NETWORK SECURITY:

How do servers store passwords?

Servers avoid storing the passwords in plaintext on their servers to avoid possible intruders to gain all their users' passwords.
A hash of each password is stored.

In the event an attacker is able to break into the server and retrieve the list of hashes of password, how does he/she go about retrieving the plaintext passwords?

1) Dictionary:

The attacker can pre-compute a list of <password,hash> pairs and stores it on his/her own machine. This way, when he retrieves the hash list from the server, he can easily find out the passwords.

How much does it cost (in terms of space and tie) to compute this list?

BACK OF ENVELOPE CALCULATIONS

Space:
Let's assume each character in a password is chosen from 72 different characters (26 upper case letters + 26 lower case letters +10 numbers + 10 special characters) and we only store passwords that are 6 characters long.
We have 72^6 possible passwords.
If we use SHA1 then it's a 160 bits for each password --> 72^6 * 160 bits = 72^6 * (160 /8) / (1024*1024*1024) = 2594 GB ~ 2 TB ($100 hard disk on the market)

Time:
How long does it take to compute this list?
~ 10 micro seconds to compute each hash => 72^6*10 / [(10000000) * (60 * 60 * 24)] = 16 days

Question: without dictionary, how long would it take to find the password for the hash? average case takes half the time --> 8 days.

Now that we have a list of hashes (ordered by password ID), how long does it take to find the match between retrieved hash and <password,hash>?

Assumption: upper bound of contiguous bulk reads from disk is 100MB/s

Naïve linear scan: if we were to scan naively this list => 2594 GB / [100/1024] = 26000 seconds / [60*60]--> 7.3 hours.
The average case is again half of that.

A better approach is to store hashes ordered by value.
In this case we would still need to store the value of the password.
We can compress this by storing the binary representation of the password and not the password per-se.
Therefore each element in this long list has <hash, value in bits of password> pairs. Each pair is 160 bits + lg(72^6) = 198 bits. The space needed for the table is 198*72^6 = 3.2 TB. ($200 hard disk?)

The time complexity though is much better: with a binary search on the list we reduce it to O(log n). It is noteworthy though that each step in the binary search corresponds to a disk seek. Each disk seek is ~ 5-10 ms. Therefore the time complexity for this approach is at max ~ 0.38 seconds.

AS WE CAN SEE WE HAVE A BIG TRADEOFF BETWEEN SPACE AND TIME:

2) Rainbow tables

Another approach that can be followed is by using rainbow tables!

To understand rainbow tables first we have to understand hash chains:

HASH CHAIN
We construct a chain of alternating passwords and hashes using two functions H and R (reduction function). Example:
aaaaaa -H-> 281DAF40 -R-> sgfnyd -H-> 920ECF10 -R-> kiebgt

We store the start and end point (aaaaaa and kiebgt).

To retrieve the plaintext password from the hash, we start applying R and H until we get to one stored endpoint.
Then we take its start-point and we start applying H and R until we get the hash.
The value of the password will be the one right before the H function.
Example:
if hash is 920ECF10 then through R we get kiebgt --> we start from aaaaaa until we get to 920ECF10 again.
We find the password we are looking for is sgfnyd

We might incur though in what is called a false alarm.
Even if we find the endpoint when we reconstruct the chain from the start-point we do not run into the hash value h. This is because the hash value h might be present before the startpoint (our hash chain is not long enough?).

In this case keep on applying H and R to the first hash value until we find another match.

FLAW of hash chains: collisions. There can be chains that at a certain point collide (two different hashes redirect to the same password – function R is not

collision resistant). From that point on the two chains are equal. The problem is that chains are not stored in their entirety so collisions are hard to detect.
Also, if collisions happen at different positions in the chain then we will have different endpoint values.

RAINBOW TABLES:

Rainbow tables make the problem of collisions harder. They use k different reduction functions. This way, in order for two chains to collide and merge, they must hit the same value on the same iteration.

But this also changes the way the lookup is done.
Now we have to generate k different chains.
We make the first chain by using R_k and see if that matches any endpoint.
Otherwise we start with R_{k-1} -H->R_k meanwhile we check if any match the endpoint.
Once we find an endpoint we retrieve its respective startpoint and we start creating the hash chain using R_1.... R_k.

