MISC

# INTRO TO CRYPTOGRAPHY

Sem 2, 2019 Workshop 2

# Challenges (in Discord)

Challenges are available via the Discord bot.

Quick start:

- `$challs` lists the challenges
- `$chall <challid>` displays details for a challenge
- `$submit <challid> <flag>` submits flag (only available via dm)

Invite link: https://discord.gg/ph2sseW

Slides and copies of the challenges are also available on GitHub: https://github.com/umisc/workshops/tree/master/Cryptography

# Notation and Functions

$a$ and $b$ are congruent mod $c$ implies that $c$ divides $a - b$

$a \equiv b \pmod{c} \implies c \mid (a - b)$

The notation: $a \bmod b$ is sometimes used to denote the value of the remainder when $b$ divides $a$ (i.e. $7 \bmod 4 = 3$)

See: [modular arithmetic]

$\gcd(a, b)$ denotes the largest positive integer that divides both $a$ and $b$

See: [gcd]

# Notation and Functions (continued)

Functions from the PyCrypto library:

Import: `from Crypto.Util.number import *`

`bytes_to_long` - converts bytes to an integer by creating a hex string from the hex representation of the bytes

```
bytes_to_long(b'a') == 0x61 == 97
bytes_to_long(b'ab') == 0x6162 == 24930
```

`long_to_bytes` - inverse function of `bytes_to_long`

```
long_to_bytes(24930) == b'ab'
```

# Notation and Functions (continued)

`inverse` - calculates the modular inverse of the first argument mod the second

```
inverse(3, 17) == 6 # x such that 3x is congruent to 1 (mod 17)
```

`GCD` - calculates the gcd of two numbers

```
GCD(3, 7) == 1
```

`pow` - exponentiation/modular exponentiation function (Python in-built)

```
pow(2, 3) == 8
pow(2, 3, 5) == 3 # (2**3)%5 or pow(2,3)%5 except more efficient
```

# What is Cryptography?

**Cryptography** is the practice and study of techniques for secure communication in the presence of third parties.

# Functions of Cryptography

Cryptography is used for:

1. **Privacy**: Ensuring only intended recipients can read messages
2. **Authentication**: Proving one's identity
3. **Integrity**: Ensuring receieved messages have not been altered or corrupted.
4. **Non-repudiation**: Proving authorship of a message

# Types of Cryptograhy

- Symmetric-key Cryptography
- Public-key Cryptography (asymmetric)
- Hashing

# Symmetric-key Cryptography

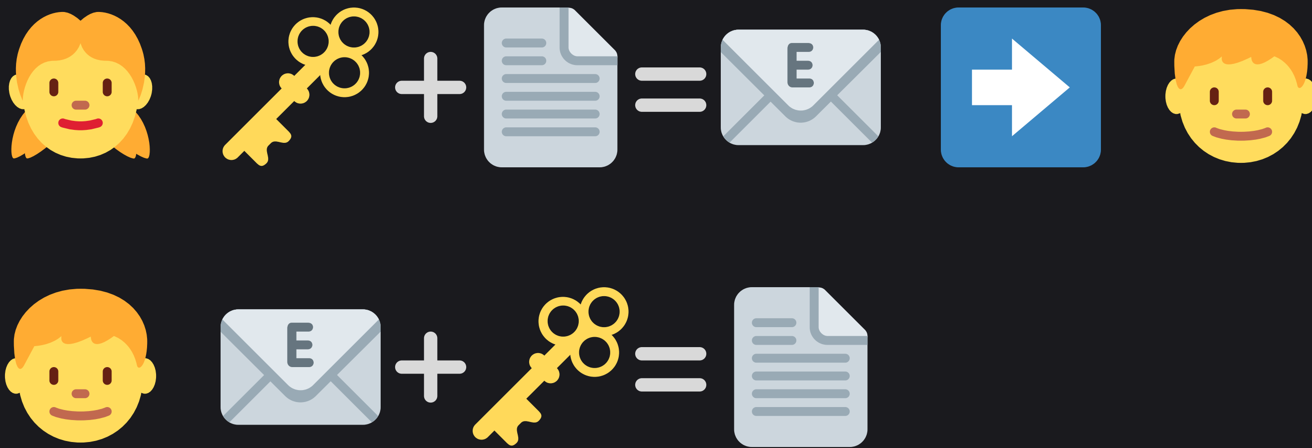
Symmetric-key cryptography involves two parties using the same shared key for encryption and decryption.

This shared key needs to be kept secret and distributed securely.

Some examples include:

- Caesar Cipher
- Single byte XOR
- Block Ciphers
- Stream Ciphers

# Symmetric-key Cryptography (continued)



# Key Exchange

Symmetric-key cryptography relies on both parties having a shared key.

