Cryptography and Security

Cryptography

What is cryptography?

- ***crypto*** = hidden, secret
- ***graph*** = to write

The practice and study of techniques for secure communication in the presence of third parties

Introduction

Oh, no! Our Rubber Duck was kidnapped by DevVolcano, another bootcamp. Is he safe?

Cryptography Characters

Commonly, in security field, our “characters” are the following.

“Alice” needs to send “Bob” a message.

plaintext is the original message.

ciphertext is the enciphered message.

Note: Common Characters

To make them easier to follow, most cryptographic protocols use the same cast of example characters. Some of these are:

- **Alice**, **Bob**, **Carol**, and **Dave**: the main actors, who want to send messages, vote, exchange cash, etc.
- **Eve**, who is evil and tries to steal messages.
- **Mallory**, who is malicious, and may disrupt messages, even if he isn’t trying to steal secrets. Mallory often plays a role known as the “man in the middle”.
- **Trent**, who is a trusted authority and often provides neutral information, like timestamps.
- **Walter**, who is a warden and can hold on to information securely.

Caesar Cipher

- This is one of the earlier ciphers known to have been used, by Julius Caesar, for military messages.
- Shift each letter by 3, Z wrapping back to A.
- So, for example, if you shift ***A*** by 2, it becomes ***C***. If you shift ***Y*** by 2, it becomes ***A***.

Shifting by 3:

```
RUBBERDUCKISSAFE + 3 →
UXEEHUGXFNLVVDIH
```

To decode, just shift by -3:

```
UXEEHUGXFNLVVDIH -3 →
RUBBERDUCKISSAFE
```

Trivial to break — you just have to know how it works

The encryption is always the same (by 3).

Better if we introduced a key — We can change the key to change the code.

Then we have some protection if someone learns our algorithm.

> *A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.*
>
> *—Dutch cryptographer Auguste Kerckhoffs*

Simple Substitution

Make a key of the alphabet, rearranged:

```
      abcdefghijklmnopqrstuvwxyz
key = zyxwvutsrqponmlkjihgfedcba
```

Note: Key Creation

In the real world, to make it easier to remember these keys, they were often made up of phrases, where you used each letter in order, and then followed with the rest of the alphabet. This one is made with the letters in the memorable phrase "obscenicorn was dancing happily with much joy", followed by the rest of the alphabet.)

Then, change each letter to key letter:

```
RUBBERDUCKISSAFE + key →
IFYYVIWFXPRHHZUV
```

To decode, reverse process:

```
IFYYVIWFXPRHHZUV + key →
RUBBERDUCKISSAFE
```

How many key possibilities are there?

- ~4,032,914,611,266,056,355,840,000,000 possibilities
- Even if a very fast computer attempted every possibility, it wouldn't finish in our lifetime

Note: Frequency Analysis

While it's impractical to solve this by trying all the combinations, these can be easily solved with "frequency analysis", by looking at which ciphertext letters are most common, and matching those to the most common letters in the plaintext language. Also, you can look for common words, bigraphs, trigraphs, etc.

This is a good example of how many cryptographic algorithms may be much easier to break than their key length suggests: we have a huge key length here, but can typically solve these very easily without trying anywhere number every possible key.

Note: Another Cipher

If you'd like to learn more, check out the Vigenère cipher– "The unbreakable cipher"

