Paxson
Spring 2013

CS 161
Computer Security

Discussion 9

March 20, 2013

Question 1 ***Another Use for Hash Functions*** **(15 min)**

The traditional Unix system for password authentication works more or less like the following. When a user u initially chooses a password p, a random string s (referred to as the "salt") is selected (and kept secret) and the value $r = H(p \mid\mid s)$ is computed, where H is a cryptographic hash function. The tuple (u, r, s) 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', r', s) in `/etc/passwd` and checks that $H(p' \mid\mid s) = r'$.

(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?

> **Solution:** 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.
>
> Even if an attacker must obtain privileges to read the passwords (and therefore could directly access user accounts, or change their passwords), there's still a security benefit, which is that the users may have chosen the same password on other systems. Preventing recovery of the passwords from the password file will then protect those other systems.

(b) 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`?

> **Solution:** Under the described scheme, the best way for an attacker to find 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.

(c) Three key properties of *cryptographic hash functions* are:

- **One-way:** The hash function can be computed efficiently: Given x, it is easy to compute $H(x)$. However, given a hash y, it is infeasible to find any input x' such that $y = H(x')$. (This property is also known as "**preimage resistant**.")

- **Second preimage resistant:** Given a message x, it is infeasible to find another message x' such that $x' \neq x$ but $H(x) = H(x')$. This property is closely related to *preimage resistant*; the difference is that here the adversary also knows a starting point, x, and wishes to tweak it to x' in order to produce the same hash—but cannot.
- **Collision resistant:** It is infeasible to find *any* pair of messages x, x' such that $x' \neq x$ but $H(x) = H(x')$. Again, this property is closely related to the previous ones. Here, the difference is that the adversary can freely choose their starting point, x, potentially designing it especially to enable finding the associated x'—but again cannot.

Suppose you have three candidate hash functions H_1, H_2, and H_3 and that H_1 has property (1), H_2 has properties (1) and (2), and H_3 has all of the above 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)?

Solution: 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`, 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: it does not help to find to two passwords that collide, i.e., hash to the same value. The attacker wants to find a *particular* password for a given hash.

Question 2 ***Timestamps*** **(8 min)**

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

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

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?

Solution: The protocol would not work if N_a were predictable. An attacker M can run the protocol with the server at time T_0 pretending to be A using a nonce N_m, whose value will be used by A at some future point. Then, at some point $T > T_0$, A makes a request to the server to which M responds with the known valid reply $\{T_0, N_m\}_{K_{as}}$.

A client with a slow clock can be tricked into accepting old communications that rely on timestamps to indicate freshness. This type of attack is one form of a *replay attack*. A slow clock may also allow an attacker to trick a client into accepting an expired certificate.

This example was taken from a famous paper by Abadi and Needham [1], discussing design principles for the implementation of cryptographic protocols.

Question 3 ***Perfect Forward Secrecy*** **(8 min)**

Alice (A) and Bob (B) want to communicate using some shared symmetric key encryption scheme. Consider the following key exchange protocols which can be used by A and B to agree upon a shared key, K_{ab}.

RSA-Based Key Exchange Protocol

Message 1	$A \to B$:	$\{K_{ab}\}_{K_B^{pub}}$
	Key exchanged	
Message 2	$A \leftarrow B$:	$\{secret1\}_{K_{ab}}$
Message 3	$A \to B$:	$\{secret2\}_{K_{ab}}$

Diffe-Hellman Key Exchange

Message 1	$A \to B$:	$g^a \mod p$
Message 2	$A \leftarrow B$:	$g^b \mod p$
	Key exchanged	
	$K_{ab} = g^{ab} \mod p$	
Message 3	$A \leftarrow B$:	$\{secret1\}_{K_{ab}}$
Message 4	$A \to B$:	$\{secret2\}_{K_{ab}}$

Some additional details:

- K_B^{pub} is Bob's long-lived public key.
- All messages are destroyed upon receipt.
- K_{ab} and DH exponents a and b are destroyed once all messages are sent.

Eavesdropper Eve records all communications between Alice and Bob, but is unable to decrypt them. At some point in the future Eve is lucky and manages to compromise Bob's computer.

(a) Is the confidentiality of Alice and Bob's prior RSA-based communication in jeopardy?

> **Solution:** Yes. The compromise of Bob's computer gives Eve access to Bob's private key, allowing Eve to decrypt the traffic she previously recorded that was encrypted using Bob's public key. Once decrypted, she obtains K_{ab}, and can then apply it to decrypt the traffic encrypted using symmetric key encryption.

(b) What about Alice and Bob's Diffe-Hellman-based communication?

> **Solution:** No. Since Alice and Bob destroy the DH exponents a and b after use, and since the key computed from them itself is never transmitted, there is no information present on Bob's computer that Eve can leverage to recover K_{ab}. This means that with Diffie-Hellman key exchanges, later compromises in no way harm the confidentiality of previous communication, even if the ciphertext for that communication was recorded in full. This property is called *Perfect Forward Secrecy*.

References

[1] Martín Abadi and Roger Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Trans. Softw. Eng.*, 22:6–15, January 1996.