CS 161 Spring 2010 Computer Security Paxson/Wagner Discussion 7

March 16, 2010

1. **Key Exchange** Consider the following key exchange protocol:

   Message 1 $A \rightarrow S:$ A, B
   Message 2 $S \rightarrow A:$ CA, CB
   Message 3 $A \rightarrow B:$ $CA, CB, \{\{K_{ab}, T_a\}_{K_a^{-1}}\}_{K_b}$

   In the first message A requests from S the certificates for A and B. In the second message, S returns both certificates. These can be used to verify the public key of A and B. In the final message, A sends to B a symmetric key K_{ab} for use in subsequent communication. A timestamp T_a is included in the message. A knows that only B can read the key because it is encrypted with K_b and B knows that A sent the key because it is signed by K_a^{-1}.

   What is wrong with this protocol? How could you fix it?

   Answer: Once A has established communication with B, B can later (within the duration of the valid period specified by T_a) communicate with a third party C, pretending to be A. C will believe it has a shared key with A. The attack would look like this:

   Message 1 $A \rightarrow S:$ A, B
   Message 2 $S \rightarrow A:$ CA, CB
   Message 3 $A \rightarrow B:$ $CA, CB, \{\{K_{ab}, T_a\}_{K_a^{-1}}\}_{K_b}$
   Message 1' $B \rightarrow S:$ B, C
   Message 2' $S \rightarrow B:$ CB, CC
   Message 3' $B \rightarrow C:$ $CA, CC, \{\{K_{ab}, T_a\}_{K_a^{-1}}\}_{K_c}$

   A fix is to require message 3 of the protocol to explicitly state that key K_{ab} is good for communication between A and B:

   Message 3 $A \rightarrow B:$ $CA, CB, \{\{A, B, K_{ab}, T_a\}_{K_a^{-1}}\}_{K_b}$

   Another issue is that B is trusting A to choose the session key. A may not be so good at generating secure keys.

   This example is taken from Prudent Engineering Practice for Cryptographic Protocols by Martín Abadi and Roger Needham. The paper includes many interesting examples of broken protocols that were at one time thought to be secure.

2. **Timestamps**

   Timestamps are often an integral part of cryptographic protocols. Consider the following protocol for synchronizing a clock with a time server.

   Message 1 $A \rightarrow S$: A, N_a
   Message 2 $S \rightarrow A$: $\{T_s, N_a\}_{K_{as}}$

A sends a message to the timeserver and includes a nonce N_a. The timeserver responds with the current time, T_s, and the nonce, using a shared key previously agreed upon by A and S. If the response arrives in a reasonable amount of time, A will accept T_s as the current time.

How can an active attacker trick A into setting back its clock? What sort of damage can a slow clock cause?

Hint: The protocol doesn't specify the length or randomness of the nonce. What can an attacker do if the nonce is predictable?

Answer: If the nonce is predictable, an attacker M can request the time from the server using a nonce M predicts will be used by A at some future time. Then when A makes its request M can respond with the server's earlier response.

A client with a slow clock can be tricked into accepting old communications that rely on timestamps to indicate freshness. A slow clock may also trick a client into accepting an expired certificate.

This example was also taken from Abadi and Needham's paper.

3. **Another Use for Hash Functions** The traditional Unix system for password authentication works more or less like the following. When a user u initially chooses their password p, a random string s (referred to as the "salt") is selected and the value $r = H(p\|s)$ is computed, where H is a cryptographic hash function. The tuple (u, s, r) is then added to the file `/etc/passwd`. When some user later attempts to log in by typing a username u' and password p', the system looks for a matching entry (u', s', r') in `/etc/passwd` and checks that $H(p'\|s') = r'$.

   (a) In this system, what do you suppose the purpose of the hash function H is? Why not just store (u, p) directly in `/etc/passwd` without computing any hashes? Is there an advantage in terms of security or efficiency?

   (b) Suppose you have three candidate hash functions H_1, H_2, and H_3. Suppose further that H_1 is one-way but not second preimage resistant or collision resistant, H_2 is one-way and second preimage resistant but not collision resistant, and H_3 satisfies all three properties. Which of these hash functions would be a suitable choice for the password hashing system described? Would any fail to gain the security or efficiency advantage described in part (a)?

   (c) What do you suppose the purpose of the "salt" s is? Why not just compute $r = H(p)$ and store (u, r) in `/etc/passwd`?

