Today: Using cryptography in practice

In practice, things are messier than in theory, and that is where the attacks often happen.

- Public-key Infrastructure
- TLS (Transport Layer Security)

Q: When I get an email from Kyle, how do I know that Kyle indeed sent the email?

A: Kyle can append a digital signature to her email.

To verify I need to know Kyle's pk.

Q: How do I know Kyle's pk??

So far, we assumed that the pk's are magically known!

How do we all know each others pk's??

In some applications, such as bitcoin, the pk is the "name".

Or one can use identity based encryption!

Problem: What if sk gets lost/stolen?

Our identity is lost/stolen!

Goal: Have a mapping from human intelligible names to pk's

IsKeyFor(name,pk) = 0/1

Public-key Infrastructure (PKI)

Example 1: TOFU (Trust on First Use)

Keep a table of pairs (name,pk).

IsKeyFor(name,pk)=1 if and only if name is not in the table

or if (name,pk) in the table.

Used in SSH

Pros:
1. Super simple.
2. It is secure if first communication is secure
3. Surprisingly effective

Cons:
1. No protection if 1st interaction is attacked.
2. Not clear how to handle a key change

(SSH sends a warning but then what?)

Certificate based PKI

A client accepts (name,pk) if and only if a known

"certification authority" (CA) signed it.

The CA is in charge of certifying the mapping from

human memorable names to pk's

Example: MIT runs a CA.

When I set up my account at MIT, I received the pk of MIT CA.

When Ron sends a message to me, it is signed with Ron's pk,

and the pair ("Ron",pk) is signed by the MIT CA,

which attest that indeed this pk corresponds to "Ron".

In general, each client keeps a list of CA that it trusts,

and their PKs (Ex. Google runs a CA).

A client will accept (name,pk) iff this pair is signed by one of the

CA's that it trusts.

Which CA's do we trust, and how do we get their pk's??

When the MIT CA send me its pk, how do I know that it is indeed the pk of MIT?

Chain of trust:

At the top of the chain are the root CAs.

Computers such as Lenovo or Del (that run Windows) come with a list of CA's that the computer trusts.

(My computer has 62 such CA's.)

These CA's are called root CA's.

MIT CA is not a root CA, and hence its pk will be signed by another CA, whose pk may be signed by another CA, etc. until we reach a root CA.

We need to trust Del or Lenovo, and their CAs

The fist CA on the list in my computer is "AAA certificate Services", Not sure what this company is nor why should I trust it.

Example 1: There was an issue with Lenovo:

They shipped computers with a malicious CA installed, that was intercepting all the traffic from the laptop for the purpose of injecting adds. This attack is called Superfish.

Example 2: In 2011 the sk of a CA called Diginotar was stolen or given away. Attackers used it to fake a certificate for google.com. They used this certificate to intercept hundreds of thousands of users in Iran.

How can we detect rogue CA's?

Certificate transparency

A new technology (last 8 years or so) whose purpose is to deal with rogue CAs

1. All CA's posts all the certificates they issued in a public log.
2. Before a certificate is accepted, the browser checks that it is in the public log.
3. Owner of a name (say mit.edu) checks that this log does not contain certificates that were issued for mit.edu that should not be there.

Grossly oversimplified...

Certificate Revocation

After a CA has issued a certificate, it may want to revoke it.

Why?

1. The sk of the owner was stolen
2. Account deactivated (ex. MIT student graduated).

Once a CA signed a (name, pk) pair, it can't unsign it!

1. Expiration: All certificates have an expiration date.

Ex. MIT's certificates expire on June 30th

2. Certificate Revocation Lists:

Client software contains a list of revoked certificates (CRL).

(There is a window of vulnerability...)

How does the CA verify that I am who I claim I am??

Certificate Issuance:

A user sends the CA his "name" and pk (and money if it is a commercial CA). Then it runs a protocol with the CA where the user proves that he "owns the name".

Ex., if I want a certificate for a domain, say www.yael.org, the CA can ask me to post something on my domain to see that it is indeed me.

If the adv can corrupt this protocol he can obtain certificates for domains that it does not control!

