13. Certificates

13. Certificates

So far we've seen powerful techniques for securing communication such that the only information we must carefully protect regards "keys" of various sorts. Given the success of cryptography in general, arguably the biggest challenge remaining for its effective use concerns exactly those keys, and how to *manage* them. For instance, how does Alice find out Bob's public key? Does it matter?

13.1. Man-in-the-middle Attacks

Suppose Alice wants to communicate securely with Bob over an insecure communication channel, but she doesn't know his public key (and he doesn't know hers). A naive strategy is that she could just send Bob a message asking him to send his public key, and accept whatever response she gets back (over the insecure communication channel). Alice would then encrypt her message using the public key she received in this way.

This naive approach is insecure. An *active attacker* (Mallory, in our usual terminology) could tamper with Bob's response, replacing the public key in Bob's response with the attacker's public key. When Alice encrypts her message, she'll be encrypting it under Mallory's public key, not Bob's public key. When Alice transmits the resulting ciphertext over the insecure communication channel, Mallory can observe the ciphertext, decrypt it with his private key, and learn the secret message that Alice was trying to send to Bob.

You might think that Bob could detect this attack when he receives a ciphertext that he is unable to decrypt using his own private key. However, an active attacker can prevent Bob from noticing the attack. After decrypting the ciphertext Alice sent and learning the secret message that Alice wanted to send, Mallory can re-encrypt Alice's message under Bob's public key, though not before possibly tampering with Alice's packet to replace her ciphertext with new ciphertext of Mallory's choosing. In this way, neither Alice nor Bob would have any idea that something has gone wrong. This allows an active attacker to spy on—*and alter*—Alice's secret messages to Bob, without breaking any of the cryptography.

If Alice and Bob are having a two-way conversation, and they both exchange their public keys over an insecure communication channel, then Mallory can mount a similar attack in both directions. As a result, Mallory will get to observe all of the secret messages that Alice and Bob send to each other, but neither Alice nor Bob will have any idea that something has gone wrong. This is known as a "*man-in-the-middle*" (MITM) attack because the attacker interposes between Alice and Bob.

Man-in-the-middle attacks were possible in this example because Alice did not have any way of authenticating Bob's alleged public key. The general strategy for preventing MITM attacks is to ensure that every participant can verify the authenticity of other people's public keys. But how do we do that, specifically? We'll look next at several possible approaches to secure key management.

13.2. Trusted Directory Service

One natural approach to this key management problem is to use a trusted directory service: some organization that maintains an association between the name of each participant and their public key. Suppose everyone trusts Dirk the Director to maintain this association. Then any time Alice wants to communicate with someone, say Bob, she can contact Dirk to ask him for Bob's public key. This is only safe if Alice trusts

Dirk to respond correctly to those queries (e.g., not to lie to her, and to avoid being fooled by imposters pretending to be Bob): if Dirk is malicious or incompetent, Alice's security can be compromised.

On first thought, it sounds like a trusted directory service doesn't help, because it just pushes the problem around. If Alice communicates with the trusted directory service over an insecure communication channel, the entire scheme is insecure, because an active attacker can tamper with messages involving the directory service. To protect against this threat, Alice needs to know the directory service's public key, but where does she get *that* from? One potential answer might be to **hardcode** the public key of the directory service in the source code of all applications that rely upon the directory service. So this objection can be overcome.

A trusted directory service might sound like an appealing solution, but it has a number of shortcomings:

- *Trust:* It requires complete trust in the trusted directory service. Another way of putting this is that everyone's security is contingent upon the correct and honest operation of the directory service.
- *Scalability:* The directory service becomes a bottleneck. Everyone has to contact the directory service at the beginning of any communication with anyone new, so the directory service is going to be getting a lot of requests. It had better be able to answer requests very quickly, lest everyone's communications suffer.
- *Reliability:* The directory service becomes a single central point of failure. If it becomes unavailable, then no one can communicate with anyone not known to them. Moreover, the service becomes a single point of vulnerability to denial-of-service attacks: if an attacker can mount a successful DoS attack on the directory service, the effects will be felt globally.
- *Online:* Users will not be able to use this service while they are disconnected. If Alice is composing an email offline (say while traveling), and wants to encrypt it to Bob, her email client will not be able to look up Bob's public key and encrypt the email until she has connectivity again. As another example, suppose Bob and Alice are meeting in person in the same room, and Alice wants to use her phone to beam a file to Bob over infrared or Bluetooth. If she doesn't have general Internet connectivity, she's out of luck: she can't use the directory service to look up Bob's public key.
- *Security:* The directory service needs to be available in real time to answer these queries. That means that the machines running the directory service need to be Internet-connected at all times, so they will need to be carefully secured against remote attacks.

