CS 161 Computer Security
Spring 2010 Paxson/Wagner

Notes 3/8

Key Management

In this lecture, we'll talk about how to manage keys. For instance, how does Alice find out Bob's public key? Does it matter?

1 Cryptographic Hash Functions

See the 3/5 notes for material on cryptographic hash functions.

2 Man-in-the-middle Attacks

Suppose Alice wants to communicate security 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 what his public key is, 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 strategy is insecure. An active attacker 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 the attacker's public key, not Bob's public key. When Alice transmits the resulting ciphertext over the insecure communication channel, the attacker can observe the ciphertext, decrypt it with his (the attacker's) 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, the attacker could re-encrypt Alice's message under Bob's public key and tamper with Alice's packet to replace her ciphertext with the new ciphertext generated by the attacker. In this way, neither Alice nor Bob would have any idea that something has gone wrong. This allows an active attacker to spy on 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 an attacker could mount a similar attack on both directions. As a result, the attacker 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" attack, because the attacker interposes himself in 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 man-in-the-middle 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.

3 Trusted Directory Service

One natural-seeming approach to this key management problem is to use a trusted directory service: some organization who 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 with 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 getting 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 from 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:* This 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. Also, it had better be able to answer them with very little latency, otherwise everyone's communications will be negatively affected.
- *Reliability:* The directory service becomes a single central point of failure. If it is down or unavailable or not responding, then no one on the Internet can communicate with anyone not known to them. Moreover, this is a single point of vulnerability to denial-of-service attacks: if an attacker manages to mount a successful DDoS attack on the directory service, the effect of that will be felt across the world.
- *Online:* Users will not be able to use this service while they are disconnected. If I'm composing an email on the plane, and I want to encrypt it to Bob, my email client will not be able to look up Bob's public key and encrypt the email until my plane lands. As another example, suppose Bob and I are meeting in person in the same room, and I want to use my phone to beam a file to Bob over infrared or Bluetooth. If I don't have general Internet connectivity, I'm out of luck: I cannot 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 generally used in practice. 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.

4 Digital Certificates

Digital certificates were invented by a MIT student as part of his undergraduate thesis[1]. They 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, Arnold Schwarzenegger, might control a private key that is used to sign statements about the public key associated to each public key. For instance, Arnold could sign a statement attesting that "David Wagner's public key is `0x092...3F`", signed using the private key that Arnold controls.

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

$$\{\text{David Wagner's public key is } \texttt{0x092...3F}\}_{K^{-1}_{\text{Arnold}}}$$

where here $\{M\}_{K^{-1}}$ denotes a digital signature on the message M using the private key K^{-1}. In this case, K^{-1}_{Arnold} is Arnold Schwarzenegger'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 Arnold's public key, she can verify the signature on David's digital certificate. If Alice also considers Arnold 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 communicate securely 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, 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 Arnold for these purposes. The certificate is, in some sense, self-validating.

5 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 find out Bob's public key and securely communicate with him. For instance, in the example of the previous section, Arnold Schwarzenegger 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 applications 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.

[1] Loren Kohnfelder, "Towards a Practical Public-Key Cryptosystem", May 1978.

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 who 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. For instance, Verisign, one well-known CA, has an annual revenue around over $1 billion/year. They are a thriving business.

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 a `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 Edit / Preferences / Advanced / Encryption / View Certificates / Authorities. Firefox ships with about 75 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? I don't know about you, but I've never heard of many of these CAs, and I have no clue whether they are trustworthy.

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 who they trust. However, that's not how web browsers work today. Your web browser will accept any certificate issued by any of these 75 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 75 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. Some people criticize the CA model for these reasons.

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. In January 2001, Verisign issued bogus Class 3 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, the certificate has effectively been revoked. However, the problem with short lifetimes is that legitimate parties must frequently contact their CA to get a new certificate; 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 DDoS the CA's revocation servers, to prevent revocation of the bogus certificate. If clients err on the safe side by refusing to 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 DDoS 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.

7 Cryptographic Protocols

Finally, let me introduce some notation for cryptographic protocols. If A is some party to the protocol, we may use K_A to denote A's public key and K_A^{-1} to denote A's private key. We might use K to represent a shared (symmetric) key that is known only to some pair of unspecified parties. Or, if we want to be explicit about who knows the symmetric key, we might write $K_{A,B}$ to represent a symmetric key shared between A and B.

We also introduce special notation for encryption/signing that focuses more on what is signed and what key is used for signing, without dwelling on specifically what cryptographically algorithm is used for this purpose. Roughly speaking, $\{M\}_K$ represents an encryption of the message M under the key K, although the details depend upon what kind of key K is:

- If K is a symmetric key, $\{M\}_K$ represents applying both encryption and a MAC to M. This can be assumed to provide both confidentiality and integrity protection.

  (For instance, a typical instantiation might be to send $C, \text{MAC}_{K_1}(C)$ where $C = \text{Encrypt}_{K_0}(M)$ and $K = (K_0, K_1)$. But the focus is typically not on the specifics of the algorithms used.)

- If K_A is a public key, $\{M\}_{K_A}$ represents the encryption of message M under the public key K_A.

- If K_A^{-1} is a private key, $\{M\}_{K_A^{-1}}$ represents a signature on the message M under the private key K_A^{-1}. Typically we assume that this includes both the message M and a signature on M: anyone who receives $\{M\}_{K_A^{-1}}$ can recover M and also can (if they know the public key K_A) verify the validity of the signature.

  (For instance, a typical instantiation might be to send $M, \text{Sign}_{K_A^{-1}}(M)$.)

Please note: I am not defending this notation. This is not the notation that I would invent, were I crowned king of the (cryptographic) world. But this notation is a loose standard among the cryptographic community, and I've decided to teach you the accepted standard so that you can understand protocols written by other cryptographers. Expect to see this notation again in this course.

If a cryptographic protocol calls for Alice to send an encrypted message to Bob as the first message of the protocol, we might write it like this:

$$1. \quad A \rightarrow B: \quad \{M\}_{K_B}$$

The "1." indicates that this is the first message in the protocol. The "$A \rightarrow B$" indicates that the protocol designer intended this message to be sent by A to B (however, these identities are not present in the data sent over the network, so it may well be possible for an attacker to spoof this message). Finally, everything after the colon (":") is the data that is included in payload of the packet sent to B.