FINDING THE PASSWORD WITHOUT THE RAINBOW TABLE:
- question: what about the birthday paradox? can we apply it in this case? --> NO because we are mapping from a small space to a larger space. The birthday paradox finds two inputs that collide after 1.25* H^(1/2) evaluations. (H is the output space). But in our case we are a looking for a collision for a particular hash value not just an existential collision (between any two inputs). We want to have the pre-image of a specific hash value so we have to just to brute force all of the possible hashes and therefore it's the expectation of 16 days ~ 8 days (on average you expect to go through half of the possible hashes before finding a collision)

Salting or preventing rainbow table attacks

- Instead of storing a hash(password) the server stores hash(salt||password)||salt. Therefore it is harder to calculate rainbow tables. For each salt you have to calculate a rainbow table.

- Another way to look at this is that you have increased the password by salt random characters. Your rainbow table is not 6 characters but 6 + salt characters. (salt is also 255 possible characters not 72). A salt is usually 16 bytes. Your rainbow table size explodes to 6e30 GB.

Site specific passwords:

- a user usually uses one password for multiple sites. This is a problem if anyone of those sites does not salt its passwords. A different solution can be a plugin into your browser that calculates a salted hash of your password before sending it to the remote site. It stores the salt locally on your disk. Usually the browser does not use a random salt but rather the domain name.

```
human --------------> browser -----------------------------------> server
  pwd                   pwd' = hash(domain_name||pwd)      hash(salt||pwd')||salt
```

- The problem is that the attacker can create a rainbow table per domain name. (more feasible than per-salt rainbow table).

- Solution: add a user specific secret as part of the salt:
  pwd' = hash(domain_name||pwd||secret).

- Problem now is that we can't access the sites with a different browser/computer. We can therefore have a trusted third party (on the WEB) which stores the secret.

MALICIOUS CA

- A malicious root CA can create a certificate that is deemed valid for any site on the Internet including sites like Google and Yahoo. Your browser would not have anyway of telling the difference between a valid certificate and an invalid certificate

- list of all the certificate authorities accepted by firefox:
http://spreadsheets.google.com/pub?key=ttwCVzDVuWzZYaDosdU6e3w&single=true&gid=0&output=html

- Problem: root CAs are self-signed. 20 years ago there was only a couple of companies who had the privilege of being root CAs (Verisign). Today our browser, by default, accepts 83 different root authorities.

SDSI / SPKI : Simple Distributed Security Infrastructure / Simple Public Key Infrastructure

Idea: there is no root authorities. You trust your friends and then who your friends trust - distributed social network. Didn't take off in the 90s but now with the advent of social networks it's time for another go.
- every person is its own certificate authority
- we can imagine building a public key infrastructure based on who your friends trust. That's where the concept of social network comes in.

User could pick whoever they want to be their trust providers.

If I want to establish a trusted web site (distribute trust transitively), I bind my website to my public key and I start forwarding this binding to my close friends who, hopefully, will trust me. (authorization certificate)
In the paper the term used is not trust but 'delegation'. Alice issues an authorization certificate to Bob, granting him some authority. Bob then grants Carol some subset of the authority

When we want to connect to a person effectively we want to find a chain of delegations of authority. If we have that chain then we trust the person we are connecting it.

Example:
- Alice wants to connect securely to foo
- foo says 'domain is some PK' SDSI wants to find a chain from Alice-Bob-Charlie-Foo so that we have a chain of keys
- Alice trusts Bob, Bob trusts Charlie, Charlie trusts Foo is that PK

FIRESHEEP

web sites that only encrypt the login process and not the cookie(s) created during the login process.

The extension uses a packet sniffer to intercept unencrypted cookies from certain websites (such as Facebook and Twitter) as the cookies are transmitted over networks, exploiting session hijacking vulnerabilities.

Once you have the unencrypted cookie you can use it to bypass the login when you connect to the site and you have direct access to the user's account.

The problem is even bigger if the site allows a user to change his/her password without asking for the old password. Such unwary websites allow the attacker to steal the account.