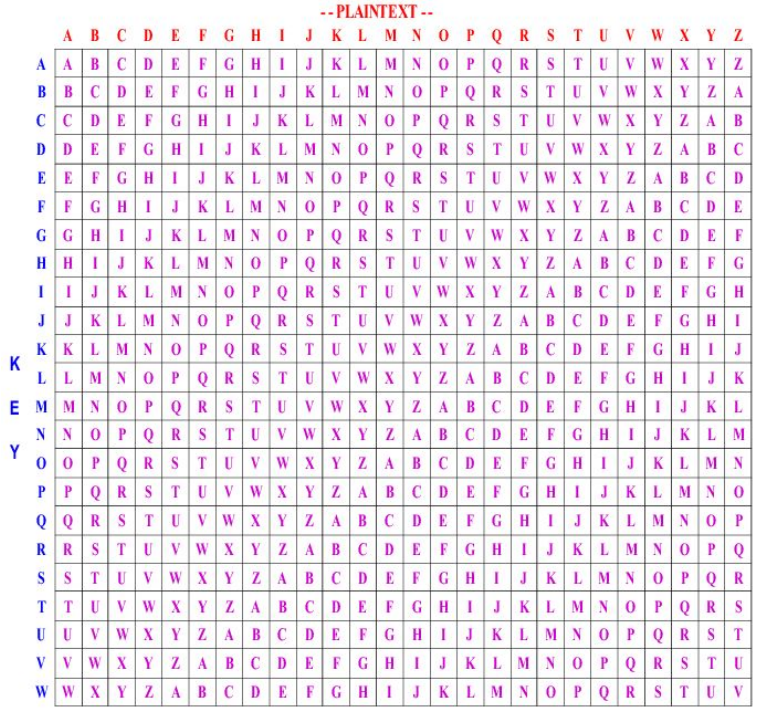


Modular Arithmetic (in Caesar Cipher)

- Let's write the caesar cipher as a modular equation
  - `p + n = c (mod 26)`
    - p = plaintext character
    - n = number to shift by
    - c = ciphertext character
- a = 0, b = 1, ..., z = 25
- Example with n=3
  - `0 + 3 = 3 (mod 26)`
    - a -> d
  - `23 + 3 = 0 (mod 26)`
    - x -> a
  - `24 + 3 = 1 (mod 26)`
    - y -> b

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*
  - 12 + 10 = 22 (mod 26)
    - m -> w
  - 0 + 4 = 4 (mod 26)
    - a -> e
  - 18 + 24 = 42 = 16 (mod 26)
    - s -> q
- Decryption
  - 22 - 10 + 26 = 38 = 12 (mod 26)
    - w -> m
  - 4 - 4 + 26 = 26 = 0 (mod 26)
    - e -> a
  - 16 - 24 + 26 = 18 (mod 26)
    - q -> s



KEY \ --PLAINTEXT--	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V

Modular Exponentiation

- Basis of RSA
- Ex:
  - 2^{10} mod 17 = 4
- Python
  - Bad way:

```
>>> 2**10 % 17
4
```

  - Good way:

```
>>> pow(2,10,17)
4
```

- 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:
  - `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.
    - `gcd(48,18) = 6`

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

    return gcd(b, a%b)
```

$$\begin{aligned} \gcd(48, 18) &\rightarrow \gcd(18, 48 \bmod 18) = \gcd(18, 12) \\ &\rightarrow \gcd(12, 18 \bmod 12) = \gcd(12, 6) \\ &\rightarrow \gcd(6, 12 \bmod 6) = \gcd(6, 0). \end{aligned}$$

  - Extended euclidean algorithm
    - 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?
  - $x * y^{-1} = 1$
    - y is our multiplicative inverse
    - Multiply x by something to get 1
- What is a modular multiplicative inverse?
  - $ax \equiv 1 \pmod{m},$
    - Integer a such that ax is congruent to 1 with respect to the modulus m
- Example: $x = 12$ and $m = 5$
  - $12*a \equiv 1 \pmod 5$
    - $a = 5$
  - $60 \equiv 1 \pmod 5$ ✅
- 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!

