| | |
|---|---|
| **6.857 Computer and Network Security** | September 11, 2001 |

Lecture Notes 2 : User Authentication

*Lecturer: Ron Rivest/Kevin Fu* *Scribe: Cadar/Frâncu/Gheorghioiu*

## Outline

- User Authentication Overview
- Passwords
- Hash Functions
- MACs
- Cookies

# 1 User Authentication Overview

One important aspect of security is user authentication. The user needs to identify himself as being who he claims to be. Several techniques exist:

1. Something the user is – biometrics. Does not work in remote systems.
2. Something the user knows – a password.
3. Something the user has – a token or device as in Figure 1.
4. Where the user is – does the user have access to a specific terminal?

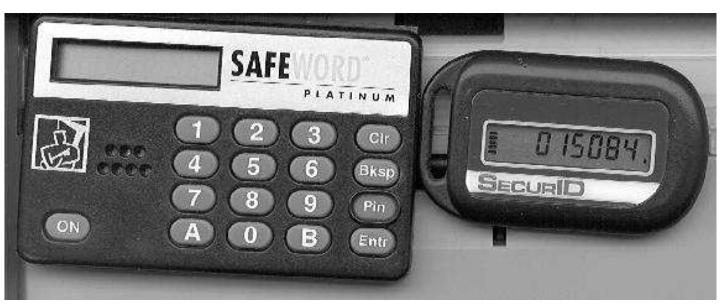


Figure 1: Hardware tokens for user authentication.

## 2 Passwords

In a typical authentication by password, the user supplies a user name and a password (e.g., *"Alice"*, *pw*) to the system. The system then looks up the user name in a database of (name, password) pairs and checks that the two passwords match.

The problems with this scheme are:

- An attacker can eavesdrop or wiretap the telephone line. This problem is often solved by encryption (e.g., an SSL session).
- One can attack the database of passwords and attempt to observe (or even change) Alice's password in the database.
- Session hijacking: an attacker hijacks the connection after Alice logs in, disconnects Alice and begins talking to the server in the name of Alice.
- Users risk losing their passwords. We can cope with this by changing passwords frequently or by using hardware password generators. Password generators can produce a new password every minute using a master secret. The server side stores a database of (user, master secret) pairs. There is still the risk of an attack on the database.

## 3 Hash Functions

A database of plaintext passwords is very vulnerable. We can protect it by "hashing" the passwords in the database. In this scheme, the user sends a pair (user, $pw$) to the system, but the database stores a pair (user, $h(pw)$). There are a variety of names for the function $h$, the most common being **hash function**.

A hash function must be a **one-way function**, meaning it is:

1. Public.
2. Easy to compute.
3. Hard to invert, i.e. given $y$, it is hard to find any $x$ such that $h(x) = y$.

Examples: MD4, MD5 (due to Prof. Rivest), SHA-1 (due to the NSA). SHA-1 maps any string to a 160-bit string, so we can describe it as $h : \{0,1\}^* \rightarrow \{0,1\}^{160}$.

**Question:** Don't we want hash functions to be injective?

**Answer:** First, that would be impossible because there are infinitely many inputs to $h$, but only $2^{160}$ possible outputs (for SHA-1). That is not a problem, however; we are concerned that somebody, given only $h(pw)$, might find an input that hashes to $h(pw)$. We are not necessarily concerned that Alice might be able to log in with several different passwords that hash to the same value.

In previous years, we emphasized ways of building such hash functions. This year, we'll concentrate more on the applications where we use them, and on the properties we want them to have.

Ideally, we want $h$ to behave as a **public random oracle**. It is public, since anyone can evaluate $h$ on any input. It should always produce the same output for the same input, but otherwise the output should be random. We can imagine an elf in a black box, flipping coins as in Figure 2. Whenever the elf receives an input he has never seen before, he produces a new random output string, which it stores in a database. If the elf receives an input string which he has seen before, it returns the stored output.

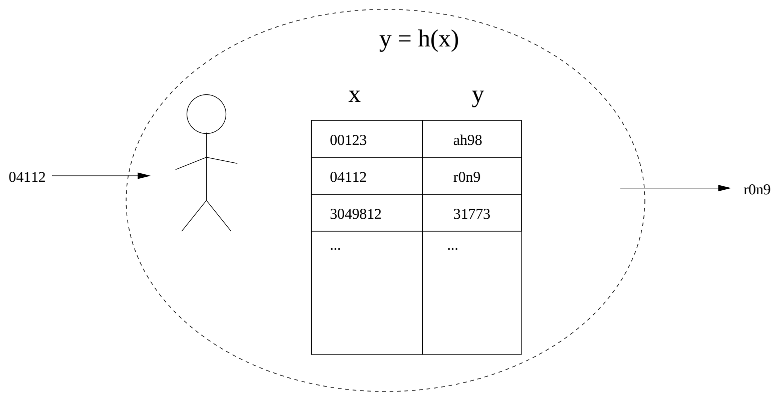


Figure 2: One can think of a random oracle as a box containing an elf that maps inputs to outputs. Given an already asked for input, the elf returns the same answer from the table. Given a new input, the elf flips coins to generate an output and remembers this mapping in the table.