Brute Force

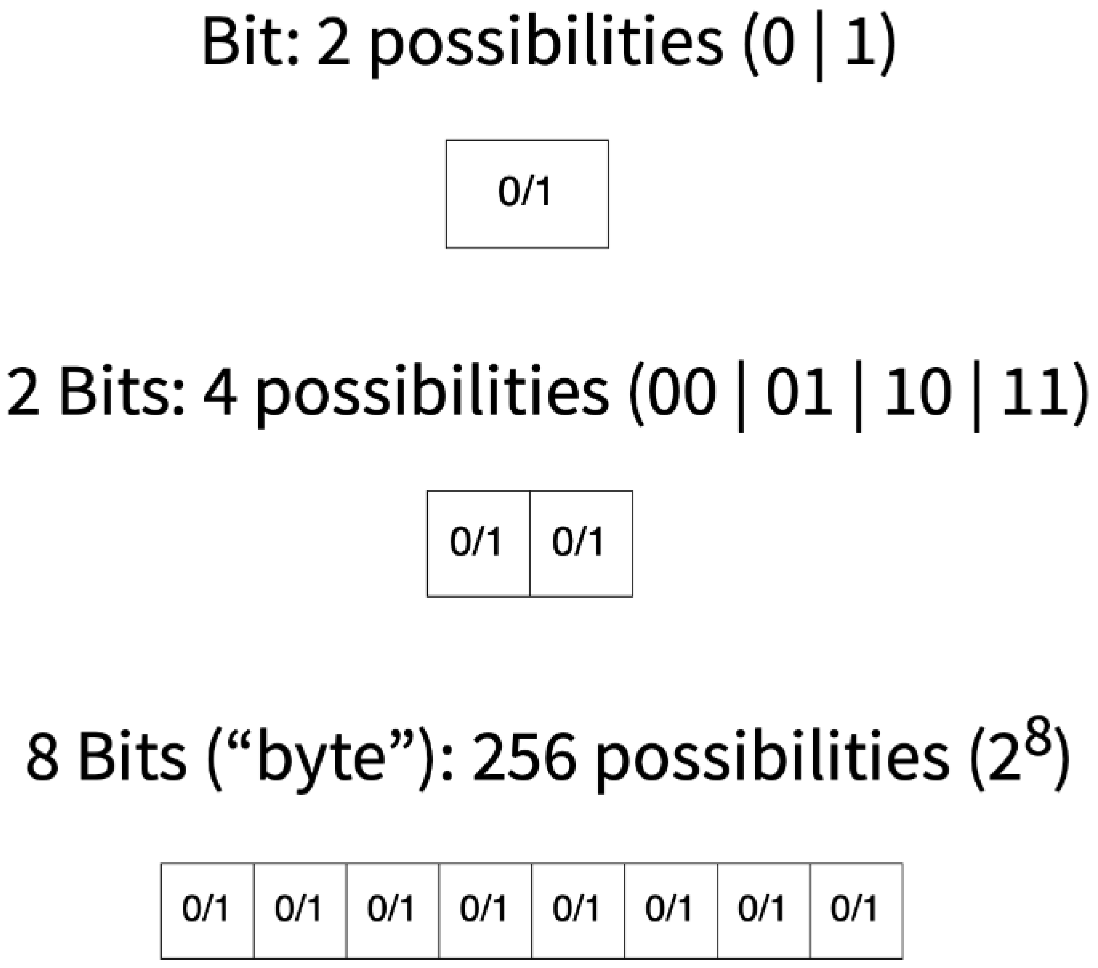
- In cybersecurity, **brute force** refers to a method of "breaking" a code by simply trying all possible combinations, usually with the help of a computer
- One mark of a good (safe) cryptographic algorithm is one that cannot be solved using brute force

Digital Algorithms

Modern digital algorithms are similar in idea to simple ciphers like ***Caesar***, but rely on complex math transformations instead of just “move forward by 3 letter in alphabet”.

- Encrypt bits, not letters
- Use more complex math transformations
  - **But principles are the same as classic cryptography**
  - Key length typically expresed in bits
    - 20-bit key length means 2^{20} possible keys
    - Each new bit makes brute force attack twice as hard

Quick Intro to Binary


Bit: 2 possibilities (0 | 1)
0/1
2 Bits: 4 possibilities (00 | 01 | 10 | 11)
0/1 0/1
8 Bits (“byte”): 256 possibilities (2^8)
0/1 0/1 0/1 0/1 0/1 0/1 0/1 0/1

Public Key Encryption

Note: A Major Discovery

Public Key encryption was discovered by numerous people — the first public disclosure was in the mid 1970s by a team of American and Israeli computer scientists, but since then, British Intelligence has revealed and proven that they had secretly made the same discovery years earlier. Whether other intelligence agencies made the discovery even earlier than that is not known.

