Week 12: Security in Distributed Systems
Part 2: Data Integrity
Integrity: Goals

Use cryptographic techniques to detect that data has not been modified

Integrity mechanisms can help to

– Detect data corruption
– Detect malicious data modification
– Prove ownership of data
How do we detect that a message has been tampered?

- A cryptographic hash acts as a checksum

  A hash is a small, fixed amount of information that lets us have confidence that the data used to create the hash was not modified

- We associate a hash with a message
  - we’re not encrypting the message
  - we’re concerned with integrity, not confidentiality

- If two messages hash to different values, we know the messages are different
  \[ H(M) \neq H(M') \]
Cryptographic hash functions

Properties

– Arbitrary length input → **fixed-length output**

– **Deterministic**: you always get the same hash for the same message

– **One-way function** (pre-image resistance, or **hiding**)
  • Given $H$, it should be difficult to find $M$ such that $H = \text{hash}(M)$

– **Collision resistant**
  • Infeasible to find any two different strings that hash to the same value:
    Find $M, M'$ such that $\text{hash}(M) = \text{hash}(M')$

– **Output should not give any information about any of the input**
  • Like cryptographic algorithms, relies on **diffusion**

– **Efficient**
  • Computing a hash function should be computationally efficient

Also called **digests** or **fingerprints**
Hash functions are the basis of integrity

• Not encryption

• Can help us to detect:
  – Masquerading
    Insertion of message from a fraudulent source
  – Content modification
    Changing the content of a message
  – Sequence modification
    Inserting, deleting, or rearranging parts of a message
  – Replay attacks
    Replaying valid sessions
### Some Popular Hash Functions

<table>
<thead>
<tr>
<th>Hash Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MD5</strong></td>
<td>128 bits</td>
</tr>
<tr>
<td></td>
<td>Linux passwords used to use this</td>
</tr>
<tr>
<td></td>
<td><strong>Rarely used</strong> now since weaknesses were found</td>
</tr>
<tr>
<td><strong>SHA-1</strong></td>
<td>160 bits – was widely used: checksum in Git &amp; torrents</td>
</tr>
<tr>
<td></td>
<td>Google demonstrated a <em>collision attack</em> in Feb 2017</td>
</tr>
<tr>
<td></td>
<td>... Google had to run &gt;9 quintillion SHA-1 computations to complete the attack</td>
</tr>
<tr>
<td></td>
<td>... but already being phased out since weaknesses were found earlier</td>
</tr>
<tr>
<td></td>
<td>Was for message integrity in GitHub (SHA-256 fully supported as of 2023)</td>
</tr>
<tr>
<td><strong>SHA-2</strong></td>
<td>Believed to be secure</td>
</tr>
<tr>
<td></td>
<td>Designed by the NSA; published by NIST</td>
</tr>
<tr>
<td></td>
<td>Variations based on bit length: SHA-224, SHA-256, SHA-384, SHA-512</td>
</tr>
<tr>
<td></td>
<td>Linux passwords use SHA-512</td>
</tr>
<tr>
<td></td>
<td>Bitcoin uses SHA-256</td>
</tr>
<tr>
<td><strong>SHA-3</strong></td>
<td>Believed to be secure</td>
</tr>
<tr>
<td></td>
<td>256 &amp; 512 bit</td>
</tr>
<tr>
<td><strong>Blowfish</strong></td>
<td>Used for password hashing in OpenBSD</td>
</tr>
<tr>
<td><strong>3DES</strong></td>
<td>Linux passwords used to use this</td>
</tr>
</tbody>
</table>

**Derived from ciphers**
Tamperproof Integrity: Message Authentication Codes and Digital Signatures
Message Integrity: MACs

• We rely on hashes to assert the integrity of messages.

• An attacker can create a new message $M'$ & a new hash and replace $H(M)$ with $H(M')$.

  "Hello, Jib!"
  Hash=a8e02b1...
  "Hello, Jab!"
  Hash=4d77ea1...