Because of these limitations, the trusted directory service concept is not widely used in practice, except in the context of messengers (such as Signal), where in order to send a message, Alice already has to be online.

In this case, the best approach is described as "trust but verify" using a key transparency mechanism. Suppose Alice and Bob discovered each others keys through the central keyserver. If they are ever in person, they can examine their devices to ensure that Alice actually has the correct key for Bob and vice versa. Although inconvenient, this acts as a check on a rogue keyserver, as the rogue keyserver would know there is at least a chance of getting caught.

However, some of these limitations—specifically, the ones relating to scalability, reliability, and the requirement for online access to the directory service—can be addressed through a clever idea known as digital certificates.

13.3. Digital Certificates

Digital certificates are a way to represent an alleged association between a person's name and their public key, as attested by some certifying party.

Let's look at an example. As a professor at UC Berkeley, David Wagner is an employee of the state of California. Suppose that the state maintained a list of each state employee's public key, to help Californians communicate with their government securely. The governor, Jerry Brown, might control a private key that is used to sign statements about the public key associated with each employee. For instance, Jerry could sign a statement attesting that "David Wagner's public key is `0x092...3F`", signed using the private key that Jerry controls.

In cryptographic protocol notation, the certificate would look like this:

Encryption under a public key: $\{$David Wagner's public key is `0x092...3F`$\}_{PK}$

Signing with private key: $\{$David Wagner's public key is `0x092...3F`$\}_{SK^{-1}_{\text{Jerry}}}$

where here $\{M\}_{SK^{-1}}$ denotes a digital signature on the message M using the private key SK^{-1}. In this case, SK^{-1}_{Jerry} is Jerry Brown's private key. This certificate is just some digital data: a sequence of bits. The certificate can be published and shared with anyone who wants to communicate securely with David.

If Alice wants to communicate securely with David, she can obtain a copy of this certificate. If Alice knows Jerry's public key, she can verify the signature on David's digital certificate. This gives her high confidence that indeed Jerry consented to the statement about the bit pattern of David's public key, because the valid signature required Jerry to decide to agree to apply his private key to the statement.

If Alice also considers Jerry trustworthy and competent at recording the association between state employees and their public keys, she can then conclude that David Wagner's public key is `0x092...3F`, and she can use this public key to securely communicate with David.

Notice that Alice did not need to contact a trusted directory service. She only needed to receive a copy of the digital certificate, but she could obtain it from *anyone*—by Googling it, by obtaining it from an untrusted directory service, by seeing it scrawled on a whiteboard, or by getting a copy from David himself. It's perfectly safe for Alice to download a copy of the certificate over an insecure channel, or to obtain it from an untrustworthy source, as long as she verifies the signature on the digital certificate and trusts Jerry for these purposes. The certificate is, in some sense, self-validating. Alice has *bootstrapped* her trust in the validity of David's public key based on her existing trust that she has a correct copy of Jerry's public key, *plus* her belief that Jerry takes the act of signing keys seriously, and won't sign a statement regarding David's public key unless Jerry is sure of the statement's correctness.

13.4. Public-Key Infrastructure (PKI)

Let's now put together the pieces. A *Certificate Authority* (CA) is a party who issues certificates. If Alice trusts some CA, and that CA issues Bob a digital certificate, she can use Bob's certificate to get a copy of Bob's public key and securely communicate with him. For instance, in the example of the previous section, Jerry Brown acted as a CA for all employees of the state of California.

In general, if we can identify a party who everyone in the world trusts to behave honestly and competently—who will verify everyone's identity, record their public key accurately, and issue a public certificate to that person accordingly—that party can play the role of a trusted CA. The public key of the trusted CA can be hardcoded in applications that need to use cryptography. Whenever an application needs to look up David Wagner's public key, it can ask David for a copy of his digital certificate, verify that it was properly signed by the trusted CA, extract David's public key, and then communicate securely with David using his public key.

