Lecture 09: Security and Cryptography

Entropy

Entropy is a measure of randomness. It 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) used to select a particular password.

Hash functions

A 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, 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
```

A hash function can be thought of as a **hard-to-invert random-looking** (but *deterministic*) function. 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).

SHA-1 is **no longer** considered a strong cryptographic hash function. You might find this table of lifetimes of cryptographic hash functions interesting.

Applications

- Git, for content-addressed storage.
- 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. 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

Key derivation functions (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.

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 `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

Symmetric cryptography accomplishes hiding message contents 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.

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).

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

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

sign(message: array<byte>, private key) -> array<byte>  (the signature)
verify(message: array<byte>, signature: array<byte>, public key) -> bool  (whether or not the signature is valid)
```

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).

he 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.

Applications

- PGP email encryption. People can have their public keys posted online (e.g. in a PGP keyserver, or on Keybase). Anyone can send them encrypted email.
- Private messaging. Apps like Signal and Keybase 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. Keybase has yet another solution of social proof (along with other neat ideas). Each model has its merits; we (the instructors) like Keybase's model.

Case studies

Password managers

Password managers let you use unique, randomly generated high-entropy passwords for all your websites, and they save all your passwords in one place, encrypted with a symmetric cipher with a key produced from a passphrase using a KDF.

Two-factor authentication

Two-factor authentication (2FA) requires you to use a passphrase ("something you know") along with a 2FA authenticator (like a YubiKey, "something you have") in order to protect against stolen passwords and 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 on Linux, BitLocker on Windows, or FileVault on macOS. This encrypts the entire disk with a symmetric cipher, with a key protected by a passphrase.

Private messaging

Use Signal or Keybase. 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

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. 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.

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.

Others

Cryptographic Right Answers

We're less interested in empowering developers and a lot more pessimistic about the prospects of getting this stuff right. There are, in the literature and in the most sophisticated modern systems, "better" answers for many of these items. If you're building for low-footprint embedded systems, you can use STROBE and a sound, modern, authenticated encryption stack entirely out of a single SHA-3-like sponge

https://latacora.micro.blog/2018/04/03/cryptographic-right-answers.html

What cripto should I use for X

When to use dash (-) in yaml?

One thing that's always confuses me is when to use dash in a yaml file. The way I understand it is that a list needs to start with a dash, and a dictionary (key value pair) doesn't. Take following playbook for example, I don't understand why debug needs to have

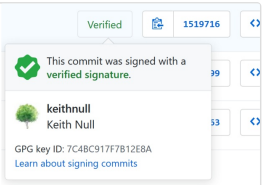
https://www.reddit.com/r/ansible/comments/5jhff3/when_to_use_dash_in_yaml/

Hyphens indicate list items

在Github上使用GPG的全过程

原发布时间：2019-08-04原发布地址：在Github上使用GPG的全过程 其实在很早之前 Github 就已经

https://zhuanlan.zhihu.com/p/76861431



Adding a new GPG key to your GitHub account

To configure your GitHub account to use your new (or existing) GPG key, you'll also need to add it to your GitHub account. Before adding a new GPG key to your GitHub account, you

https://help.github.com/en/github/authenticating-to-github/adding-a-new-gpg-key-to-your-github-account