• So, let’s create a checksum that relies on a key for validation:
  Message Authentication Code (MAC) = $hash(M, key)$.
Message Authentication Codes (MAC)

Hash of message and a symmetric key:
An intruder will not be able to replace the hash value
  – You need to have the key and the message to recreate the hash

MACs provide message **integrity**
  • The hash assures us that the original message has not been modified
  • The encryption of the hash assures us that an attacker could not have re-created the hash
Digital Signatures

Create a hash that anyone can verify but only the owner can create:

Hash of message encrypted with the owner’s private key

• Alice encrypts the hash with her private key
• Bob validates by decrypting it with her public key & comparing with a hash of the message

Digital signatures add non-repudiation

• Only Alice could have created the signature because only Alice has her private key
1. **Key generation:** \{ signing_key, verification_key \} := \texttt{gen_keys}(key\_size)

   signing_key = private_key, \( k \)
   verification_key = public_key, \( K \)

2. **Signing:** signature := \texttt{sign}(message, private_key)

   signature := \texttt{sign}(message, private_key)
   \[\Rightarrow\] signature := \( E_k(\text{hash}(\text{message})) \)

   The signature uses a \texttt{hash(message)} instead of the \texttt{message}
   - We’d like the signature to be a small, fixed size
   - We are not hiding the contents of the message
   - We trust hashes to be collision-free

3. **Validation:** verify(verification_key, message, signature)

   \( D_K(\text{signature}) \equiv \text{hash}(\text{message}) \)
Digital signatures

Alice generates a hash of the message, $H(P)$.
Digital signatures: public key cryptography

Alice encrypts the hash with her private key
This is her signature.
Using Digital Signatures

Alice

H(P) → S = E_a(H(P))
moderation?

Bob

Alice sends Bob the message & the encrypted hash
Using Digital Signatures

1. Bob decrypts the hash using Alice’s public key
2. Bob computes the hash of the message sent by Alice
Using Digital Signatures

If the hashes match, the signature is valid
⇒ the encrypted hash *must* have been generated by Alice
Digital signatures & non-repudiation

• Digital signatures provide **non-repudiation**
  – Only Alice could have created the signature because only Alice has her private key

• **Proof of integrity**
  – The hash assures us that the original message has not been modified
  – The encryption of the hash assures us that an attacker could not have re-created the hash
Digital signatures: multiple signers

Charles:

- Generates a hash of the message, $H(P)$
- Decrypts Alice’s signature with Alice’s public key
  - Validates the signature: $D_A(S) \overset{?}{=} H(P)$
- Decrypts Bob’s signature with Bob’s public key
  - Validates the signature: $D_B(S') \overset{?}{=} H(P)$
While encrypting a hash with a private key produces a digital signature, there are dedicated algorithms that use public key cryptography to produce and validate signatures

- **RSA** – based on RSA cryptography & keys
  - Difficulty based on factoring products of primes

- **DSA** (Digital Signature Algorithm)
  - Developed by NIST (National Institute of Standards and Technology)
  - Difficulty based on discrete logarithms & modular exponentiation

- **ECDSA, EdDSA**: Elliptic Curve Digital Signature Algorithm
  - Difficulty based on discrete logarithms on elliptic curves
  - (EdDSA = Edwards curve DSA – uses Twisted Edwards curves)
  - Newest & fastest signature algorithm
Covert AND authenticated messaging

If we want to keep the message secret

– combine encryption with a digital signature

Use a session key:

– Pick a random key, $K$, to encrypt the message with a symmetric algorithm

– Encrypt $K$ with the public key of each recipient

– For signing, encrypt the hash of the message with sender’s private key
Alice generates a digital signature by encrypting the message with her private key.

\[ S = E_a(H(M)) \]
Covert and authenticated messaging

Alice picks a random key, $K$, and encrypts the message $M$ with it using a symmetric cipher.

$C = E_K(M)$

$H(M)$

$S = E_a(H(M))$
Covert and authenticated messaging

Alice encrypts the session key for each recipient of this message using their public keys.
Covert and authenticated messaging

The aggregate message is sent to Bob & Charles
The End