This key needs to be known by the two parties without anyone else being able to obtain a copy.

There are methods that utilise trapdoor functions to aid with the task of key exchange. One such method is Diffie-Hellman.

# Public-key Cryptography

Public-key cryptography involves two parties using two public-key/private-key pairs to encrypt and decrypt messages between each other.

For instance, say Bob wants to send a message to Alice.

Alice generates a public and private key and sends the public key to Bob.

Bob uses this public key to encrypt his message, and sends it back to Alice, who can decrypt it with her public key.

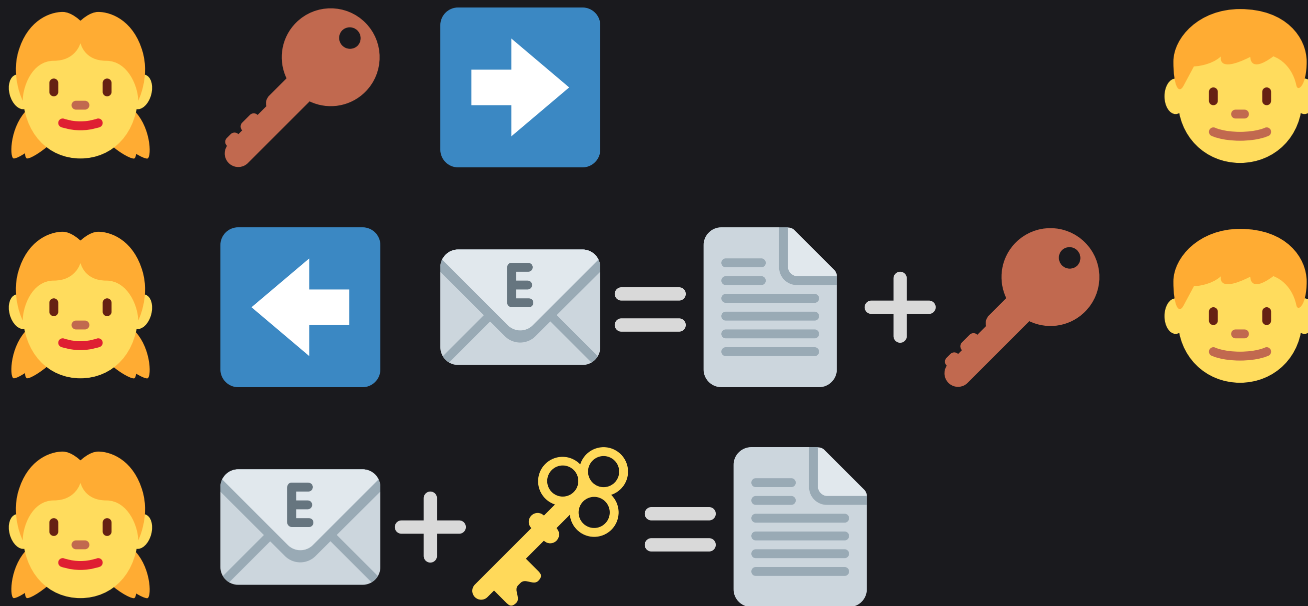
If Alice wanted to send a message to Bob, Bob would generate his own public and private key pair and send the public key to Alice.

# Public-key Cryptography (continued)

The main idea is that the private key is mathematically related to the public key so that the private key can be used to decrypt a text encrypted with the public key, but this mathematical relation needs to be such that the private key cannot be feasibly determined from the public key.

A prime example of a public-key cryptosystem is RSA.

# Public-key Cryptography (continued)



# Principles of Cryptography

We could achieve a certain level of security by transforming our message into some garbled form such that only the intended recipient can understand it.

For example:

My friend and I can agree on a protocol for encrypting and decrypting messages beforehand; if we decide that we'll encrypt messages by shifting each letter in the message by 1, and decrypt by doing the opposite of that, we can send messages without anyone ever finding out what we're saying!

```
encrypt('hello') -> 'ifmmp'
```

```
decrypt('ifmmp') -> 'hello'
```

This isn't very secure though...

Instead, we could try using a secret key that we use in our encrypt and decrypt algorithms, so even if eavesdroppers know *how* to decrypt our messages, they won't be able to without the key. Instead of shifting by 1 letter, we could jumble up the letters (substitution cipher). So if the substitution alphabet (the key) was `azertyuiopqsdfghjklmwxcvbn`, the encrypt and decrypt functions would work as follows:

```
encrypt('hello') -> 'itssg'
```

```
decrypt('itssg') -> 'hello'
```

However, if the eavesdropper knows what kind of messages we're sending, this cipher can be quite unsecure too.

# RSA Overview

RSA is one of the first public-key cryptosystems and is still used today.

