CS 161 Computer Security
Spring 2010 Paxson/Wagner **Notes 3/10**

More on Key Management

These notes go into a little more detail on Public Key Infrastructure (PKI) and other approaches to key management that we didn't have time to cover in the previous lecture.

1 Certificate Chains and Hierarchical PKI

Last lecture, I mentioned as an example that Arnold Schwarzenegger could sign a certificate attesting to the public key of each California state employee. However, in practice that may not be realistic. There are over 200,000 California state employees, and Arnold couldn't possibly know every one of them personally. Even if Arnold 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. Arnold might issue certificates to the heads of each of the major state agencies. For instance, Arnold might issue a certificate for the University of California, delegating to UC President Mark Yudof the responsibility and authority to issue certificates to UC employees. Yudof might sign certificates for all UC employees. We get:

$$\{\text{The University of California's public key is } K_{\text{Yudof}}\}_{K^{-1}_{\text{Arnold}}}, \{\text{David Wagner's public key is } K_{\text{daw}}\}_{K^{-1}_{\text{Yudof}}}$$

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 Yudof to personally sign the certificates of all UC employees. We can imagine more elaborate and scalable scenarios. Arnold might issue a certificate for UC to Mark Yudof; Yudof might issue a certificate for UC Berkeley to UCB Chancellor Robert Birgeneau; Birgeneau might issue a certificate for the UCB EECS department to EECS Chair Stuart Russell; and Russell 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, Arnold acts as a Certificate Authority (CA) who is the authoritative source of information about the public key of each state agency; Yudof serves as a CA who manages the association between UC campuses and public keys; Birgeneau serves as a CA who is authoritative regarding the public key of each UCB department; and so on. Put another way, Arnold delegates the power to issue certificates for UC employees to Yudof; Yudof further sub-delegates this power, authorizing Birgeneau 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 often is chosen to reflect organizational structures.

2 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 who 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 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. I know a security expert who likes to say: "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 facts 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—or at least, that's the theory. It's not clear whether there is any principled basis behind this, 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.

3 Key Continuity Management

Another approach 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. 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 during the first interaction.

This is known as *leap-of-faith authentication* 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 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 hard-core cryptographer might say: this is totally insecure, because an attacker could just mount a man-in-the-middle 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[1].

Another design principle: "Users shouldn't have to understand cryptography to use the system securely." While it may be reasonable to ask the designers of the system to understand cryptographic concepts, it is not reasonable to expect users to know anything about cryptography. We'll see more about these design principles later in the course, when we look at human factors and usable security.

[1] I'll share with you one story that may be apocryphal but illustrates the concept. Several decades ago, a large company in the UK installed a line encryptor to encrypt all of their external communications. Shortly thereafter, they found that the bit-error rate on their communication link suddenly shot way up, rendering the link unusable. If they removed the encryptor, the bit-error rate soon went back to normal. The paranoid explanation: an intelligence agency didn't like losing the ability to eavesdrop on the traffic, so they made sure to jam the communications at a low level (just enough to increase the bit error rate by a large factor) whenever the company turned on encryption. Company employees decided that the encryptor was flaky, removed it, and went back to communicating in cleartext. Today, many phishing and malware attacks on the web are based on social engineering: fooling the user into ignoring security alerts or bypassing security checks that were intended to protect the user.

4 Key Exchange with a Trusted Third Party

If everybody trusts Trent, Alice can contact Trent and ask him to generate a fresh new session key that she can use to communicate with Bob, and ask Trent to securely share this with her and with Bob.

The basic idea seems fairly simple. Let's look at how to instantiate this, in detail. In a highly influential paper written in 1978, Roger Needham and Michael Schroeder proposed a specific cryptographic protocol to do this. The next section describes the protocol.

5 The Needham-Schroeder Protocol

Here is the Needham-Schroeder protocol. Assume Alice has a symmetric key K_A that has been shared with Trent and Bob has a key K_B shared with Trent.

1. $A \rightarrow T$: $\{$I want to talk to Bob$\}_{K_A}$
2. $T \rightarrow A$: $\{$Use session key k, and send Bob this: $\{$This is Alice; let's use session key $k\}_{K_B}\}_{K_A}$
3. $A \rightarrow B$: $\{$This is Alice; let's use session key $k\}_{K_B}$

Now Alice and Bob can communicate securely using the key k. For instance, if Alice wanted to send the message "Launch the missiles!" to Bob, she might follow the above three steps by this message:

4. $A \rightarrow B$: $\{$Launch the missiles!$\}_k$

Unfortunately, the Needham-Schroeder protocol has a security vulnerability: it is vulnerable to replay attacks. A bad guy who eavesdrops on the messages above can later replay messages 3 and 4 to Bob.

3. Bad Guy $\rightarrow B$: $\{$This is Alice; let's use session key $k\}_{K_B}$
4. Bad Guy $\rightarrow B$: $\{$Launch the missiles!$\}_k$

The result is that Bob will think that Alice wanted him to launch the missiles a second time—even though Alice never authorized this. That's a security breach. This replay attack was first discovered in 1981 by Dorothy Denning and Giovanni Sacco.

One solution is to add nonces that are unique for each session. A simple approach is to add a timestamp:

1. $A \rightarrow T$: $\{TS,$ I want to talk to Bob$\}_{K_A}$
2. $T \rightarrow A$: $\{TS,$ Use session key k, and send Bob : $\{TS,$ This is Alice; let's use session key $k\}_{K_B}\}_{K_A}$
3. $A \rightarrow B$: $\{TS,$ This is Alice; let's use session key $k\}_{K_B}$
4. $A \rightarrow B$: $\{TS,$ Launch the missiles!$\}_k$

Kerberos, a widely used authentication system, is based upon this protocol.

The Needham-Schroeder protocol is a classic example of the subtlety of cryptographic protocols. The original 1978 paper by Needham and Schroeder actually included two protocols: a symmetric-key version (shown above), and a public-key version (not shown here). As mentioned before, Denning and Sacco discovered an attack on the symmetric-key version in 1981. It was not until 1995 that researchers first discovered that the public-key version is also secure: Gavin Lowe found a subtle man-in-the-middle attack. This discovery motivated a great deal of advanced research into the design and analysis of cryptographic protocols.

6 Attacks

See the slides.