17/11/2017

PASSWORDS

Passwords are so common in computer systems, despite their insecurity. The main problem is that humans have not capabilities for remembering long and secure passwords, and in many cases they use passwords containing words belonging to some dictionary (words wich are meaningful), and so they are vulnerable to dictionary attacks, not to mention Trojan Horses attacks that can be performed while the user types the passowrd. In this context, computers are much more secure since they can store long and secure passwords/keys, even encrypted.

Dictionary attack: attacker take a long list of words from a long list of words taken maybe from the web, and they try all possible words. ***Crackstation.net*** allow you to write an hash and they will send you back the password (done by precomputation, lookup tables).

Human beings: you can replace o with 0, the e with 3 and so on, but the attacker knows it!

Eavesdropping: even if you send the password encrypted, the attacker can just store the encrypted password and use it, without knowing the plain password. This is why authentication should be different every time.

Unix password hash

One can think of inventing a password and hashing it, then using the hash as password. You can also store somewhere the hash. So the idea is that the password typed by the user is converted to a key of 56 bit with 8 parity bits added (DES key structure). At the beginning UNIX used this approach, storing only the first 8 characters of the key (when ASCII was the standard, which used 7 bits in order to represent characters). What changed then is that DES was used, and was performed 25 times on a 8 bytes string of zeroes using the key created as mentioned above, so the key stored in the system was $DES_K(DES_K(...DES_K(00000000)))$.

File */etc/passwd* → In this file there is a row for every user, where there are collected useful informations, there are collected also the encrypted passwords. Note that every user can read such file. In the modern Unix implementation the passwd file does not contain anymore the encrypted passwords because it was a weakeness since one could copy that file and do some precomputation of stored encrypted passwords.

Password salting: when a user is storing the password, the system generates a random string, the salt, that is cuncatenated to the password, this is the salted password. Then the salted password will be encrypted. The salt is stored as *clear text*, so the attacker can no longer use dictionary attack because of the salt (actually he can but it requires more computational effort).

What we want is to reach an approach in which users only have to remember their password, without further heavy crypto-informations, and then we want to use cryptography in order to make this approach secure.

Lamport's Hash

Lamport invented the **OneTimePassword**. It works in the following way: Alice is choosing a password, Bob is the server that stores the password. Instead of storing the password, Bob stores the name of Alice, a number *n* like 200 or 300, and the n-times hash of the Alice's password. Then when Alice has to authenticate, she sends Bob the password hashed n-1 times, then Bob computes the hash of what receives and checks whether the result is the same as what Bob stores in the database for Alice. Then when there is a matching, it is okey but Bob now does also another thing, n is decremented by one and he stores in the database the n-1 times hash of the password, so in the database there is no longer the n-times hash. Note that this prevents the *replay attack* since next time Alice will send the n-2 times the hash (the attacker cannot guess the previous hash knowing the next one), and so on.

Then there is also the *salted* version, where the password is cuncatenated with the salt. This approach is good because Alice can always reuse the same password, what is important is to constantly change the salt.

Small n attack

When the Server has to send to Alice the number n, Trudy can impersonate the Server and can send to Alice a n' such that n'<n. So what happens is that Alice sends to Trudy the n'-times hash, Trudy performs the remaining n-n' hashes ans the Server will store the result. So Trudy is able to impersonate Alice for a number of times equal to n-n', so until the hash reaches the n'-times hashed value.

Encrypted Key Exchange (EKE)

This is an *authentication* approach that is deeply based on Diffie Hellman. It is assumed that the two parties share a secret W, which is a function that only the two parties know. Now what happens is that Alice chooses a secret *a* and sends to Bob $W(g^a \bmod p)$. Bob is able to decrypt because know the secret W, then will choose a secret *b* and will send to Alice $W(g^b \bmod p, C_1)$ where C_1 is a challenge. In the meanwhile Bob chooses the key as the DH protocol states. Alice will be now able to generate the key too, and will send to Bob $K\{C_1,C_2\}$. When Bob will reply with C_2, both the parties will authenticate. Note that this solution solves the *man-in-the-middle* weakness of Diffie Hellman. There is also a variant, called SPEKE, which states to transfer $W^a \bmod p$ and $W^b \bmod p$ and choose as key $K=W^{ab} \bmod p$.