Its security lies in the difficulty of factorising the product of two large prime numbers.

# RSA Key Generation

## Public Key

The public key consists of two numbers: $(n, e)$

$n$ is formed from taking the product of two large primes $p$ and $q$

$e$ is any number such that $\gcd(e, \phi(n)) = 1$. ($e = 65537$ is common.)

## Private Key

The private key also consists of two numbers: $(n, d)$

where $d \equiv e^{-1} \pmod{\phi(n)}$ (i.e. $d$ such that $de \equiv 1 \pmod{\phi(n)}$)

# RSA Key Generation (Notes)

Euler's totient function $\phi(n)$ counts the positive integers up to $n$ that don't share any factors besides $1$. ( e.g. $\phi(9) = 6$ )

If $p$ and $q$ are assumed prime, then $\phi(pq) = (p-1)(q-1)$.

# RSA Key Generation (Notes)

Euler's totient function $\phi(n)$ counts the positive integers up to $n$ that don't share any factors besides $1$. ( e.g. $\phi(9) = 6$ )

If $p$ and $q$ are assumed prime, then $\phi(pq) = (p-1)(q-1)$.

This follows from the fact that all multiples of $p$ will share a factor with $pq$ and all multiples of $q$ will share a factor with $pq$.

$1, 2, \ldots, p, \ldots, q, \ldots, 2p, \ldots, 2q, \ldots, pq = n$

$pq - p - q + 1 = (p-1)(q-1)$

# RSA Algorithm

## Encryption

If $m$ is the plaintext message, the ciphertext $c$ is calculated as follows:

$$c = m^e \bmod n$$

## Decryption

Decryption of a ciphertext $c$ using the private key $n, d$ is done as follows:

$$m = c^d \bmod n$$

# Textbook RSA Example

```python
from Crypto.Util.number import getPrime, inverse, bytes_to_long

message = b'hello!'
m = bytes_to_long(message)

# key generation
p, q = getPrime(512), getPrime(512)
n = p * q
e = 0x10001

# encryption
c = pow(m, e, n) # ciphertext

# decryption
phi = (p - 1)*(q - 1)
d = inverse(e, phi)
decrpyted_message = long_to_bytes(pow(c, d, n)) # b'hello!'
```

# Another RSA Example

Can you find the vulnerability?

```python
from Crypto.Util.number import bytes_to_long, getPrime
from secret import secret_message

p1 = p2 = getPrime(512)
q1, q2 = getPrime(512), getPrime(512)

n1 = p1*q1
n2 = p2*q2

m = bytes_to_long(secret_message)

c1 = pow(m, e, n1)
c2 = pow(m, e, n2)
```

You know the values of the public keys $(n_1, e), (n_2, e)$ and the ciphertexts $c_1$ and $c_2$.

# Solution

Since $p_1 = p_2$, it follows that $n_1 = p_1 q_1$ and $n_2 = p_2 q_2 = p_1 q_2$ will have a common factor. We can therefore calculate the $\gcd$ of $n_1$ and $n_2$ to find $p_1$ and hence, $p_2$, $q_1$ and $q_2$. We can then decrypt the ciphertext.

```python
from Crypto.Util.number import inverse, long_to_bytes, GCD
p1 = GCD(n1, n2)
q1 = n1 // p1
phi = (p1 - 1)*(q1 - 1)
d = inverse(e, phi)
m = pow(c, d, n)
print(long_to_bytes(m)) # got the secret message!
```

# RSA Problems

RSA is pretty good, but there can be some issues which make it vulnerable in certain cases:

- If $n$ is easily factorisable:
  - If $n$ is small
  - If $p$ and $q$ are close (precisely, if $p - q < n^{\frac{1}{4}}$)
- If $e$ is small and $m^e < n$ then $m^e \bmod n$ is a no-op
- If $e$ is small and the same message is sent to multiple recipients with different moduli
- If a message is sent multiple times using different moduli, of which some have common factors
- If a message is sent with a common modulus, and different public exponents that are coprime
- and many more...

# Other CTF Crypto Problems

- One time pads with weak PRNGs
- Random classical ciphers
- Discrete log problem in DH key exchange
- Discrete log problem in Elliptic-curve cryptography (ECC)
- Random number theory/group theory/abstract algebra problems
- Decryption oracles
- and many more...

# Resources and Links

Readings and Challenges:

- Crypto 101 (book)
- An Overview of Cryptography - Gary C. Kessler (book)
- Cryptography: An Introduction - Nigel Smart (book)
- Crypton (readings + challenges)
- Dan Boneh's RSA Survey
- cryptopals (challenges)
- CTF Writeups

Tools:

- factordb
- Alpertron integer factorisation calculator
- SageMath