Public-key Infrastructure

6.1600 Course Staff

Fall 2023

In the last chapter, we discussed digital signatures, which allow us to authenticate messages without a shared secret. For example, if I have the public signature-verification key of the university dean, I can verify that signed emails from the dean really came from her and not from someone pretending to be her. But to verify the signature on the dean's message, I need to know her signature-verification key vk. How can I (the recipient) obtain this verification key without a secure channel to the dean (the sender)?

Unfortunately, there are no perfect solutions to this problem. In this section, we will discuss some of the approaches that we use in practice.

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

1 *Public-key infrastructure*

The goal of a public-key infrastructure is to facilitate the mapping of **human-intelligible names** to **signature-verification keys**. Examples of human-intelligible names that we map to keys are: email addresses, domain names, legal entities, phone numbers, and usernames (e.g., within a company).

We can think of the public-key infrastructure as implementing the following (grossly simplified) API:

$$\mathsf{IsKeyFor}(\mathsf{vk}, \mathtt{name}) \to \{0,1\}.$$

That is, given a verification key vk and a name $\mathtt{name}$, the public-key infrastructure gives a way to check whether this mapping is valid.

We now discuss some ways to implement this API.

2 *Option 1: Use verification keys as names*

One option is to just refer to everyone by the bytes of their signature-verification key. This way, there is no need to do a messy name-to-verification-key translation at all.

This is not practical for humans generally: it would be difficult to remember your friends' names if you had to call them by random 32-byte strings! However, some digital services such as Bitcoin indeed use verification keys as identities: when you want to transfer Bitcoin to someone, you send the coins to an account identified by their public key. The public key *is* name of that account.

Using keys as names has a two major problems:

- Verification keys are hard to remember. Things like email addresses, domain names, kerberos usernames, phone numbers, and so on, are much easier for humans to remember.
- There is no way to update the name-to-key mapping. If you lose the secret key associated with your name/account, there is no way to "update" the key to a new value. In practice, people lose their secret keys all the time, so supporting key updates is critical in most systems.

3 *Trust on first use (TOFU)*

Another strategy is to avoid having any global mapping from names to verification keys. Instead, a client can just accept the first verification key that it sees associated with a given name. The secure shell system (SSH) uses TOFU for key management by default.

In particular, the key-validation logic looks like this:

> keymap $\leftarrow \{\}$.
>
> IsKeyFor(vk, name) :
> - If keymap[name] is undefined:
>   - Set keymap[name] $\leftarrow$ vk.
>   - Return true..
> - Else: Return keymap[name] $==$ vk.

That is, the client will accept the *first* verification key it sees associated with a particular name. Later on, the client will only accept the same verification key for that name.

TOFU is very simple to implement and provides a meaningful security guarantees. There are two drawbacks:

- If the first key that client receives for a particular name is incorrect/attacker-generated, the attacker can forge signatures under that name.
- It is not clear with TOFU how to handle key updates. In most systems that use TOFU, whenever the sender's key changes, the system notifies the user and allows them to accept or reject the new key. The burden is then on the user to figure out whether the sender really did change their signing keypair, or whether there is an attack in progress.

4 *Certificates*

Another option is to rely on a few parties to manage the mapping of names to public keys. These entities are called *Certificate Authorities*

(CAs). Your operating system and web browser typically come bundled with a set of roughly 100 public signature-verification keys, owned by each of 100 CAs.

Whenever the owner of website `example.com`, for example, generates a new public key $\mathsf{vk}_{\texttt{example.com}}$, the website owner can ask a certificate authority to certify that $\mathsf{vk}_{\texttt{example.com}}$ really belongs to `example.com`. The certificate authority does this by signing the pair $(\texttt{example.com}, \mathsf{vk}_{\texttt{example.com}})$ using its own signing keypair vk_{CA} to generate a signature σ_{CA}. This signed attestation $(\texttt{example.com}, \mathsf{vk}_{\texttt{example.com}}, \sigma_{\text{CA}})$ is called a *certificate*.

In practice the structure of certificates is much more complicated than we are showing here, and include all sorts of additional metadata. But the basic idea is the same as we describe here.

When a client connects to `example.com`, the server at `example.com` will supply the client with the certificate $(\texttt{example.com}, \mathsf{vk}_{\texttt{example.com}}, \sigma_{\text{CA}})$. As long as this certificate was signed by a CA that the client trusts (i.e., a CA for which the client has a verification key), the client can validate the certificate and conclude that the verification key $\mathsf{vk}_{\texttt{example.com}}$ really belongs to `example.com`.