```python
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

$$C = M^e \bmod N$$

- N
  - N is our modulus
  - It is the result of a product of two large primes, p and q (both should be kept secret)
    - `N = p*q`
  - Part of the public key
- e
  - e is our *public* exponent
  - Should be coprime with (p-1)*(q-1) [also known as totient of N, `phi(N)`, `Φ(N)`, `λ(n)`]
    - `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
- C
  - 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?

```python
from Crypto.Util.number import *

p = 298174781055212569377693674289636639973
q = 180835433984839462454851554513887694483
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()

m = b"super_secret_message"
# 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: 53920565935453852982466377932894470061908883670715571059399990418621349345040
n: 5392056593545385298246637793289447006507146691525253621563177871603160343959
m: 659152916929540644587747821302251083870823868261

c: 490192002269033914655819606199396924796093129211976074719681889968947773100 6
```

RSA Decryption

$$M = C^d \bmod N$$

- N
  - N is our modulus
  - Part of the private key
- d
  - d is our *private* exponent
  - Modular multiplicative inverse of e mod phi(N)
    - $d = e^{-1} \pmod{phi(n)}$
  - Part of the private key
- C
  - 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 = 180835433984839462454851554513887694 83
n = p*q
e = 65537

c = 490192002269033914655819606199396924796093129211976074719681889968947773 1006

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: 376835949307361852740120091258744299826899414054063290303590088217906685 1857

m: b'super_secret_message'
```

RSA: Why is it secure?

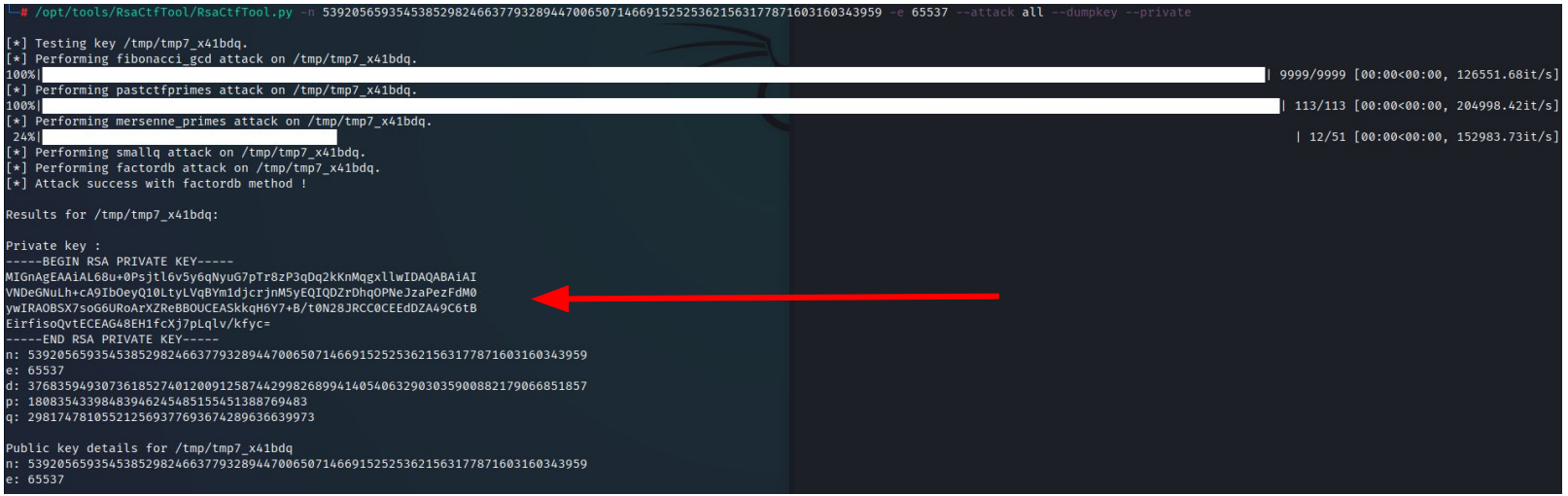
- Suppose all we know is the public key (N, e)