Some of the criticisms of the trusted directory service mentioned earlier also apply to this use of CAs. For instance, the CA must be trusted by everyone: put another way, Alice's security can be breached if the CA behaves maliciously, makes a mistake, or acts without sufficient care. So we need to find a single entity whom everyone in the world can agree to trust—a tall order. However, digital certificates have better scalability, reliability, and utility than an online directory service.

For this reason, digital certificates are widely used in practice today, with large companies (e.g., Verisign) having thriving businesses acting as CAs.

This model is also used to secure the web. A web site that wishes to offer access via SSL (`https:`) can buy a digital certificate from a CA, who checks the identity of the web site and issues a certificate linking the site's domain name (e.g., `www.amazon.com`) to its public key. Every browser in the world ships with a list of trusted CAs. When you type in an `https:` URL into your web browser, it connects to the web site, asks for a copy of the site's digital certificate, verifies the certificate using the public key of the CA who issued it, checks that the domain name in the certificate matches the site that you asked to visit, and then establishes secure communications with that site using the public key in the digital certificate.

Web browsers come configured with a list of many trusted CAs. As a fun exercise, you might try listing the set of trusted CAs configured in your web browser and seeing how many of the names you can recognize. If you use Firefox, you can find this list by going to Preferences / Advanced / Certificates / View Certificates / Authorities. Firefox currently ships with about 88 trusted CAs preconfigured in the browser. Take a look and see what you think of those CAs. Do you know who those CAs are? Would you consider them trustworthy? You'll probably find many unfamiliar names. For instance, who is Unizeto? TURKTRUST? AC Camerfirma? XRamp Security Services? Microsec Ltd? Dhimyotis? Chunghwa Telecom Co.? Do you trust them?

The browser manufacturers have decided that, whether you like it or not, those CAs are trusted. You might think that it's an advantage to have many CAs configured into your browser, because that gives each user a choice depending upon whom they trust. However, that's not how web browsers work today. Your web browser will accept *any* certificate issued by *any* of these 88 CAs. If Dhimyotis issues a certificate for `amazon.com`, your browser will accept it. Same goes for all the rest of your CAs. This means that if any one of those 88 CAs issues a certificate to the wrong person, or behaves maliciously, that could affect the security of everyone who uses the web. The more CAs your browser trusts, the greater the risk of a security breach. That CA model is under increasing criticism for these reasons.

13.5. Certificate Chains and Hierarchical PKI

Above we looked at an example where Jerry Brown could sign certificates attesting to the public keys of every California state employee. However, in practice that may not be realistic. There are over 200,000 California state employees, and Jerry couldn't possibly know every one of them personally. Even if Jerry spent all day signing certificates, he still wouldn't be able to keep up—let alone serve as governor.

A more scalable approach is to establish a hierarchy of responsibility. Jerry might issue certificates to the heads of each of the major state agencies. For instance, Jerry might issue a certificate for the University of California, delegating to UC President Janet Napolitano the responsibility and authority to issue certificates to UC employees. Napolitano might sign certificates for all UC employees. We get:

$\{\text{The University of California's public key is } PK_{\text{Napolitano}}\}_{SK^{-1}_{\text{Jerry}}}$

$\{\text{David Wagner's public key is } PK_{\text{daw}}\}_{SK^{-1}_{\text{Napolitano}}}$

This is a simple example of a *certificate chain*: a sequence of certificates, each of which authenticates the public key of the party who has signed the next certificate in the chain.

Of course, it might not be realistic for President Napolitano to personally sign the certificates of all UC employees. We can imagine more elaborate and scalable scenarios. Jerry might issue a certificate for UC to Janet Napolitano; Napolitano might issue a certificate for UC Berkeley to UCB Chancellor Nicholas Dirks; Dirks might issue a certificate for the UCB EECS department to EECS Chair Randy Katz; and Katz might issue each EECS professor a certificate that attests to their name, public key, and status as a state employee. This would lead to a certificate chain of length 4.

In the latter example, Jerry acts as a Certificate Authority (CA) who is the authoritative source of information about the public key of each state agency; Napolitano serves as a CA who manages the association between UC campuses and public keys; Dirks serves as a CA who is authoritative regarding the public key of each UCB department; and so on. Put another way, Jerry delegates the power to issue certificates for UC employees to Napolitano; Napolitano further sub-delegates this power, authorizing Dirks to control the association between UCB employees and their public keys; and so on.

