Lecture 7: Public-key Infrastructure
Plan
* Recap: Digital Signatures
* Signatures in practice
* Public-key infrastructure (PKI)
  - API/Goal
  - Common strategies
  - Common pitfalls

Logistics
* Lab 1 theory & code due tomorrow 10pm ET
* Lab 2 out on 9/30

(set up laptop)
Recap: Digital Signatures

Key idea: Message integrity w/o shared secret
- (Gen, Sign, Verify)
- Unlike MAC or password-based auth
- Not really revolutionary - no shared secret!

Hash-based signature
- (Unbounded msg len, many tme sec)

To sign msg, use $vk_m$ to sign.
* Return all $vk_i$ on path to root with siblings
* Use $ski_i$ to sign $(vk_{i0} || vk_{i1})$
* Return all signatures.

[See lecture notes for a more formal description.]
Signatures in practice (briefly)

- One of the most widely used crypto tools
  * HTTPS
  * Software updates
  * Encrypted messaging
  * SSH
  * VPN
  * Essentially any protocol that sends msgs over the Internet

- Two widely used protocols... both use "hash & sign"
  1. RSA (classic, going away)
  2. EC-DSA + friends (extremely popular)

  (both based on hard problems in number theory)
## Choice of Sig Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>PK size</th>
<th>Sig size</th>
<th>Sign/s</th>
<th>Ver/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPHINCS+ - 128</td>
<td>32B</td>
<td>8000B</td>
<td>5</td>
<td>750</td>
</tr>
<tr>
<td>(2010s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSA - 2048</td>
<td>256B</td>
<td>256B</td>
<td>2,000</td>
<td>50,000</td>
</tr>
<tr>
<td>(1970s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECDSA 256</td>
<td>32B</td>
<td>64B</td>
<td>42,000</td>
<td>14,000</td>
</tr>
<tr>
<td>(Schnorr, Ed25519)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SHA256 Hash 64 bytes \( \approx 10,000,000 / s \)

- 99% of time, use ECDSA (or modern variant)

- In rare cases, want to choose a different scheme:
  - Post-quantum security (RSA and ECDSA aren't Hash-based sigs seem to be. Also lattice-based.)
  - Extra Features: aggregation, blind signing, etc...

![Table with Sig Scheme Comparisons](image-url)
Public-key infrastructure (PKI)

Last year

The right image...

How do I know it was dean who sent me this email?

Now that we have signatures, answer is clear!

But where do we get \( \text{VK}_{\text{Dean}} \)?
Option: Use public key as name.

Dean's "name" is the vk.
Instead of calling him "Dan",
call him 0x2EEC9DB3...0668

- Can imagine that at birth, we're each given an (sk, vk) pair. Everyone calls us by vk.

This sort-of works! Used in Bitcoin & friends, also Tor hidden services, ...

Problem: Cumbersome. Hard to remember 32B names.

Problem: What happens if you lose your secret key? Or if it gets stolen? Or you realize you generated it incorrectly?
PKI is all about mapping...

human-intelligible Names to public keys.

email addr
domain name
legal entity
phone #
kerberos ID

Can think of PKI as having the API (grossly simplified)

\( \text{IsKeyFor}(vk, \langle \text{name} \rangle) \rightarrow \{0, 1\} \)

* Many many ways to implement a PKI...
  ... we will see some.

* But all serve this same purpose.

* No “perfect” solution here — lots of trade-offs.

We will look at a few common schemes...
  * key as name, TOFU, cert based
Trust on first use (TOFU)

- Accept only first key you see for a name.

Client keeps a cache = \$3

```
IsKeyFor \( (vk, \text{name}) \):

if name not in cache:
    cache [name] = vk
    return true

else:
    return rk == cache [name]
```

Used in SSH, Signal, WhatsApp
(Could use this in my email example. Protection if I have already gotten email from Dean)

Pros:
- Simple
- Easy to understand
- Surprisingly effective - protects you against an attacker that hijacks 2nd connection

Cons:
- No protection on first communication
- What happens when key changes?

\( \rightarrow \) SSH: Warn... then what?
Trust on first use (TOFU)

Accept only first key you see for a name.

\((\text{phrpean, msg, } \sigma)\)

Check \(\hat{pk}_{\text{dean}} = pk_{\text{dean}}\)

Verify \(\sigma\) on \(\text{msg}...\)
Certificate-Based System

Let certification authorities (CA) manage name → key mapping

Client keeps a list of known CAs’ Verif keys:

CA\_s = \{ vk\_verisign, vk\_google, \ldots \}

List of CAs is packaged with browser/OS.

Client accepts (vk, name) pair if known CA signed it.

CAs "attest" to name → vk mappings.

\[ \text{IskeyFor(}(vk, o), \text{name}) : \]

\[
\begin{align*}
\text{For each vk\_CA in CA\_s:} & \\
\text{if Verify(} & \text{vk\_CA, (vk, name), o) } \\
\text{return true} & \\
\text{return false} & \\
\end{align*}
\]

When a client generates a new keypair, it must get a CA to sign its vk

[PB key certs introduced in 1978 by Loren Kohnfelder]
Certificate Issuance

\[(sk, vk) \leftarrow \text{Gen}(\cdot)\]

\[\rightarrow (vk, \text{me@mit.edu}), $$\$\$

\[\text{Verify that I own me@mit.edu}\]

\[\rightarrow \sigma\]

\[\sigma \leftarrow \text{Sign}(sk_{ca}, (vk, \text{me@mit.edu}))\]

Common extension: Accept a \((vk, \text{name})\) pair if it's signed by someone whose key was signed by a known CA

Lots of extra metadata in cert: Expiration date, ...