Digital Algorithms

Important digital encryption algorithms:

DES
“Data Encryption Standard”. From the 1970s. Uses a 56-bit key; no longer safe from well-funded cryptanalysis.

Triple-DES
DES three times in a row. Provides ~118-bit key length. Slow but secure.

AES
“Advanced Encryption Standard”. Available in three key lengths (128-, 192-, 256-bit). Extremely secure.

Public Key Encryption

A type of **asymmetric encryption** (different key to encrypt than decrypt)

- public key *(tell everyone)*
- private key *(only owner knows)*

Things encrypted with public key only decrypted with private key

Things encrypted with private key only decrypted with public key

Sending Secret Message

You can send me a message without my private key.

```
public_key  = UKNOW
private_key = IKNOW
```

For now, let’s assume these two words are magically related, so that things encrypted with UKNOW are decrypted with IKNOW, and vice-versa.

You encrypt with public key:

```
RUBBERDUCKISSAFE + UKNOW →
AFDFBJDFBDGDFBSFSE
```

I decrypt with private key:

```
AFDFBJDFBDGDFBSFSE + IKNOW →
RUBBERDUCKISSAFE
```

How This Works: RSA

The RSA algorithm is a **trap door function**

- There are certain mathematical operations that are extremely *easy* to do in one direction, but extremely **difficult** to do backwards
  - Even with a very powerful computer!
- Imagine mixing two paints together
  - It’s very easy to mix two paints together to make a new color
  - It’s practically impossible to derive which two paints made the new color

Note: Factoring Semiprimes is a Trapdoor Function

> *Example: given the product of two primes, identify the primes.*

36,853 = ____ × ____ ?

> *It takes a modern computer over 1,000 times*

as long to find those prime numbers from 36,853 as it takes to multiply them to produce that number. In real life, the prime numbers used are typically dozens of digits long, and the difference in speed is in the quadrillions or more — making it very, very impractical to calculate, even with significant computing resources and lots of time.

Note: RSA Algorithm Pseudocode

- Three numbers are found, ***d***, ***e***, ***n***, which have trapdoor relationships
- Private key = ***(d, n)***
- Public key = ***(e, n)***
- It’s very hard to discover ***e*** from ***(d, n)***
- It’s very hard to discover ***d*** from ***(e, n)***

“Have a relationship” is hard to describe non-mathematically, but you can read about the RSA Algorithm.

You can study the algorithm here — but it requires a good understanding of collegiate math. Rest assured that almost all software developers use RSA without understanding how the math works.

If you’d like to see a simplified version of a public/private key system, see ***crypto-demo/kidrsa.py***.

A Python implementation of the full algorithm is in ***crypto-demo/rsa.py***.

There’s an IPython notebook to walk through the RSA algorithm in ***crypto-demo/rsa.ipynb***.

Basic Security Protocols

Send an Encrypted Message

- Asymmetric encryption is slow (~1,000× slower!)
- Often, you use symmetric and asymmetric together

First, make a random "session key" (example: XZJ)

Use this to encipher with symmetric algorithm:

```
RUBBERDUCKISSAFE + XZJ →
GJTXHJRAYRWSVWRQLO
```

Encrypt session key with public key:

```
XZJ + UKNOW →
FDFDFE
```

Send both ciphertext and encoded session key.

Recover the session key using private key:

```
FDFDFE - IKNOW →
XZJ
```

Decrypt message with session key:

```
GJTXHJRAYRWSVWRQLO - XZJ →
RUBBERDUCKISSAFE
```

The key never traveled in the clear!

Digital Signature

Only the private key enciphers things decipherable with public key.

Can use this to prove I wrote a message.