In pseudocode the logic for verifying certificates looks like this:

$\mathsf{caKeys} \leftarrow \{\mathsf{vk}_{\text{Verisign}}, \mathsf{vk}_{\text{Let's Encrypt}}, \dots\}$.

$\mathsf{IsKeyFor}((\mathsf{vk}, \sigma), \mathsf{name})$:

- For each vk_{CA} in caKeys:
  - If $\mathsf{Ver}(\mathsf{vk}_{\text{CA}}, (\mathsf{name}, \mathsf{vk}), \sigma) = 1$: Return true.
- Return false.

A very nice feature of certificate-based public-key infrastructure is that the client does not need to communicate with the CA to validate a name-to-key mapping. The client only needs to perform one signature-verification check.

Certificates in practice works quite well:

- The client only needs to store ≈ 100 CA verification keys, and yet the client can validate the name-to-key mappings for millions of websites.
- A client can choose which CAs to trust (though in practice, clients typically delegate this decision to software vendors).
- The client never needs to interact with the CA.

However, certificates still have some drawbacks:

- If an attacker compromises *any* CA, they can generate certificates for any domain.
- Certificate authorities often perform quite minimal validation of domain ownership.
- If a server's private key gets stolen, there is no great plan for *revoking* or updating a name-to-key mapping.

In order to use TLS on a website you own, you need to convince one of the certificate authorities to give you a certificate—i.e., to sign your $(\mathsf{name}, \mathsf{vk})$ pair. To do so, the CA will have some protocol to follow—typically, you will send your $(\mathsf{name}, \mathsf{vk})$ pair to the CA, who will then ask you to verify that you own the name somehow. In the case of web certificates, the CA may verify ownership by requiring you to upload a file file to your server, to add a new DNS record with a random value, or something similar. Once the CA is convinced that you own the domain, the CA will reply with a certificate: a signature over the tuple $(\mathsf{name}, \mathsf{vk})$. This $(\mathsf{name}, \mathsf{vk}, \sigma_{\text{CA}})$.

4.1 Revocation

In many cases, a CA will want to delete or change a name-to-key mapping. This process is called *certificate revocation*. There are several possible reasons for this:

- The owner of a verification key may have their corresponding secret key be lost or stolen.
- A company may want to rotate keys, for example to update to a new cryptographic algorithm.
- A website may go out of business and another entity buys their domain name.
- Software bugs may lead a user to generate an insecure keypair that they later want to revoke.[1]

[1] Scott Yilek et al. "When private keys are public: Results from the 2008 Debian OpenSSL vulnerability". In: *SIGCOMM*. 2009; Matus Nemec et al. "The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli". In: *CCS*. 2017.

In a scheme that uses certificates, this seems like a hard problem: since there is no interaction with the CA to verify a certificate, there is no way for a CA to "take back" a certificate. There are again no excellent solutions to this, but there are a few strategies used in practice.

Expiration Dates One pragmatic way to handle revocation is to add an expiration date to each generated certificate—if this expiration date has passed, the client will reject the certificate. This way, a server will need to re-authenticate to the CA that they own the name that they claim to own periodically. So, for example, an attacker that steals a website's secret key will only be able to use it until the certificate expires. In practice, certificates used on the Internet typically expiration dates between 90 days and 1-2 years.

Software Updates Another solution is for the browser (or client, more generally) to maintain a list of revoked certificates. On every connection, the browser will check whether the provided certificate is in this local revocation list and refuse it if so. Since browsers today check for updates very frequently, this strategy can respond to a stolen secret key quickly. However, there is a large storage cost since now every browser needs to store this (potentially large) list of revoked certificates.

CA Revocation List To avoid depending on browser manufacturers to update this revocation list, another strategy is to ask the CA for it directly. One way to do this is similar to the above: periodically query the CA to download its updated revocation list and check that each certificate is not in this list. This method has fallen out of favor, in part because clients (e.g., behind corporate firewalls) cannot connect directly to the CAs to download these revocation lists.

References

Nemec, Matus et al. “The Return of Coppersmith’s Attack: Practical Factorization of Widely Used RSA Moduli”. In: *CCS*. 2017.

Yilek, Scott et al. “When private keys are public: Results from the 2008 Debian OpenSSL vulnerability”. In: *SIGCOMM*. 2009.