Used on web (HTTPS/TLS), code signing, SMIME, ...

... also at MIT

Pros: - Client only needs a few vks - scales well!
- Client can choose which CAs to trust
- No online interaction w/ CA

Cons: - Weakest link security - attacker who compromises one CA can impersonate anyone!
- Validation is typically pretty weak... TOFU almost...
Demo

- Show cert & chain of trust for mit.edu

- Dump CRL data
  ```bash
  openssl crl -inform DER -text -noout -in <CRL>
  ```

- Q: Why intermediate CAs?

There are many variants on certificate-style systems - key directory, web of trust, ...  

"key" idea: To prove (vk, name) binding, I can give you signature on (vk, name) from someone you trust.
Problems with CA-based PKI

1. Any malicious/comprised CA can issue certs for any domain.
   → Your browser trusts many sketchy CAs (goats, random businesses, etc.)
   → "AAA cert service" can issue cert for mit.edu... you'll never know

2011: - Digistar signing key stolen
   - Attacker used it to issue cert for google.com
   - Used to decrypt Gmail traffic in Iran
   - Browsers pull Digistar from list of known CAs
   - Dutch govt websites break

"Certificate transparency" is one partial answer...

2. Revocation is difficult...
Revocation
- After a CA has issued a cert, it may want to revoke it → make sure clients reject it in the future.

Why?
* Site owner has their secret key stolen (Heartbleed) - 2011
* Site owner realizes they generated key using bad randomness (Debian bug) - 2008
* MIT student graduates, account inactivated
* Crypto standards change (SHA1, RSA1024, ...)

Approach: Expiration
* Cert has expiration date, clients will reject cert after that date
* If expiration date is not far away, this handles many routine revocation cases
  e.g. MIT certs expire June 30 every year.
  e.g. Let's Encrypt uses 90-day expiration
Approach: Software vendor (e.g. Mozilla) ships update to client w/ full list of revoked certs.

- Window of vulnerability: as long as update latency
- B/w storage cost after wave of revocations

“CRLSet” “CRLike”

Approaches: fallen out of favor

- Certificate revocation list (CRL)
  - Ask CA for list of all revoked unexpired certs
    - expensive after a wave of revocations
    - what happens if can’t reach CA server?

OCSP
  - Ask CA each time you use cert
    - browsing history leaks to CA
    - CA on critical path of page load

“Stapling” = short-lived cert
Bottom line:

PKI is about names → public keys

Key idea: Certificates signed attestation of name → vk binding

Key challenge: Revocation stolen key, invalid binding
Recap: Many-time signatures from one-time sigs

Claim: Given a PRF w/ key space $\mathcal{K}$, a one-time signature scheme $(Gen_0, Sign_0, Ver_0)$

Can construct a $2^t$-time secure sig scheme for all $t \geq 0$ where running time of all algfs grows as $\text{poly}(t)$.

Pf idea: By induction on $t$

Base case ($t=0$): This is one-time scheme. Done.

Induction: Assume for $t-1$.

$$
\begin{align*}
\text{Gen}_t(\cdot): & \quad \exists k \in \mathcal{K} \quad \text{PRF Key} \\
& \{ (sk_\pi, vk_\pi) \leftarrow \text{Gen}_k(\cdot) \} \\
& \text{output } (k, vk_\pi)
\end{align*}
$$

$$
\begin{align*}
\text{Sign}_t(k, m): & \quad (sk_\pi, vk_\pi) \leftarrow \text{Gen}_k(\cdot) \\
& \{ (sk_0, vk_0) \leftarrow \text{Gen}_{k-1}(\cdot) \\
& \{ (sk_1, vk_1) \leftarrow \text{Gen}_{k-1}(\cdot) \\
& \sigma_\pi \leftarrow \text{Sign}_0( sk_\pi, v k_0 \parallel v k_1 ) \\
& \sigma_m \leftarrow \text{Sign}_{k-1}( sk_m \parallel v k_1, m[1:] ) \\
& \text{output } \sigma = (vk_0, vk_1, \sigma_\pi, \sigma_m)
\end{align*}
$$

Grows linearly with $t$!

$$
\begin{align*}
\text{Ver}_t(\cdot, \cdot, \cdot): & \quad \exists (vk_0, vk_1, \sigma_\pi, \sigma_m) \leftarrow \sigma
\end{align*}
$$

$$
\begin{align*}
\text{Ver}_t(vk_\pi, vk_0 \parallel vk_1, \sigma_\pi) & \quad \forall \sigma_\pi \leftarrow \text{Sign}_0( sk_\pi, v k_0 \parallel v k_1 ) \\
\text{Ver}_{t-1}(vk_m, m[1:], \sigma_m) & \quad \forall \sigma_m \leftarrow \text{Sign}_{k-1}( sk_m \parallel v k_1, m[1:] )
\end{align*}
$$
How to detect "rogue" CA?

- Have client software look for certain misbehavior.
  e.g. Chrome has list of Google vcs hardcoded.
  If CA issues a rogue Google cert, Chrome will (I believe) notify Google.

  Doesnt really solve the problem.
  Only works for friends of Google.
  If client knew what the right cert was, wouldnt need PKI.

Certificate Transparency (some browsers, sort of)

- Require CAs to publish all certs they sign in a public log...many logs run by many different orgs.

- mit.edu can inspect logs regularly to make sure that no CA has issued rogue certs for its domains.

- In theory, when browser gets a cert from a web server, it can "audit" the cert by checking that it appears in the log.

- Lots of messy implementation details
  - prevent logs from cheating
  - ensure that everyone sees same log
  - ensure that client can audit recently issued certs
  - privacy issues w/ auditing