This is useful for "non-repudiation", the ability to have communication where one party can't claim they didn't actually write a message believed to come from them. For example, in banking, if a client withdraws cash, the bank wants to find ways to prove the client made the request, so that the client can't turn around and say that they never got the money, and that someone else made the request.

```
public_key  = UKNOW
private_key = IKNOW
```

I encrypt with private key:

```
RUBBERDUCKISSAFE + IKNOW →
BGFDVSFREGDGTGSASD
```

Send both plaintext and signature

Decrypt signature, confirm match:

```
BGFDVSFREGDGTGSASD + UKNOW →
RUBBERDUCKISSAFE
```

Matches plaintext, authentic!

Hashes

Imagine the first sentence of a novel:

“It was a queer, sultry summer, the summer they electrocuted the Rosenbergs, and I didn’t know what I was doing in New York.”

– Sylvia Plath, *The Bell Jar*

From this, we could get a “hash”:

```
hash = "iwaqsststetraidkwiwdiny"
```

Hash: stable “one-way” conversion of data to fixed-size result.

Can use hashes to ensure file hasn’t changed/been corrupted:

```
$ shasum -a 256 my-file.txt     # use "SHA256" hash
c70ef758da7167bdf1a2be7c4db14beab593eccb4f21042d9e1a663fdd23c641
```

There’s no way to turn that number back into full file …

… but any change to the file will result in a different hash.

Hashes for Signatures

- Remember, “signing” sends both plaintext + signature
  - You authenticate signature by making sure they match
- That means sending a message at least 2x as plaintext
  - And public-key encryption is slow
- Instead:
  - Make hash of message
  - Digitally sign hash
  - Send original message + signed hash

So, for example, let’s say we had a hash algorithm of “take every 3rd letter” (this is a terrible algorithm, but easy to understand for our example).

Our message of **RUBBERDUCKISSAFE** would hash into ***BLNOIA***. This would be encrypted with the private key — becoming, say, **AGHOEQ**. Both the real message and the signature are sent together:

```
RUBBERDUCKISSAFE/AGHOEQ
```

The recipient peels the two apart, hashes the original message, and decrypts the signature. They then compare the decrypted signature to the hash, and if they match, can trust that only the real sender could have produced that signature.

Web Security Protocols

HTTPS

HTTPS is a secure form of HTTP

- Inside all of the wrapping, it’s normal HTTP
- Wrapped around that is **TLS** (transport layer security)

Note: SSL

Different security layers have been used to provide HTTPS; originally, a scheme called “SSL” (secure sockets layer) was used. This was found to have vulnerabilities, so newer versions were introduced; these also had vulnerabilities. This was switched to TLS in most browsers around 2001 — but many people and books still talk about “SSL” to mean “the security layers used in HTTPS”, even though the ones actually used now are TLS.

You can learn about TLS but, even for security specialists, it’s a complex topic and very few developers need to know much about it.

HTTPS Protocol

- **Client**: “Hi, I want to use TLS. We’ll use RSA + AES.” *(it suggests both an asymmetric and symmetric method to use together)*
- **Server**: “I can do that. Here’s my ‘certificate’.” *(public key) (also: certificate ‘signed’ by CA; proving server is not imposter)*
- **Client**: “Here’s a random ‘session key’, encoded with that key.”
- **Server**: “Ok.” *(decodes session key using its private key)*
- **Client**: “Here’s my encrypted HTTP request …” *(encoded with session key)*
- **Server**: “Here’s my encrypted reply…” *(encoded with session key)*

The outside world can only see the server that the client is talking to — the URL requested, cookies, form info, and everything else is encrypted.

