Cryptography Applications

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

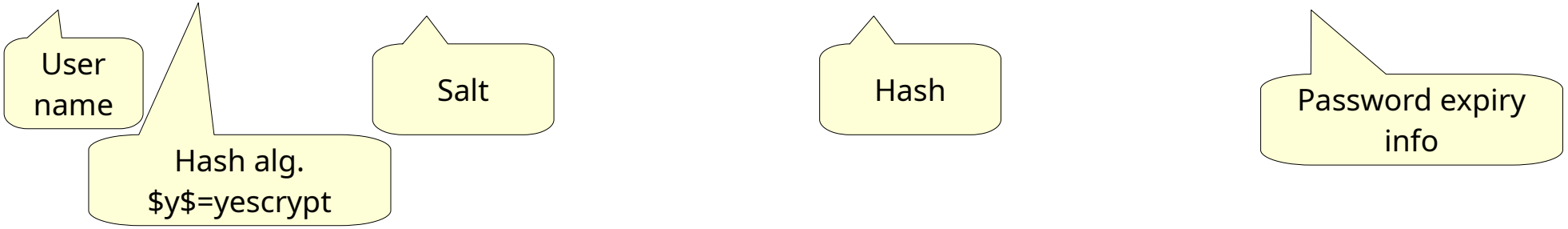
- How can we safely store passwords?
- How do we verify a document is authentic?
- How can we trust websites?

Storing Passwords

Storing Passwords

- *Password verification systems must not store plain text passwords*
  - Even storing encrypted passwords is unsafe!
  - Best practice: passwords are hashed, store only the hashes
  - If an attacker gets a copy of the password file, the hash cannot be used to log in, only to *check* a password
  - System checks: **hash(user-input) = stored hash?**
- Linux stores password hashes in /etc/shadow (accessible by root):

```
pwdis_1234:$y$j9T$LirzYIOxlHbbu6Wi/9zCl.$I9AZk5jAaY0uRPQfPnTEu.x5UeMVROrhP.i9gI96DD7:20159:0:99999:7:::
pwdis_1235:$y$j9T$Rnadt7C63/7Bl1s/3Gx8v0$fvKI5ljHrj2hhnIhb6SkvjPvDQb7Awb3wUKU5YNmKK.:20159:0:99999:7:::
pwdis_abcd:$y$j9T$/TAaIyA61/lpbWM0vB5wA1$jofzhQHkwXjtpqfZ6cXIEwepD1L1V75gNriCM5C3pS7:20159:0:99999:7:::
```



Rainbow Table Attack

- *Rainbow table attack*
  - Attacker pre-computes the hash of common passwords
  - Can then quickly search password file of hashes for known passwords
- *Defence: salt the password*
  - **Salt:** Just a random number or string
  - Store two values:
    1) salt
    2) hash(concat(user-password, salt))
  - Verify password:
    - hash(concat(user-input, salt)) = stored-hash
- Attacker cannot reasonably precompute hash of all possible passwords along with all possible salts
  - The salt is not a secret! It's just there to force the attacker to do more work
  - For this to be effective, *every salt should be different*

Verifying Documents

Secure Digest

- *Secure digest "summary" of document*
  - Often used to verify a downloaded file is not corrupted
  - A secure digest is a summary of a message: a *fixed-length* string that characterizes an *arbitrary-length* message
  - Typically produced by a cryptographic hash function, e.g. SHA-256
- *Example*

```
$ sha256sum ./README.md
e293cdc4f5c4686772fba8159be9e9636654fed7ce72bfd2e75add8a6833c5ab  ./README.md
```

Digital Signature

- *Digital signatures combine public key crypto and hashing*
  - Goal: Verify a message (or document) is an *unaltered copy* of the one produced by the signer
  - The message can be public; we just want to **prove who sent it** and that it's **unaltered**
- *Two parties: signer and verifier*
  - The **signer:**
    - *Sends* a message
    - Wants to prove they sent the message
  - The **verifier:**
    - *Receives* message
    - Wants to verify the message was sent by the signer and is unaltered

