Part I

Applications of Hash Functions

1. Applications of Hash Functions

What Passwords are for

Passwords are used to authenticate entities (users), i.e. to confirm that the entity is the rightful owner of the declared identifier.

For each user, the entity performing authentication (the authenticator) stores some piece of information usually in a password file. There are two main threats to such systems:

- an attacker obtains access to the password file and tries to recover one or more passwords
- a bogus authenticator tries to obtain a password from the user

The Human Component

Humans are bad at memorizing passwords. A password made of English words has about 1.5–2 bits of entropy per letter. Therefore, to obtain the same entropy as a 128-bit random string, we need 64 characters, which is unacceptable.

Most real world passwords have fewer than 32 bits of entropy and are included in wordlists or are minor variations of common passwords.
Better results can be obtained with passphrases, resulting in more than 64 bits of entropy.

Also, humans tend to use the same password for several systems.

Hashed Passwords

Obviously storing the passwords in clear in the password file is insecure. So, what about hashing?

Given the password P, the authenticator stores $h(P)$.

The the user tries to log in, the password P' is hashed and $h(P')$ is compared to $h(P)$. If the two match, the user is authenticated.

Problems with hashed passwords

In the basic hased password scheme, for a given password P, there is a unique $h(P)$. Therefore:

- the attacker hashes all the passwords in the dictionary. The attack is successful if any of the stored hashed passwords in in the dictionary;
- the hashing of the dictionary can be precomputed before having access to the passwords file.

Generally, about 50% to 75% of the passwords are found in a few hours. Considering precomputation, the time drops to seconds.

Improvements to the hashed-password scheme

Morris and Thompson (1979) suggest three improvents over the basic scheme:

- salting;
- slower hashing (aka password stretching);
- using less predictable passwords.

Password Salting

Salting consists in appending a random number (the salt) to the password before hashing it for storing. The salt is also stored in the password file.

When the user logs in, the stored salt is concatenated to the password and hashed. If the hash corresponds to the stored hash, the password is accepted.
Salting gives two advantages:

- the password hashes cannot be precomputed, but access to the password file is necessary to start the computation;
- hashing of the dictionary must be repeated for each password, slowing the attack by several orders of magnitude.

Password stretching

Hash functions are designed to be fast. While this is generally useful, it gives the attacker the ability of trying several passwords per second.

The idea of stretching is to repeat the hashing several times. If the password hashing slows from 0.1 ms to 100 ms, the user will not notice a significant delay when logging in, but the attacker will be ablo to test 1000 times fewer passwords per second. Recently, some hash functions have been developed with the intent of being impossible to parallelize.

Password Stretching

Given: P a password, s a salt, and h a cryptographically secure hash function.

function password_hash(P, s)
    $x_0 = 0$
    for $1 \leq i \leq r$ **do**
        $x_i = h(x_{i-1} \| p \| s)$
    end for
    return x_r
end function

The result is stored in the password file.

Using less predictable passwords

Complex phrases with special symbols are less likely to be in a dictionary and, in general, require more time to discover. However, enforcing complexity by imposing rules in the composition of the password results in predictable passwords or increase the likelihood that the user writes it.

1. Applications of Hash Functions

Key Derivation Problem

Suppose you have a secret s and you want to convert it into a key k for some encryption scheme. Examples: a password, a master key, old key, non-uniform random data.
Basically the same as password hashing: pick a hash function h and calculate $k = h(s)$. Include salting and stretching as necessary.
Note that we have the additional requirement that the derived key *looks like random*. Collision resistance does not imply pseudorandomness of the derived key, neither is a necessary property. Therefore it is typical to use special keyed hash functions called Pseudorandom functions.