- To decrypt a message, we need the private exponent, d
  - $d = e^{-1} \pmod{\phi(n)}$
  - $\phi(n) = (p-1)*(q-1)$
- Basically, all we need to do is figure out what p and q are, but how?
  - Prime factorization
- We need to factor N into its prime factors, 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:
  539205659354538529824663779328944700650714669152525362156317787160316034395 9
- Maybe http://factordb.com/ already knows this one
  - $180835433984839462454851554513887694 83_{<38>} \cdot 298174781055212569377693674289636639973_{<39>}$
- This calculator is pretty good: https://www.alpertron.com.ar/ECM.HTM
  - 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 539205659354538529824663779328944700650714669152525362156317787160316034395 9 -e 65537 --attack all --dumpkey --private

[*] Testing key /tmp/tmp7_x41bdq.
[*] Performing fibonacci_gcd attack on /tmp/tmp7_x41bdq.
100%|████████████████████████████████████████████████████████████████████████████| 9999/9999 [00:00<00:00, 126551.68it/s]
[*] Performing pastctfprimes attack on /tmp/tmp7_x41bdq.
100%|████████████████████████████████████████████████████████████████████████████| 113/113 [00:00<00:00, 204998.42it/s]
[*] Performing mersenne_primes attack on /tmp/tmp7_x41bdq.
 24%|███████                                                                       | 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: 539205659354538529824663779328944700650714669152525362156317787160316034395 9
e: 65537
d: 376835949307361852740120091258744299826899414054063290303590088217906685 1857
p: 180835433984839462454851554513887694 83
q: 298174781055212569377693674289636639973

Public key details for /tmp/tmp7_x41bdq
n: 539205659354538529824663779328944700650714669152525362156317787160316034395 9
e: 65537
```