OAuth / APIs

OAuth

- When one website allows you to authenticate using your credentials from *a different* website
  - For example, Amazon, Google, Facebook, etc.
- “Rubberducky.com would like access to your friends, contacts, and birthday…”
- OAuth is the protocol for Rubberducky.com and Facebook to interact
- Rubberducky needs to be able to associate you with your Facebook account
- Facebook needs a way to securely provide access your info to Rubberducky

APIs

- When one codebase wants to interface with another codebase
- For example, Spotify has a Web-based API that other coders can use to get access to album, artist, and song information to use in their own apps
- The Spotify API needs a way to track and authenticate the requests that other developers make to Spotify
- Use hashes & public-key encryption for authenticating
- Sample OAuth authentication header:
  - your access token
  - timestamp
  - nonce *(random string, never re-used)*
  - hash of everything above, signed with private access key
- This proves:
  - When request was created *(can verify with timestamp)*
  - Prevents re-using of a request *(nonce would repeat)*
  - It was really made by you *(check signature of hash)*
  - Nothing changed/corrupted in transmission *(check hash)*

SSH

- **SSH** is a program (& protocol) to get a remote terminal on a server.
- SSH stands for secure shell

```
$ ssh rubberducky.com

Welcome to rubberducky.com
Last login: Sun Feb 14 04:55:32 2021

you@rubberducky $ # <-- shell prompt on server
```

- This can use passwords, but that’s less secure.
- More common: Put *public* key on server, and *private* key on laptop.
- SSH makes **handshake** using private key that server can tell was from you.

SSH Handshake

- Server keeps a list of allowed public keys in ***authorized_keys*** file
- User makes request to connect to server
- Server generates a “challenge” for user
  - Here, encrypt this message with your private key
- User’s computer encrypts message and send back
- Server checks to see if encrypted message can be decrypted with any public key in the ***authorized_keys*** file
  - If so, user gets access
  - if not, user is denied access

Security Vulnerabilities

OWASP Top 10

- OWASP is the Open Web Application Security Project®
- Nonprofit foundation that works to improve the security of software
- OWASP Top 10 is a standard awareness document
- Represents a broad consensus about the **most critical security risks** to web applications

#1: SQL Injection

```
let melonName = askUserForMelonName();

let sql = "SELECT price FROM melons WHERE name = " + melonName;
```

What if ***melonName*** was `Honeydew'; DROP TABLE melons; commit --` ?

```
SELECT price FROM melons WHERE name = 'Honeydew';
DROP TABLE melons; commit --'
```

- Our query looks innocent enough
- But if the user includes `';` this indicates to Postgres that the first query is finished.
- It is then followed by another, very malicious query: to delete all of the information in our table!

When an application is vulnerable SQL Injection:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly

How to prevent SQL Injection:

- Preventing injection requires keeping data separate from commands and queries.
- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).

#2 Broken Authentication

Credential stuffing is the malicious practice when a hacker has a list of usernames and passwords and uses automation to try the whole list until one allows access to a user's account.

It's sadly too common for sensitive user data like this to get into the wrong hands.

It's the responsibility of every website developer to ensure the proper protections are in place to prevent malicious automated attacks like this.

When an application is vulnerable to Broken Authentication

- Website permits automated attacks (trying to log in > 5 times in a very short amount of time)
- Website permits default, weak, or well-known passwords
- Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers", which cannot be made safe

How to prevent Broken Authentication

- Where possible, implement multi-factor authentication (prevents automated attacks)
- Implement password length, complexity and rotation policies
- Limit or increasingly delay failed login attempts

#3: Broken Access Control

- There are certain things on most applications that can only be done by logged-in users, admins, and other specific types of users
- While it's very easy to create features that are **intended** for a certain user, it's harder to ensure that no other types of users are able to intentionally gain access to those website features too
- Example: A website uses the following URL pattern for User Profile pages

  - `https://rubberducky.com/users/45`
- Here `45` is the user's ID
- What if someone edits the ID in the URL and replaces it with another user's ID?
  - Proper checks need to be in place to prevent this

When an application is vulnerable to Broken Access Control

- Allowing the primary key to be changed to another's users record, permitting viewing or editing someone else's account
- Elevation of privilege. Acting as a user without being logged in, or acting as an admin when logged in as a user

How to prevent Broken Access Control

- With the exception of public resources, deny access **by default**
- Implement access control mechanisms once and re-use them throughout the application

#4-10

- There are 7 more OWASP Vulnerabilities to learn about
- If you are excited about security, read more on the OWASP website
- https://owasp.org/

The End