Password Based Key Derivation Function 2 (PBKDF2)

PBKDF2 is an internet standard algorithm for key derivation. It takes five input parameters:

$$\mathrm{DK} = \mathrm{PBKDF2}(\mathrm{PRF}, \mathrm{Password}, \mathrm{Salt}, c, \mathrm{dkLen})$$

DK	the derived key
PRF	the underlying pseudorandom function
Password	the user-specified password
Salt	the random salt
c	the number of iterations
dkLen	the size of the derived key

Password Based Key Derivation Function 2 (PBKDF2)

Pseudorandom Functions

PKDHF2 supports many PRFs. One popular choice is HMAC-SHA-1. HMAC-SHA-1 has several applications. In this context it is defined as:

$$\text{HMAC} - \text{SHA} - 1(k, m) = h((k \oplus \text{opad}) \| h((k \oplus \text{ipad}) \| m))$$

- h is SHA-1
- k is the password paddedd or hashed to 160 bit
- m is the salt
- opad is the constant 0x5c repeated to 160 bit
- ipad is the constant 0x36 repeated to 160 bit

1. Applications of Hash Functions

Commitments

The goal of a commitment scheme is to temporarily hide a value, but ensure that it cannot be changed later.
Example: sealed bid at an auction

1st stage: commit Sender electronically "locks" a message in a box and sends the box to the Receiver

2nd stage: reveal Sender proves to the Receiver that a certain message is contained in the box

Commitment Schemes

A commitment scheme is a set of two algorithms.

$c = \text{Commit}(m, r)$ takes as input a message m and a secret *nonce* chosen uniformly at random form a large set. It outputs a commitment c.

$\text{Verify}(c, m, r)$ takes as input a commitment c, a message m, and a nonce r. It outputs true if $c = \text{Commit}(m, r)$ and false otherwise.

Properties of Commitment Schemes

hiding: Given a commitment c, an Adversary cannot find m.

binding: The committing entity cannot find two distinct pairs (m, r) and (m', r') such that $m \neq m'$ and $\text{Commit}(m, r) = \text{Commit}(m', r')$.

Hiding can be perfect (if Adversary is unbounded) or computational.
Binding can be perfect (if committing entity is unbounded) or computational.
Technical: a scheme cannot be perfectly hiding and perfectly binding.

Hash-based commitments

Given a hash function h, we define:

$$c = \text{commit}(m, r) = h(r \| m)$$

It is easy to see that a collision-resistant hash function makes a computationally binding commitment scheme.
Unfortunately, collision resistance does not imply any hiding property. We will say that h is hiding if the commitment scheme using h is computationally hiding.

1. Applications of Hash Functions

Search Puzzles

A Search Puzzle consists of:

- a hash function h, with hash size b bits
- a puzzle ID k, chosen uniformly at random from a large set
- a target set Y

A solution to this puzzle is a value x such that $h(k||x) \in Y$.

We say that h is puzzle-friendly if the expected number of evaluations of h required to solve the puzzle is $2^{b-1}/|Y|$.

In practice, if h is puzzle-friendly, the best strategy to solve a puzzle is making several random attempts until success. Note that collision resistance does not imply puzzle friendliness.

1. Applications of Hash Functions

Hash Pointers

A hash pointer is a pointer to a block of data along with a hash of the data. The hash pointer can be used to verify that the data have not changed.
A blockchain is a linked list using hash pointers. In a blockchain each block contains a hash of the previous block, therefore we just need to store a hash pointer of the most recent block to obtain a *tamper-evident log*.

Properties

Any modification in a block can be detected either by looking at the hash pointer stored in the following block, or by looking at the stored hash pointer of the chain head.
Note that a blockchain is very similar to a MD construction and similar security properties hold.
The first block of a blockchain does not contain any hash pointer. It is called *genesis block*.

1. Applications of Hash Functions

Markle Trees

A Merkle Tree is a binary tree using hash pointers. Data blocks are the leaves of the tree. Data blocks are grouped in pairs and the hash of each block is stored in the parent node. In turn, parent nodes are grouped in pairs and their hashes are stored one level up in the tree and so on. The hash pointer to the root is stored.
A basic property of the Merkle tree is that any modification can be detected in a way similar to a blockchain.

Proof of Membership

Suppose that we build a Merkle Tree and publish the hash of the root. Then, we want to prove that a given block is included in the tree.

The blocks along the path from the data block to the root are a proof of membership of the block to the tree. If there are n blocks in the tree, the proof of membership will contain approximately $\log_2 n$ blocks, so it can be verified quickly.