Answer: Note that modern systems usually keep the hash in a separate file, `/etc/shadow`, which has its permissions set to prevent ordinary users from reading it (unless they have physical access to the machine, in which case they can of course read anything). This detail isn't really relevant to this problem, though.

Also, while the Unix password system is effectively the hashing procedure described above, this fact is somewhat obscured by the unusual choice of "hash function": a DES variant keyed on the password is applied to the salt. The hashing process is also confusingly referred to as "encryption", e.g., in crypt(3). I believe the phrase "irreversible encryption" was sometimes used to refer to cryptographic hash functions a long time ago; that may be the origin of this odd terminology.

(a) The purpose is to prevent someone who can read `/etc/passwd` from discovering all the user passwords, while still allowing a typed password to be checked against that file. This is a security advantage, since a user may have chosen the same password on another system, among other reasons.

(b) Any of the three would be fine – the hash function need only be one-way. To be able to impersonate a user after looking at `/etc/passwd` (or `/etc/shadow`), an attacker would have to find a password they can type that hashes to the stored value. This is the situation described by the one-way property. Second preimage resistance would mean the attacker can't do this even if they see the original password, which isn't relevant to our threat model (if they already know one working password, they can just use that). Collision resistance is similarly unnecessary.

(c) Under the described scheme, the best way for an attacker to find out a password based on the hash is to try hashing guesses one after another (a dictionary attack). If no salt was included, this could be done more efficiently across many systems by building a big, static database of hashed candidate passwords and checking the contents of various `/etc/passwd` files against it. With salt, an attacker is forced to try hashing each password guess all over again for each account they want to crack. Salt also prevents `/etc/passwd` files from revealing when users choose the same password on multiple systems.

4. El Gamal and Chosen Ciphertext Attacks The lecture notes explain El Gamal encryption as follows. The public parameters are a large prime p and an integer g such that $1 < g < p-1$; these values are known to everyone. To generate a key, Bob chooses a random value b (satisfying $0 \leq b \leq p-2$) and computes $B = g^b \bmod p$. Bob's public key is B, and his private key is b. If Alice has a message m (in the range $1 \ldots p-1$) for Bob that she wants to encrypt, she picks a random value r (in the range $0 \ldots p-2$) and forms the ciphertext $(g^r \bmod p, m \cdot B^r \bmod p)$. To decrypt a ciphertext (R,S), Bob computes $R^{-b} \cdot S \bmod p = m$.

(a) Suppose you intercept two ciphertexts (R_1,S_1) and (R_2,S_2) that Alice has encrypted for Bob. Assume they are encryptions of some unknown messages m_1 and m_2, and that you have Bob's public key (but not his private key). Show how you can construct a ciphertext which is a valid El Gamal encryption of the message $m_1 \cdot m_2 \bmod p$.

(b) Show how the above property of El Gamal leads to a chosen ciphertext attack.

That is, assume you are given an El Gamal public key B and a ciphertext (R,S) which is an encryption of some unknown message m and that you are furthermore given access to an oracle that will decrypt any ciphertext other than (R,S). Based on these things, compute m.

Answer:

(a) The ciphertext may be constructed as follows, where all computations are done modulo p.

We have that $R_1 = g^{r_1}$, $R_2 = g^{r_2}$, $S_1 = m_1 \cdot B^{r_1}$, and $S_2 = m_2 \cdot B^{r_2}$ for some r_1, r_2. Define $r_3 = r_1 + r_2$ and compute the following.

$$R_3 = R_1 \cdot R_2 = g^{r_1+r_2} = g^{r_3}$$
$$S_3 = S_1 \cdot S_2 = m_1 \cdot m_2 \cdot B^{r_1+r_2} = m_1 \cdot m_2 \cdot B^{r_3}$$

So (R_3,S_3) is a valid encryption of $m_1 \cdot m_2$.

(b) (Again we implicitly assume computation modulo p.) Since (R,S) is encryption of m there exists an r such that $(R,S) = (g^r, m \cdot B^r)$.

Pick any $m' \neq 1$ and any $r' \neq 0$ and compute

$$R' = R \cdot g^{r'} = g^{r+r'}$$
$$S' = S \cdot m' \cdot B^{r'} = m \cdot m' \cdot B^{r+r'}$$

Submit (R',S') to the oracle for decryption. Note that (R',S') is a valid encryption of $m \cdot m'$ and $(R',S') \neq (R,S)$, so the oracle will give us $m \cdot m'$ as the result. Given $m \cdot m'$, we may simple multiply by m'^{-1} to obtain m.