# Security and Cryptography

Last year's [security and privacy lecture](#) focused on how you can be more secure as a computer *user*. This year, we will focus on security and cryptography concepts that are relevant in understanding tools covered earlier in this class, such as the use of hash functions in Git or key derivation functions and symmetric/asymmetric cryptosystems in SSH.

This lecture is not a substitute for a more rigorous and complete course on computer systems security ([6.858 (https://css.csail.mit.edu/6.858/)](https://css.csail.mit.edu/6.858/)) or cryptography ([6.857 (https://courses.csail.mit.edu/6.857/)](https://courses.csail.mit.edu/6.857/) and 6.875). Don't do security work without formal training in security. Unless you're an expert, don't [roll your own crypto (https://www.schneier.com/blog/archives/2015/05/amateurs_produc.html)](https://www.schneier.com/blog/archives/2015/05/amateurs_produc.html). The same principle applies to systems security.

This lecture has a very informal (but we think practical) treatment of basic cryptography concepts. This lecture won't be enough to teach you how to *design* secure systems or cryptographic protocols, but we hope it will be enough to give you a general understanding of the programs and protocols you already use.

Entropy

[Entropy (https://en.wikipedia.org/wiki/Entropy_(information_theory))](https://en.wikipedia.org/wiki/Entropy_(information_theory)) is a measure of randomness. This is useful, for example, when determining the strength of a password.
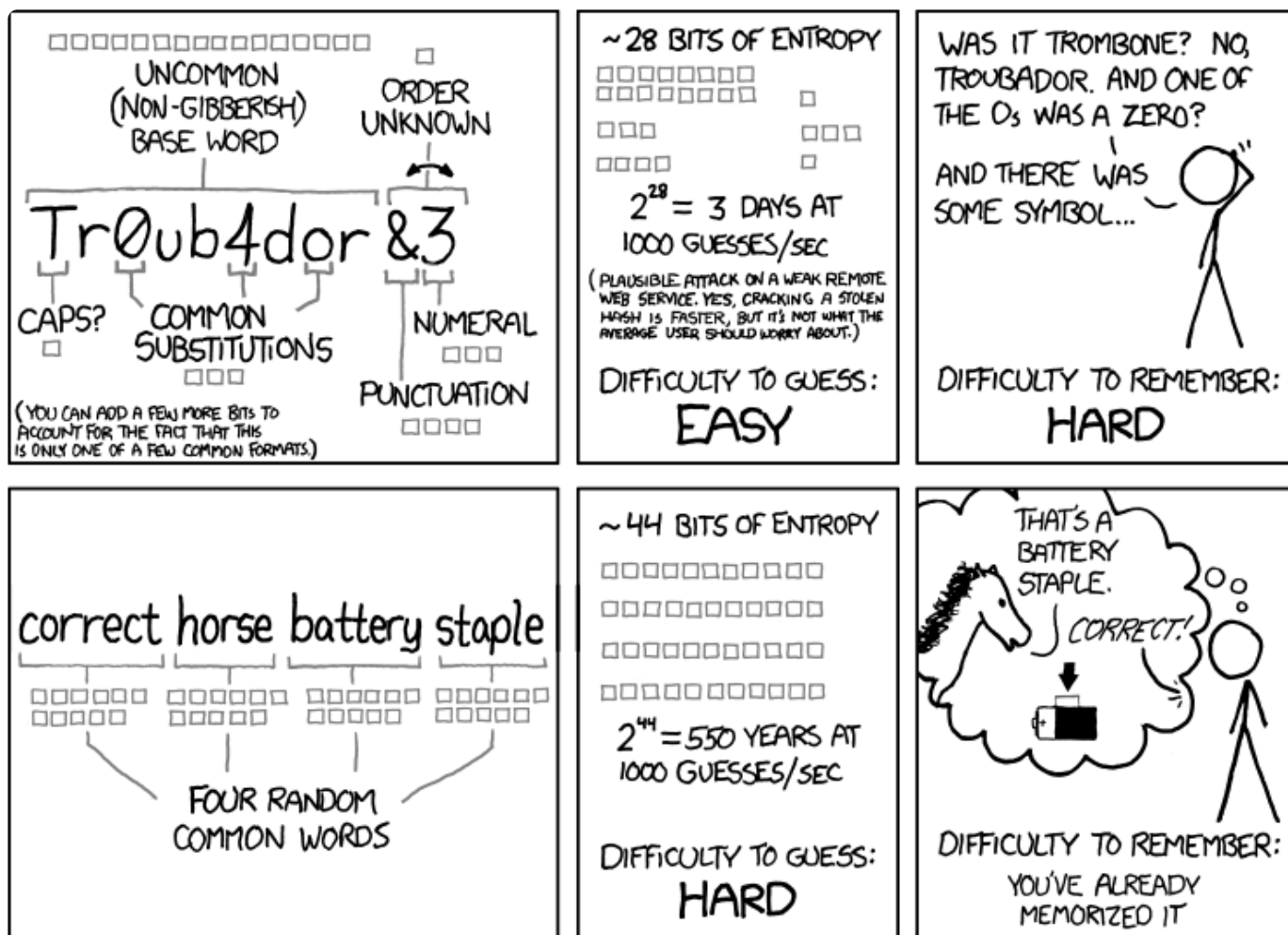
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