Summary:

Pros:

1. User has the flexibility to choose which CA's she trusts.
2. The client only needs to keep a few pk's of CA's that it trusts (scales well).
2. No online interaction with the CA's!

Cons:

1. Single point of failure. Adv. needs to bribe only one CA!
2. Validation is quite weak.

At MIT for example we use passwords, so the CA can verify who we are from our passwords, but in general validation is hard.

One example of where this is all used in practice is in the TLS protocol.

The TLS (Transport Layer Security) Protocol

TLS Converts a TCP connection to a secure TLS connection,

which is encrypted and authenticated connection.

HTTPs is the same as HTTP where TCP was replaces with TLS

Seems like TLS should be easy, given that we know how to encrypt and sign!

Why is this hard??

Client and servers run different software, which supports different

cryptographic algorithms

For example, AESGCM is relatively new, if your software is old it will not support it.

Version/protocol negotiation is difficult!

We need to ensure that the attacker cannot trick us to use an old insecure version of

the protocol. Such attacks are called downgrade attacks.

Structure of TLS (V1.3): It consists of two parts

Phase 1: Handshake protocol

This is a key exchange protocol

At the end of the handshake protocol we agree on a (AESGCM) sk

Phase 2: Record protocol

the client and server exchange messages in an encrypted and authenticated way

The complicated part is the handshake protocol, where we need to agree on a key with someone we never talked to before

Example: Suppose I wish to talk to the MIT server.

How do I know that I am indeed talking with MIT,

and agreeing on a sk with MIT and not with evil.com?

The RFC for TLS has 7 security properties!

This is a bit worrysome. First, this is very complicated.

Second, how do we know that they didn't miss a 8'th property?

The properties:

1. Correctness (if both parties are honest they both agree on the same sk).

2. Security (session sk looks random to an attacker).

3. Security against downgrade attacks

(the version they run is the same whether the attacker is there or not).

4. Peer authentication (the client is indeed sharing a sk with MIT and not evil.com).

5. Forward secrecy w.r.t. key compromise

(if adv learns all sk's stored on the server, it should not be able to decrypt data from the past).

DH does not satisfy forward security!

6. Protection against key compromise impersonation.

(If Evil compromises Alice's sk, Evil should not be able to pretend to be Google to Alice.

We don't want Evil to be able to learn all of Alice's private information, such as credit card num.)

7. Protection of end-point identities (against passive attacks).

If the adv listens to the handshake it should not be able to learn the certificate

(of the client of server),

so the adv will not learn which website we are visiting.

The handshake protocol: (Simplified protocol)

Client has a pk of CA, denoted by pk_{CA}

Server has his own sk_s, and a signature $sig(pk_s)$

from CA certifying the server's public key.

The handshake protocol consists of 4 rounds.

1. Client sends "hello" msg:

   Sends a list of ciphers it supports

(including a group G of order q for DH key exchange)

It also chooses random r_c in $\{1,\ldots,q\}$ and sends $R_c = g^{r_c}$

2. Server sends "hello" msg:

Sends a ciphertext he is willing to use from the list.

It also chooses random r_s in $\{1,\ldots,q\}$ and sends $R_s = g^{r_s}$

From now on traffic is encrypted with DH key secret

$H(g^{r_c \cdot r_s})$, using say AES-GCM.

3. Server sends its certificate and a signature on all the transcript so far, signed using its secret signing key.

4. Client does the following:

a. Computes the shared secret $K = H(g^{r_c r_s})$

b. Checks that cert is valid,

c. Checks the transcript sig (under the server's pk)

d. It sends the server a MAC over the transcript seen so far, using a key derived from K (indicating that it is happy).

This simplified version does not contain the downgrade protection part.

Quite complicated!

Note that the first two rounds have no protection,

and thus the client does not know that it is talking with

the correct server.

This is by design, it hides who the client is talking with.

We want the authentication to be encrypted

for the sake of privacy.

Indeed the server authenticates itself in round 3.

Forward security is satisfied since after the TLS connection

is closed, the server will erase the DH key used in this protocol