In general, the hierarchy forms a tree. The depth can be arbitrary, and thus certificate chains may be of any length. The CA hierarchy is often chosen to reflect organizational structures.

13.6. Revocation

What do we do if a CA issues a certificate in error, and then wants to invalidate the certificate? With the basic approach described above, there is nothing that can be done: a certificate, once issued, remains valid forever.

This problem has arisen in practice. A number of years ago, Verisign issued bogus certificates for "Microsoft Corporation" to . . . someone other than Microsoft. It turned out that Verisign had no way to revoke those bogus certificates. This was a serious security breach, because it provided the person who received those certificates with the ability to run software with all the privileges that would be accorded to the real Microsoft. How was this problem finally resolved? In the end, Microsoft issued a special patch to the Windows operating system that revoked those specific bogus certificates. The patch contained a hardcoded copy of the bogus certificates and inserted an extra check into the certificate-checking code: if the certificate matches one of the bogus certificates, then treat it as invalid. This addressed the particular issue, but was only feasible because Microsoft was in a special position to push out software to address the problem. What would we have done if a trusted CA had handed out a bogus certificate for Amazon.com, or Paypal.com, or BankofAmerica.com, instead of for Microsoft.com?

This example illustrates the need to consider revocation when designing a PKI system. There are two standard approaches to revocation:

- *Validity periods.* Certificates can contain an expiration date, so they're no longer considered valid after the expiration date. This doesn't let you immediately revoke a certificate the instant you discover that it was issued in error, but it limits the damage by ensuring that the erroneous certificate will eventually expire.

  With this approach, there is a fundamental tradeoff between efficiency and how quickly one can revoke an erroneous certificate. On the one hand, if the lifetime of each certificate is very short—say, each certificate is only valid for a single day, and then you must request a new one—then we have a way to respond quickly to bad certificates: a bad certificate will circulate for at most one day after we discover it. Since we won't re-issue certificates known to be bad, after the lifetime elapses the certificate has effectively been revoked. However, the problem with short lifetimes is that legitimate parties must frequently contact their CA to get new certificates; this puts a heavy load on all the parties, and can create reliability problems if the CA is unreachable for a day. On the other hand, if we set the lifetime very long, then reliability problems can be avoided and the system scales well, but we lose the ability to respond promptly to erroneously issued certificates.

- *Revocation lists.* Alternatively, the CA could maintain and publish a list of all certificates it has revoked. For security, the CA could date and digitally sign this list. Every so often, everyone could download the latest copy of this revocation list, check its digital signature, and cache it locally. Then, when checking the validity of a digital certificate, we also check that it is not on our local copy of the revocation list.

  The advantage of this approach is that it offers the ability to respond promptly to bad certificates. There is a tradeoff between efficiency and prompt response: the more frequently we ask everyone to download the list, the greater the load on the bandwidth and on the CA's revocation servers, but the more quickly we can revoke bad certificates. If revocation is rare, this list might be relatively short, so revocation lists have the potential to be more efficient than constantly re-issuing certificates with a short validity period.

  However, revocation lists also pose some special challenges of their own. What should clients do if they are unable to download a recent copy of the revocation list? If clients continue to use an old copy of the revocation list, then this creates an opportunity for an attacker who receives a bogus certificate to DoS the CA's revocation servers in order to prevent revocation of the bogus certificate. If clients err on the safe side by rejecting all certificates if they cannot download a recent copy of the revocation list, this creates an even worse problem: an attacker who successfully mounts a sustained DoS attack on the CA's revocation servers may be able to successfully deny service to all users of the network.

  Today, systems that use revocation lists typically ignore these denial-of-service risks and hope for the best.

13.7. Web of Trust

Another approach is the so-called *web of trust*, which was pioneered by PGP, a software package for email encryption. The idea is to democratize the process of public key verification so that it does not rely upon any single central trusted authority. In this approach, each person can issue certificates for their friends, colleagues, and others whom they know.

Suppose Alice wants to contact Doug, but she doesn't know Doug. In the simplest case, if she can find someone she knows and trusts who has issued Doug a certificate, then she has a certificate for Doug, and everything is easy.