As the above XKCD comic (https://xkcd.com/936/) illustrates, a password like "correcthorsebatterystaple" is more secure than one like "Tr0ub4dor&3". But how do you quantify something like this?

Entropy is measured in *bits*, and when selecting uniformly at random from a set of possible outcomes, the entropy is equal to `log_2(# of possibilities)`. A fair coin flip gives 1 bit of entropy. A dice roll (of a 6-sided die) has ~2.58 bits of entropy.

You should consider that the attacker knows the *model* of the password, but not the randomness (e.g. from dice rolls (https://en.wikipedia.org/wiki/Diceware)) used to select a particular password.

How many bits of entropy is enough? It depends on your threat model. For online guessing, as the XKCD comic points out, ~40 bits of entropy is pretty good. To be resistant to offline guessing, a stronger password would be necessary (e.g. 80 bits, or more).

Hash functions

A cryptographic hash function (https://en.wikipedia.org/wiki/Cryptographic_hash_function) maps data of arbitrary size to a fixed size, and has some special properties. A rough specification of a

hash function is as follows:

```
hash(value: array<byte>) -> vector<byte, N>  (for some fixed N)
```

An example of a hash function is SHA1 (https://en.wikipedia.org/wiki/SHA-1), which is used in Git. It maps arbitrary-sized inputs to 160-bit outputs (which can be represented as 40 hexadecimal characters). We can try out the SHA1 hash on an input using the `sha1sum` command:

```
$ printf 'hello' | sha1sum
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
$ printf 'hello' | sha1sum
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
$ printf 'Hello' | sha1sum
f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0
```

At a high level, a hash function can be thought of as a hard-to-invert random-looking (but deterministic) function (and this is the ideal model of a hash function (https://en.wikipedia.org/wiki/Random_oracle)). A hash function has the following properties:

– Deterministic: the same input always generates the same output.
– Non-invertible: it is hard to find an input `m` such that `hash(m) = h` for some desired output `h`.
– Target collision resistant: given an input `m_1`, it's hard to find a different input `m_2` such that `hash(m_1) = hash(m_2)`.
– Collision resistant: it's hard to find two inputs `m_1` and `m_2` such that `hash(m_1) = hash(m_2)` (note that this is a strictly stronger property than target collision resistance).

Note: while it may work for certain purposes, SHA-1 is no longer (https://shattered.io/) considered a strong cryptographic hash function. You might find this table of lifetimes of cryptographic hash functions (https://valerieaurora.org/hash.html) interesting. However, note that recommending specific hash functions is beyond the scope of this lecture. If you are doing work where this matters, you need formal training in security/cryptography.

Applications

– Git, for content-addressed storage. The idea of a hash function (https://en.wikipedia.org/wiki/Hash_function) is a more general concept (there are non-cryptographic hash functions). Why does Git use a cryptographic hash function?
– A short summary of the contents of a file. Software can often be downloaded from (potentially less trustworthy) mirrors, e.g. Linux ISOs, and it would be nice to not have to trust them. The official sites usually post hashes alongside the download links (that point to third-party mirrors), so that the hash can be checked after downloading a file.

- Commitment schemes [(https://en.wikipedia.org/wiki/Commitment_scheme)](https://en.wikipedia.org/wiki/Commitment_scheme). Suppose you want to commit to a particular value, but reveal the value itself later. For example, I want to do a fair coin toss "in my head", without a trusted shared coin that two parties can see. I could choose a value `r = random()`, and then share `h = sha256(r)`. Then, you could call heads or tails (we'll agree that even `r` means heads, and odd `r` means tails). After you call, I can reveal my value `r`, and you can confirm that I haven't cheated by checking `sha256(r)` matches the hash I shared earlier.

Key derivation functions

A related concept to cryptographic hashes, key derivation functions [(https://en.wikipedia.org/wiki/Key_derivation_function)](https://en.wikipedia.org/wiki/Key_derivation_function) (KDFs) are used for a number of applications, including producing fixed-length output for use as keys in other cryptographic algorithms. Usually, KDFs are deliberately slow, in order to slow down offline brute-force attacks.