Signer

- *The signer…*
  - Writes a document: m
  - Computes a message digest: $h(m)$ (e.g., using SHA-256)
    - Not good enough yet: Adversary could write document z, compute $h(z)$, and plant both
  - Encrypts the digest with their *private* key: $\text{enc}(h(m))$
    - We're using asymmetric AKA public key cryptography! E.g., RSA
    - This is called signing
    - Only the signer has the private key, so only the signer can encode with it
  - Sends the message together with the signature: $<m, \text{enc}(h(m))>$

Verifier

- *The verifier…*
  - Receives the message and the signature: <m, $\text{enc}(h(m))$>
  - Decrypts the signature with the signer's public key:
    - $\text{dec}(\text{enc}(h(m))) = h(m)$
  - Computes a message digest: $h(m)$, call it h'
  - Verifies that $h(m) = h'$
- *If this verification works, then the message is authentic*
  - If the attacker modifies the message, the computed h' will change
    - Technically speaking, there must be other messages that produce the same hash, but the strength properties of the hash tell us *the attacker has no way to find them*
    - This is why we care about the quality of the hash function: “weak” hash functions like CRC or MD5 make it easy to find these other messages
  - Without the private key, any signature an attacker adds to the message will decrypt to some random nonsense, and it won’t match the new h'

Trusting Unknown Companies

Digital Certificate

- ***Digital certificates*** *use digital signatures*
- *Scenario*
  - Imagine sending password to website (e.g., Instagram)
  - You encrypt your password with Instagram's public key
  - Only Instagram can decrypt the message, so password is safe
- *Questions*
  - How do you get the public key of Instagram?
  - Suppose:
    - Instagram sends you their public key when you first visit
    - A rogue website impersonating Instagram could send you a wrong key!
    - **How do you trust the public key really belongs to Instagram?**
      - (but put a pin in this idea, we will revisit it)

Secure Browsing

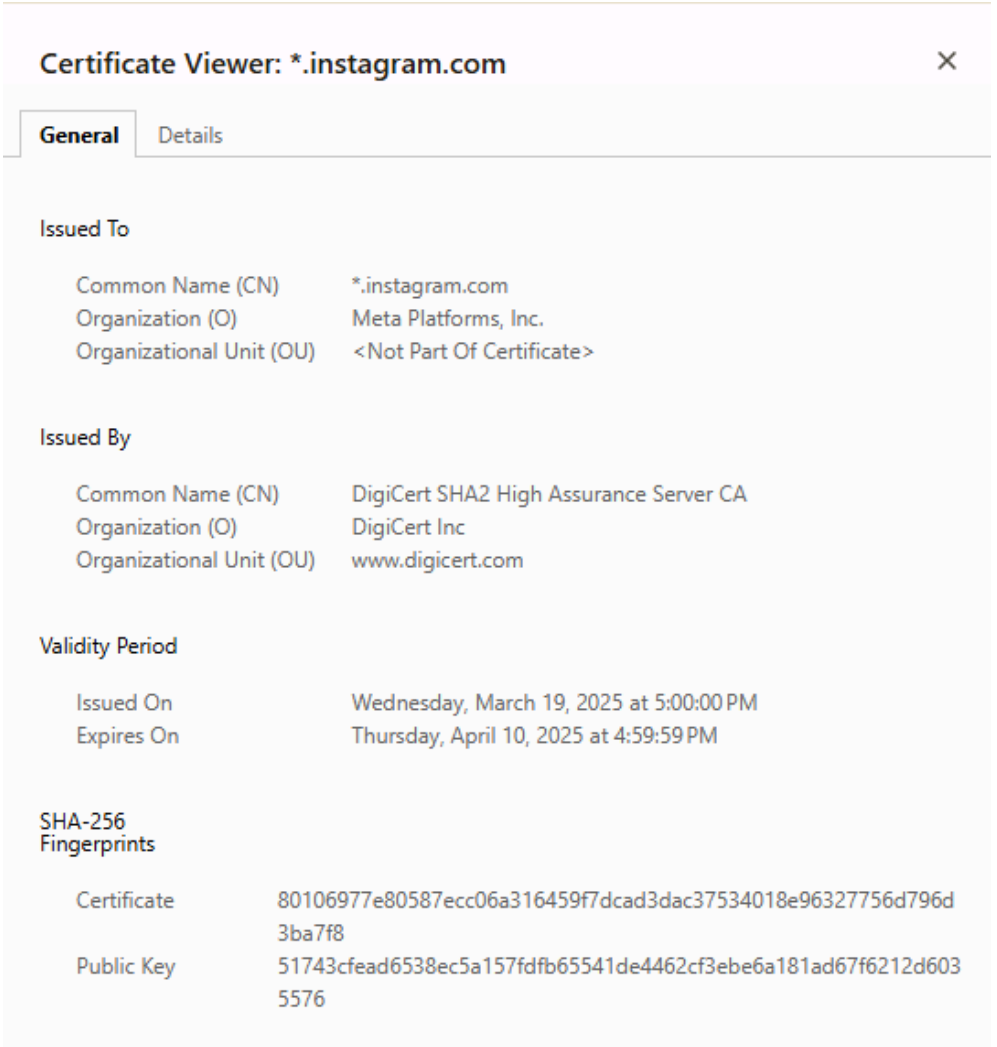
- ***HTTP*** *has no encryption*
- ***HTTPS*** *uses encryption*
  - Instagram sends you its public key in a **digital certificate**
  - **Digital certificate:** proves that the public key indeed belongs to Instagram
    - Your OS (and/or browser) verifies the authenticity of the digital certificate
    - The system has some built-in *trusted signing authorities*
      - Called “trusted/root certificates” or “trusted CAs”
      - Detail: the certificate Instagram sends may contain multiple certificates, each one signed by another, going back to some root CA
      - Detail: each certificate is only valid for a period of time (remember the *window of validity*), so *your system clock must be set correctly to validate certificates!*
      - Detail: there is also a revocation list for known compromised certificates
  - Your browser can use Instagram's public key to encrypt messages to Instagram
    - Only Instagram can decrypt messages encrypted with their public key