How hard is it to invert $h$? Given $y = h(x)$, the adversary cannot do any better (unless he has some mathematical shortcuts) than to try different values for $x$. So on the average we expect $2^{160}$ attempts. (If an event happens with probability $p$ (e.g., $p = 2^{-160}$), then we expect to have to wait $1/p$ trials on the average before the event happens.)

To get a sense of this, suppose you have $2^{30}$ computer chips (roughly one billion), each doing $2^{30}$ trials per second. The average time to invert $h$ would be $2^{100}$ seconds, or about $10^{22}$ years.

The preceding analysis presumed that Alice's password was a randomly-chosen long password. If so, it would probably easier for an adversary to attack the database and replace Alice's password.

However, people are notoriously poor at choosing good passwords. Hence, password searching programs are very often successful. English words or the name of Alice's dog require many fewer trials to find with a well-organized search. Password cracking programs do exactly this.

# 4 MAC (Message Authentication Code) Functions

Let's go back to session hijacking. If integrity is intrinsic to the communication line (e.g., steel pipes around the wires), then we know the communication is safe. But otherwise, once Alice logs in, how does the server know that she is the author of every subsequent message? This type of authentication is crucial in bank transactions when someone logs in and requests money out of an account.

In the MAC scheme, Alice has a **secret key** $K$ which is known only to herself and the server. For each message she sends to the server, Alice also sends a MAC (message authentication code) of the message computed using the secret key. So Alice sends pairs of the form ($m_1$, $\mathrm{MAC}_K(m_1)$), ($m_2$, $\mathrm{MAC}_K(m_2)$) and so on. The MAC is a tag that proves that the message comes from Alice. (The server recomputes each MAC and compares it to the MAC received.)

## 4.1 Types of attacks on MACs

1. The adversary can try to guess the MAC for a message.
2. Known message attack: The adversary can listen to several messages with their associated MACs, and then try to break the MAC scheme.
3. Chosen message attack: The adversary can see MACs of messages of his choice, and then try to break the MAC scheme.

Ideally, we want MACs to be *unforgeable*. An adversary who does not know $K$ should not, with more than negligible probability, be able to forge a MAC and produce a pair ($m$, $\mathrm{MAC}_K(m)$), even after seeing many valid pairs ($m_i$, $\mathrm{MAC}_K(m_i)$) for messages $m_i$ of the adversary's choice.

The adversary can presumably replay messages. In practice, this needs to be guarded against by using timestamps, sequence numbers, or other techniques. But a replay does not count as a "forgery" in the above definition.

Notice that MACs are not the same as digital signatures, although they are related. A digital signature requires different keys for encryption and decryption, whereas the key for MACs is symmetric (same at both ends).

We can view $\mathrm{MAC}_K(m)$ as a family of random functions. Again, imagine the elf in the black box.

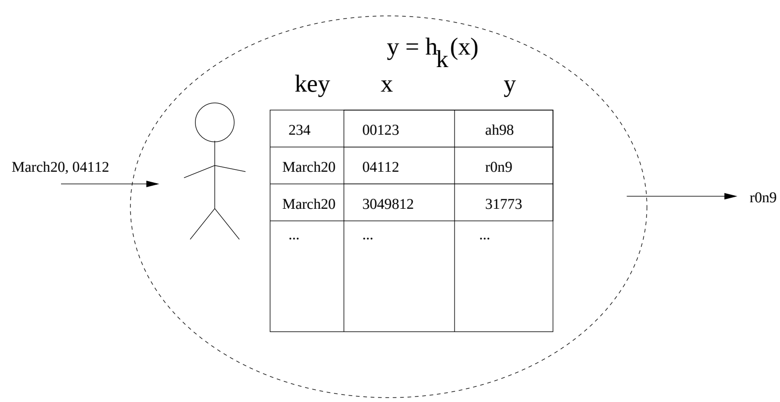


Figure 3: One can model a random oracle for a MAC as an elf again. This time the elf keeps another column for the key to generate a family of random functions.

### 4.2 Possible application: Online Voting

To vote online, Alice sends a pair ($ballot$, $\mathrm{MAC}_K(ballot)$) to the voting server. The main problems are:

1. The server can forge ballots.
2. There is no privacy.

(So we see that we need something stronger here than just MACs.)

## 5 Cookies

Kevin Fu presented slides from a recent USENIX Security talk. See also the reading material distributed in class.

In particular, he focused on how the interrogative adversary can defeat weak cookie-based user authentication schemes. An interrogative adversary can treat a server as an oracle for an adaptive chosen message attack. It can then adaptively query a Web server a reasonable number of times.

One of the recommendations was to properly marshal data before signing or MACing it. That is, sign an unambiguous representation of a message to avoid field shifting attacks. For example:

- badauth = sign (username + expiration, key)
- (Alice, 21-Apr-2001) → sign (Alice21-Apr-2001, key)
- (Alice2, 1-Apr-2001) → sign (Alice21-Apr-2001, key)
- Two different messages have the same signature!
- Use an unambiguous representation or delimiters