Applications

- Producing keys from passphrases for use in other cryptographic algorithms (e.g. symmetric cryptography, see below).
- Storing login credentials. Storing plaintext passwords is bad; the right approach is to generate and store a random salt [(https://en.wikipedia.org/wiki/Salt_(cryptography))](https://en.wikipedia.org/wiki/Salt_(cryptography)) `salt = random()` for each user, store `KDF(password + salt)`, and verify login attempts by re-computing the KDF given the entered password and the stored salt.

Symmetric cryptography

Hiding message contents is probably the first concept you think about when you think about cryptography. Symmetric cryptography accomplishes this with the following set of functionality:

```
keygen() -> key   (this function is randomized)

encrypt(plaintext: array<byte>, key) -> array<byte>   (the ciphertext)
decrypt(ciphertext: array<byte>, key) -> array<byte>   (the plaintext)
```

The encrypt function has the property that given the output (ciphertext), it's hard to determine the input (plaintext) without the key. The decrypt function has the obvious correctness property, that `decrypt(encrypt(m, k), k) = m`.

An example of a symmetric cryptosystem in wide use today is AES [(https://en.wikipedia.org/wiki/Advanced_Encryption_Standard)](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard).

Applications

- Encrypting files for storage in an untrusted cloud service. This can be combined with KDFs, so you can encrypt a file with a passphrase. Generate `key = KDF(passphrase)`, and then

```
store encrypt(file, key).
```

Asymmetric cryptography

The term "asymmetric" refers to there being two keys, with two different roles. A private key, as its name implies, is meant to be kept private, while the public key can be publicly shared and it won't affect security (unlike sharing the key in a symmetric cryptosystem). Asymmetric cryptosystems provide the following set of functionality, to encrypt/decrypt and to sign/verify:

```
keygen() -> (public key, private key)  (this function is randomized)

encrypt(plaintext: array<byte>, public key) -> array<byte>  (the cipherte
decrypt(ciphertext: array<byte>, private key) -> array<byte>  (the plaint

sign(message: array<byte>, private key) -> array<byte>  (the signature)
verify(message: array<byte>, signature: array<byte>, public key) -> bool
```

The encrypt/decrypt functions have properties similar to their analogs from symmetric cryptosystems. A message can be encrypted using the *public* key. Given the output (ciphertext), it's hard to determine the input (plaintext) without the *private* key. The decrypt function has the obvious correctness property, that `decrypt(encrypt(m, public key), private key) = m`.

Symmetric and asymmetric encryption can be compared to physical locks. A symmetric cryptosystem is like a door lock: anyone with the key can lock and unlock it. Asymmetric encryption is like a padlock with a key. You could give the unlocked lock to someone (the public key), they could put a message in a box and then put the lock on, and after that, only you could open the lock because you kept the key (the private key).

The sign/verify functions have the same properties that you would hope physical signatures would have, in that it's hard to forge a signature. No matter the message, without the *private* key, it's hard to produce a signature such that `verify(message, signature, public key)` returns true. And of course, the verify function has the obvious correctness property that `verify(message, sign(message, private key), public key) = true`.

Applications

- [PGP email encryption](https://en.wikipedia.org/wiki/Pretty_Good_Privacy). People can have their public keys posted online (e.g. in a PGP keyserver, or on [Keybase](https://keybase.io/)). Anyone can send them encrypted email.
- Private messaging. Apps like [Signal](https://signal.org/) and [Keybase](https://keybase.io/) use asymmetric keys to establish private communication channels.
- Signing software. Git can have GPG-signed commits and tags. With a posted public key, anyone can verify the authenticity of downloaded software.

Key distribution

Asymmetric-key cryptography is wonderful, but it has a big challenge of distributing public keys / mapping public keys to real-world identities. There are many solutions to this problem. Signal has one simple solution: trust on first use, and support out-of-band public key exchange (you verify your friends' "safety numbers" in person). PGP has a different solution, which is [web of trust](https://en.wikipedia.org/wiki/Web_of_trust) . Keybase has yet another solution of [social proof](https://keybase.io/blog/chat-apps-softer-than-tofu) (along with other neat ideas). Each model has its merits; we (the instructors) like Keybase's model.

Case studies

Password managers

This is an essential tool that everyone should try to use (e.g. [KeePassXC](https://keepassxc.org/) , [pass](https://www.passwordstore.org/) , and [1Password](https://1password.com)). Password managers make it convenient to use unique, randomly generated high-entropy passwords for all your logins, and they save all your passwords in one place, encrypted with a symmetric cipher with a key produced from a passphrase using a KDF.

Using a password manager lets you avoid password reuse (so you're less impacted when websites get compromised), use high-entropy passwords (so you're less likely to get compromised), and only need to remember a single high-entropy password.