Digital Cert Operation

- How digital certificates work
  - A digital certificate is signed by a **digital certificate authority**
    - E.g., VeriSign, DigiCert, Let’s Encrypt…
  - OS vendor ships OS with public keys for some trusted digital certificate authorities like DigiCert
    - This gives us a starting point: the **root of trust**

Digital Certificate Example

- *Instagram uses DigiCert*
  - Instagram goes to DigiCert, gives its public key, and requests a digital certificate
- *DigiCert creates a digital certificate*
  - It says "this public key belongs to Instagram"
  - DigiCert signs it with DigiCert's own private key



Digital Certificate Example (cont'd)

- *Instagram Digital Certificate*
  - When browser connects to Instagram, Instagram sends the digital certificate
    - Browser uses its trusted public key for DigiCert to verify the digital certificate from Instagram
  - If your OS is not compromised, this whole process is secure based on the first level of trust
    - If the OS is compromised, there is no 1st level of trust and this whole process is not secure
    - "Compromised" in this scenario means, an attacker has access to the private key for one of the trusted certificates, and can sign anything they want!
      - Bad: an attacker adds their own CA certificate as a trusted CA on your OS
      - Worse: an attacker gets signing keys from the CA
- *Encryption Use*
  - Public key is slow and generates lots of data; can we make this more efficient?
  - Yes: use public key encryption to *initiate* a session
    - Exchange a *random, session-specific* secret key
  - Rest of communication encrypted with secret key encryption

Chain of Trust

- *Digital certificates rely on the **chain of trust***
  - To *trust the public key* sent by Instagram, we need to *trust DigiCert's signature*
  - To *trust DigiCert's signature,* we need to *trust DigiCert's public key*
  - To *trust DigiCert's public key* (shipped with OS), we need to *trust that our OS is not compromised*
- **Chain of trust** relies on the **root of trust** being trustworthy
  - Our root of trust is the OS

Chain of Trust (cont'd)

- *What about "end-to-end" encryption? (E.g. Instagram/Signal/Telegram/SSH…)*
  - You don't go to DigiCert to get a certificate to prove your identity
  - **So how does anyone know they're really talking to you, and not someone impersonating you?**
- *Service providers may distribute public keys (vouch for you)*
  - If you have to give a phone number or email, there may be a server that checks that you can read messages sent there
  - **Everyone is trusting the service provider's server!**
  - **"End-to-end" is misleading if you can't be sure the application will warn you about keys changing, or other signs of MITM**

Chain of Trust (cont'd)