If that doesn't work, things get more interesting. Suppose Alice knows and trusts Bob, who has issued a certificate to Carol, who has in turn issued a certificate to Doug. In this case, PGP will use this certificate chain to identify Doug's public key.

In the latter scenario, is this a reasonable way for Alice to securely obtain a copy of Doug's public key? It's hard to say. For example, Bob might have carefully checked Carol's identity before issuing her a certificate, but that doesn't necessarily indicate how careful or honest Carol will be in signing other people's keys. In other words, Bob's signature on the certificate for Carol might attest to Carol's *identity*, but not necessarily her honesty, integrity, or competence. If Carol is sloppy or malicious, she might sign a certificate that purports to identify Doug's public key, but actually contains some imposter's public key instead of Doug's public key. That would be bad.

This example illustrates two challenges:

- *Trust isn't transitive.* Just because Alice trusts Bob, and Bob trusts Carol, it doesn't necessarily follow that Alice trusts Carol. (More precisely: Alice might consider Bob trustworthy, and Bob might consider Carol trustworthy, but Alice might not consider Carol trustworthy.)
- *Trust isn't absolute.* We often trust a person for a specific purpose, without necessarily placing absolute trust in them. To quote one security expert: "I trust my bank with my money but not with my children; I trust my relatives with my children but not with my money." Similarly, Alice might trust that Bob will not deliberately act with malicious intent, but it's another question whether Alice trusts Bob to very diligently check the identity of everyone whose certificate he signs; and it's yet another question entirely whether Alice trusts Bob to have good judgement about whether third parties are trustworthy.

The web-of-trust model doesn't capture these two facets of human behavior very well.

The PGP software takes the web of trust a bit further. PGP certificate servers store these certificates and make it easier to find an intermediary who can help you in this way. PGP then tries to find *multiple* paths from the sender to the recipient. The idea is that the more paths we find, and the shorter they are, the greater the trust we can have in the resulting public key. It's not clear, however, whether there is any principled basis for this theory, or whether this really addresses the issues raised above.

One criticism of the web-of-trust approach is that, empirically, many users find it hard to understand. Most users are not experts in cryptography, and it remains to be seen whether the web of trust can be made to work well for non-experts. To date, the track record has not been one of strong success. Even in the security community, it is only partially used—not due to lack of understanding, but due to usability hurdles, including lack of integration into mainstream tools such as mail readers.

13.8. Leap-of-Faith Authentication

Another approach to managing keys is exemplified by SSH. The first time that you use SSH to connect to a server you've never connected to before, your SSH client asks the server for its public key, the server responds in the clear, and the client takes a "leap of faith" and trustingly accepts whatever public key it receives.[1] The client remembers the public key it received from this server. When the client later connects to the same server, it uses the same public key that it obtained during the first interaction.

This is known as *leap-of-faith authentication*[2] because the client just takes it on faith that there is no man-in-the-middle attacker the first time it connects to the server. It has also sometimes been called *key*

[1] The client generally asks the user to confirm the trust decision, but users almost always ok the leap-of-faith.

[2] Another term is TOFU = Trust On First Use.

continuity management, because the approach is to ensure that the public key associated with any particular server remains unchanged over a long time period.

What do you think of this approach?

- A rigorous cryptographer might say: this is totally insecure, because an attacker could just mount a MITM attack on the first interaction between the client and server.
- A pragmatist might say: that's true, but it still prevents many kinds of attacks. It prevents passive eavesdropping. Also, it defends against any attacker who wasn't present during the first interaction, and that's a significant gain.
- A user might say: this is easy to use. Users don't need to understand anything about public keys, key management, digital certificates or other cryptographic concepts. Instead, the SSH client takes care of security for them, without their involvement. The security is invisible and automatic.

Key continuity management exemplifies several design principles for "usable security". One principle is that "there should be only one mode of operation, and it should be secure." In other words, users should not have to configure their software specially to be secure. Also, users should not have to take an explicit step to enable security protections; the security should be ever-present and enabled automatically, in all cases. Arguably, users should not even have the power to disable the security protections, because that opens up the risk of social engineering attacks, where the attacker tries to persuade the user to turn off the cryptography.

Another design principle: "Users shouldn't have to understand cryptography to use the system securely." While it's reasonable to ask the designers of the system to understand cryptographic concepts, it is not reasonable to expect users to know anything about cryptography.