Two-factor authentication

[Two-factor authentication](https://en.wikipedia.org/wiki/Multi-factor_authentication) (2FA) requires you to use a passphrase ("something you know") along with a 2FA authenticator (like a [YubiKey](https://www.yubico.com/) , "something you have") in order to protect against stolen passwords and [phishing](https://en.wikipedia.org/wiki/Phishing) attacks.

Full disk encryption

Keeping your laptop's entire disk encrypted is an easy way to protect your data in the case that your laptop is stolen. You can use [cryptsetup + LUKS](https://wiki.archlinux.org/index.php/Dm-crypt/Encrypting_a_non-root_file_system) on Linux, [BitLocker](https://fossbytes.com/enable-full-disk-encryption-windows-10/) on Windows, or [FileVault](https://support.apple.com/en-us/HT204837) on macOS. This encrypts the entire disk with a symmetric cipher, with a key protected by a passphrase.

Private messaging

Use [Signal](https://signal.org/) or [Keybase](https://keybase.io/) . End-to-end security is bootstrapped from asymmetric-key encryption. Obtaining your contacts' public keys is the critical

step here. If you want good security, you need to authenticate public keys out-of-band (with Signal or Keybase), or trust social proofs (with Keybase).

SSH

We've covered the use of SSH and SSH keys in an earlier lecture. Let's look at the cryptography aspects of this.

When you run `ssh-keygen`, it generates an asymmetric keypair, `public_key`, `private_key`. This is generated randomly, using entropy provided by the operating system (collected from hardware events, etc.). The public key is stored as-is (it's public, so keeping it a secret is not important), but at rest, the private key should be encrypted on disk. The `ssh-keygen` program prompts the user for a passphrase, and this is fed through a key derivation function to produce a key, which is then used to encrypt the private key with a symmetric cipher.

In use, once the server knows the client's public key (stored in the `.ssh/authorized_keys` file), a connecting client can prove its identity using asymmetric signatures. This is done through challenge-response (https://en.wikipedia.org/wiki/Challenge%E2%80%93response_authentication). At a high level, the server picks a random number and sends it to the client. The client then signs this message and sends the signature back to the server, which checks the signature against the public key on record. This effectively proves that the client is in possession of the private key corresponding to the public key that's in the server's `.ssh/authorized_keys` file, so the server can allow the client to log in.

Resources

- Last year's notes: from when this lecture was more focused on security and privacy as a computer user
- Cryptographic Right Answers (https://latacora.micro.blog/2018/04/03/cryptographic-right-answers.html): answers "what crypto should I use for X?" for many common X.

Exercises

1. **Entropy.**
   1. Suppose a password is chosen as a concatenation of four lower-case dictionary words, where each word is selected uniformly at random from a dictionary of size 100,000. An example of such a password is `correcthorsebatterystaple`. How many bits of entropy does this have?
   2. Consider an alternative scheme where a password is chosen as a sequence of 8 random alphanumeric characters (including both lower-case and upper-case letters). An example is `rg8Ql34g`. How many bits of entropy does this have?
   3. Which is the stronger password?
   4. Suppose an attacker can try guessing 10,000 passwords per second. On average, how long will it take to break each of the passwords?

2. **Cryptographic hash functions.** Download a Debian image from a mirror (https://www.debian.org/CD/http-ftp/) (e.g. from this Argentinean mirror (http://debian.xfree.com.ar/debian-cd/current/amd64/iso-cd/)). Cross-check the hash (e.g. using the `sha256sum` command) with the hash retrieved from the official Debian site (e.g. this file (https://cdimage.debian.org/debian-cd/current/amd64/iso-cd/SHA256SUMS) hosted at `debian.org`, if you've downloaded the linked file from the Argentinean mirror).

3. **Symmetric cryptography.** Encrypt a file with AES encryption, using OpenSSL (https://www.openssl.org/): `openssl aes-256-cbc -salt -in {input filename} -out {output filename}`. Look at the contents using `cat` or `hexdump`. Decrypt it with `openssl aes-256-cbc -d -in {input filename} -out {output filename}` and confirm that the contents match the original using `cmp`.

4. **Asymmetric cryptography.**

   1. Set up SSH keys (https://www.digitalocean.com/community/tutorials/how-to-set-up-ssh-keys--2) on a computer you have access to (not Athena, because Kerberos interacts weirdly with SSH keys). Make sure your private key is encrypted with a passphrase, so it is protected at rest.

   2. Set up GPG (https://www.digitalocean.com/community/tutorials/how-to-use-gpg-to-encrypt-and-sign-messages)

   3. Send Anish an encrypted email (public key (https://keybase.io/anish)).

   4. Sign a Git commit with `git commit -S` or create a signed Git tag with `git tag -s`. Verify the signature on the commit with `git show --show-signature` or on the tag with `git tag -v`.