- *Some systems allow you to directly validate identity*
  - e.g. Signal can display your public key as a QR code, and read it directly
  - **If you can verify the source code for the app, this is the strongest validation possible**
  - GPG (secure email) allows you to build a "web of trust" where you directly validate people you meet in person, and trust them to validate people you haven't met
- *SSH offers a "trust on first access" policy*
  - The first time accessing a new system, their public key is added to `~/.ssh/known_hosts`
  - **If SSH tells you that a key has changed, *pay attention!* This is extremely suspicious**
  - Not *really* secure, but it does help: to execute a MITM attack, the attacker has to be there the first time you connect, and also *every time thereafter,* in order to avoid being discovered
  - You can add keys to `~/.ssh/known_hosts` in advance – very secure, very annoying
    - GitHub publishes their public SSH key via HTTPS, note the chain of trust!

Optional Activity - SSH

- Spin up new container:
  ```
  $ docker run -it ghcr.io/sfu-cmpt-201/base
  ```
- Generate public/private key with ed25519 & passphrase
  ```
  $ ssh-keygen -t ed25519 -C "your email address"
  ```
  - Look at files in `~./ssh`
- SSH SFU
  ```
  $ ssh <yourID>@csil-cpu01.csil.sfu.ca -p24
  ```
  - Asks user name and password; use VPN if off campus
- SSH Keys
  - SSH SFU; manually add pub key to end of `~/.ssh/authorized_keys`
  - Log-out, log-in (asks passphrase)
- SSH Agent (Handy! Avoids repeating passphrase; stores keys in memory; can pass through intermediate connections A->B->C->etc. so *your* keys stay on *your* machine)
  ```
  $ eval ssh-agent
  $ ssh-add
  $ kill $SSH_AGENT_PID
  ```

Hash Collisions

Birthday Match = Hash Collision

- *Birthday Match*
  - In a class of 30 people, probability of two students having the same birthday is ~70%!
  - https://en.wikipedia.org/wiki/Birthday_attack
- *Hash Collisions*
  - Given enough messages, we can find a hash collision between two messages
  - Can we show that a hash function does not achieve strong collision resistance?
- *(Recall) Strong collision resistance*
  - It should be difficult to find two messages x and x' where $h(x) = h(x')$
  - I.e., given a hash function, it should be difficult to find two values that produce the same hash

Birthday Attack

- *Attacker uses a contract the customer is expected to sign*
  - (say, agreement to buy company for $100,000)
- Attacker then:
  - Creates benign altered copies of the contract (adding a space, adding commas, adding typos, …)
  - Creates malicious altered copies (sale price $100,000,000)
  - *Goal: find a malicious contract with same hash as benign contract*
  - Customer given benign copy to sign using their private key and hash of document
  - Attacker then substitutes benign contract with malicious one
  - **Since the contracts have same hash, attacker can claim customer signed malicious contract using their private key!**

Demo - Hash Collision

- Demo: Collision in Crypto Hash Functions
  - MD5 was a widely used crypto hash function but was found to be insecure by 2005
  - No longer in use
- Get images and Compare Hashes

```
$ wget https://s3-eu-west-1.amazonaws.com/md5collisions/ship.jpg
$ wget https://s3-eu-west-1.amazonaws.com/md5collisions/plane.jpg
$ openssl dgst -md5 ship.jpg
$ openssl dgst -md5 plane.jpg
```

  - Algorithm exists to manipulate a pair of images into having the same MD5 hash
- SHA-256 is not yet known to be insecure

```
$ openssl dgst -sha256 ship.jpg
$ openssl dgst -sha256 plane.jpg
```

  - (To be fair, if there were a break for SHA-256 it would be a different image pair)

Audience Participation - Birthday

- A birthday attack is successful when attackers find:

a) Two images that look the same but have different binary data.

b) Two students in CMPT 201 who have the same birthday.

c) A second document which matches the hash of a single given document.

d) Hash collision of a benign and malicious document.

Summary

- *Passwords*
  - Store **salted** and hashed passwords to mitigate **rainbow tables**
- *Digest*
  - A hash of a document
- *Digital Signatures*
  - Sign a **hash** with a **private key**
- *Digital Certificates*
  - Sign document to show who really owns a public/private key
  - **Chain of trust** for distributing certificates
  - **Root of trust** built into OS
- *Hash Collisions*
  - Duplicate hash (digital signature) is a security issue
  - **Birthday attack** to find duplicates