RSA Keyfiles



- 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
  - `$ openssl genrsa -out key.pem 2048`
- Extract the public key from the key pair
  - `$ 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)
.....................................................................................+++++
.....................................................+++++
e is 65537 (0x010001)

┌──(root💀kali)-[~/Documents/masoncc/crypto]
└─# cat privkey.pem
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAuNEUmrG1qcS5TPOiwaXW8Po6cR5EQ/IG5jhrmFt5bBCq8oBM
4ZAUz1L0cX9ymR5As3KRRv89ZsJ3kQuy/7qoql5DdULe3g/69/B4Kb0R5cdKbvdm
IGxHzYIjsxS3ujmhvMEMVfV8exMtpN8yUsjiv401Y6Vt8R82pR0jWEc5ml8FcVTc
rsGIjRK83ZryWcfpZ8qBfolp0WBE/itsXjsF26gQg3WBO1/UnE5TdnoFwwZWXQC/
POMEP6uaKuaz3W+9M3HCjyjOpG1ODuEG6iJIz9yLVePKxuuouwYW8UQBAz+D/2kU
fDhj/45U2YTBFHR7IauIrruA/rXtiaIC7/3oGQIDAQABAoIBAQCeP8jYvJnEiAHy
dhtCnPHy3Q/1k+j4FOD7eAwJJSpO8eauRlWOPoTTEUDlfi9uFI1qGgtzM1IsDw+6
Tr4yYm3Rk21Hlr6mfiBy9KCvfIk6qu59HbGAybahrXFMAc1f08E9kj8xy0WPTNik
d8jrQb+GLS8t55TABe4Qa2a8TCByoL2kjAYGdv4sGMgW45AxKEBbbavikj9eKQaI
KaQMCj2QWWCtn6iUXxQNH7jLR112VsxIOQTd8dI2HkhkGCQPfMFEEi06FJ2ONymo
t8iaBY1N6X4YUecUG8iLggFDHKgjAS/IVvVL0X1oh/o56KaeufxB+klJogPlleZe
qOsS0rdRAoGBAOB+B8W7BUuVu9U27076cpVfOqs4wN2jPVMpvCP7YdlW85TVj9i6
zBLaBZ2JwFtJ/tHOT8S01VuKABSbnsU+TnKlY0ovEYcZu/Ir8Hl0WYLEdQdqGUjb
HS+oqhdSlNB5/BvpkGMIgyJ3D0mGQMIgmUs/5q5W1Wx11Z4sUpBSrViLAoGBANLB
g2I7TIotYFaU5gkjQlfwoSz3otaqleKWX2yjFrwsImwQSc+bfpGxZ96Mk4Y+Lbw3
zpu4UzgoIw1nkjNO9pZxuFWJ9iSVh179OnNbeqz0w6F5cxpM31S+TD8+yGS1YXdo
GHZQ6jAGMygl+rpl4nj7tLT0KKjttLv0jWQ31HJrAoGAc3UeNjnLrWYjJrOMPMhe
KijCGdGkj9km7yCl3LOrTYaE3GVCCCt6Ta/H95AjWFAkOG+tYvPGT8AX7oFjFPrI
ifo5qKuwjhVULmce1LhcWP8FWXzd6DSZPCu43ynf06EJYqmrUuPL3evx9tSPdcgi
TytcTwnl1lR8O98XzHGaaUCgYB8r/pAAp883K2za7JQ06hriLSCexu+7vtwK9DW
4AISX/YZguV4SyFtpsyx8O9Sq7S5NzcoD7xEbBKY2p63dk8TIMKS8OVMLZ8CRsTu
/L8uVjzraoP7mmFCrLdFb6p1Uo4Qa+iVDzgOI4zeguG8p4x+UURPDmiD0OZtkLia
AyHQcwKBgDh85udhEye/hAQfMHnsJRz/rUtyZU5DAu3SA11aZA5W0CoLCOnILIAN
Uhk/Tt7uplOI0KlN0PWqykbwjpmkFwcNcZqVMJKMzoTe2uCAaRAQ+aZ+lmsv+cvT
P7WACE1FiLjrgZzymg6SbbSUGRhNr0EpDfkZme725/0h8goXNrXO
-----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
v401Y6Vt8R82pR0jWEc5ml8FcVTcrsGIjRK83ZryWcfpZ8qBfolp0WBE/itsXjsF
26gQg3WBO1/UnE5TdnoFwwZWXQC/POMEP6uaKuaz3W+9M3HCjyjOpG1ODuEG6iJI
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`
- Get (N,e,d,p,q) from private PEM key
  - `$ openssl rsa -inform PEM -text -noout < privkey.pem`
    - OR
  - `$ openssl asn1parse -in privkey.pem`

```
>>>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`

RSA Attacks

- Weak public key factorization
- Wiener's attack
- Hastad's attack (Small public exponent attack)
- Small q (q < 100,000)
- 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)
- Elliptic Curve Method
- Pollards p-1 for relatively smooth numbers
- Mersenne primes factorization
- Factordb
- Londahl
- Noveltyprimes
- Partial q
- Primefac
- Qicheng
- 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/UC1usFRN4LCMcfIV7UjHNuQg/videos
- https://ctf101.org/cryptography/overview/
- https://www.geeksforgeeks.org/euclidean-algorithms-basic-and-extended/
- 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

- https://tctf.competitivecyber.club/challenges
- Classical Cryptography
  - Prisoner
  - All Around The World
- RSA Corner
  - Common Modulo

Proud Sponsors

CACI

EVER VIGILANT