Hash Functions for use in Crypto

Friday, November 20, 2020 10:45 AM

What is a hash function?

It's a many to one function that takes as input an string of arbitrary length and outputs a fixed length output. The computation should be relatively fast (on a computer).

A good hash function should have the following properties (intuitively):

1. Each output value should occur roughly equally.
2. Small changes in the input should produce fairly large changes in the output (ie random).
3. Given a function output, it should be infeasible to construct an input in a reasonable amount of time that produces that output.
4. It should be infeasible in a reasonable amount of time to create any two different inputs that create the same output.

In particular, hash functions are commonly used for the three following purposes:

1. Message Authentication (making sure the message wasn't tampered with)
2. Digital Signatures (making sure that the person who claims to have sent the message is really the person who sent it)
3. Checking if passwords entered are correct

Four methods for Message Authentication

1. Source sends M || E(K, H(M)) - here we don't care that an adversary is reading the message, but we want to know that the message hasn't been tampered with.

2. Source sends M || H(M || S). Again, the message isn't encrypted here, but any modification can be detected. S is a secret value shared by the Source and Destination.

3. Source sends E(K, [M || H(M)]. Receiver Decrypts. Then takes the M and hashes it, to see if it equals the claimed H(M). If

not, someone altered the encrypted message.

4. Source sends E(K, [M || H(M || S)]) to Destination. Same as two but with message encryption.

Most commonly message authentication is achieved via Message Authentication Codes (MACs), where the two communicators share a secret key, K, and E(K, H(M)) is sent in addition to the encrypted message.

Two methods for Digital Signatures

1. Source (with private key **PR_a)** sends M || $E(PR_a. H(M))$ to Destination.
2. Source sends E(K, [M || $E(PR_a. H(M))$]) to Destination.

Only difference between 1 and 2 is that the message is encrypted in 2.

If someone encrypts with their private key, then I can decrypt with the public key. So anyone can verify the digital signature, not just me. But, if the public key really does decrypt was it sent, then we have proof that the only person who could have sent it is the person who has the private key of A, which is user A.

So, we could perfect verify everything in the situation if user A encrypts with their private key, followed by encrypting that with user B's public key: $E(PU_B, E(PR_a, M))$, then B is going to user their private key to undo the public key encryption and only B can do this. Then, B will decrypt what's left using PU_a, and that proves that A must have been the person who sent the message. **The ONLY reason we don't do this is that it's slow…it requires slow public key encryption…two times, for the duration of the message…**

Why must we use a private key from a public key scheme for a digital signature instead of a shared private key…

Answer: Say you and I share a private key. I could fake that a message came from you to me since I know our shared key…

Storing Passwords

It would be really bad if a server stored all users' passwords in

plaintext, and a hacker got into the server and got that plaintext.

To avoid this, servers don't store user's passwords. Instead, they store the hash value of a user's password.

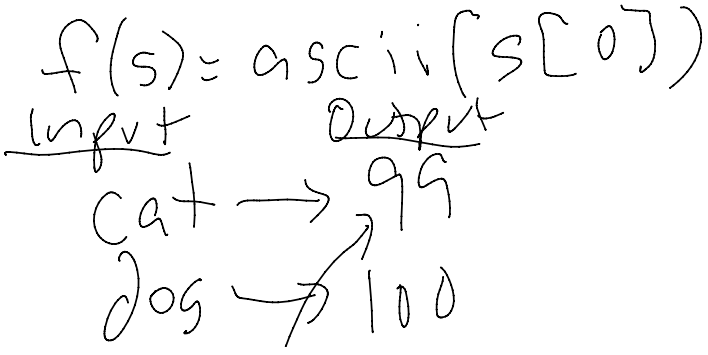
So, when you enter your password, it gets hashed. Then, this hashed value is compared against the hashed value of the correct password. If there is a match, you are authenticated. So, in theory, if someone figured out a different input with the same hash function output, they could get into your account, but for a good hash function, this is hard to do.

Thus, now, if someone steals the file on the server, they just have the hash values and not the actual passwords. They would have to try to get input values that match to those output values.

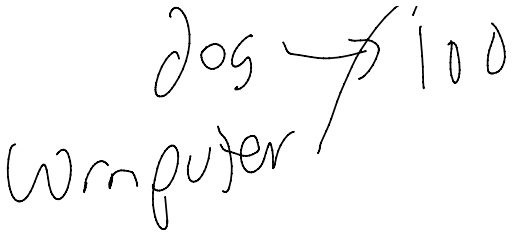
There is one weakness here: many people choose easy to guess passwords, like people's names, or words, or pets, or important dates. What someone could do, is get a large collection of these things and calculate the hash value, and then create a reverse look up table for each "common potential password". Namely, if any hash value on the table matches a list of stolen hash values, then they can get into those accounts.

To mitigate this possibility, we can salt passwords. The way this is done is as follows. On the server end, for each password, we generate a random string. This random string is added (either at the beginning or end of the password). We add a different random string for each password. After this, we calculate the hash of this new string and store this hash value AND the salt used for it.

Now, if an adversary steals the table, they don't have a global look up chart. Instead, for each salt value, they have to add it to each of their "common passwords", which reduces the efficacy of the hack and the amount of time and space the hack would have to take to compromise the same number of passwords.



This is bad, because if I gave you the output 99, you would just pick any word that starts with 'c' and you would have



output 99, you would just pick any word that starts with 'c' and you would have found an input that maps to the output. Also, it would be very easy for you to pick two strings that start with lowercase c…

User Password (Example of storing passwords in plaintext --> really bad)

Bob myangel123
Alice nothingtoseeHear
Eve Christmas

Idea #1 : Store the hash of each password

User	Hashed Password

Bob 234hjadg31 (H(myangel123))
Alice 112bai37bd (H(nothingtoseeHear))
Eve p8shqevhdf (H(Christmas))
Hacker has to figure out H(?) = 234hjadg31
PROBLEM!!!
Maybe our hacker has guesses what common passwords might be…
Say they have a file of a million strings…they precompute each hash value of what they think might be common passwords
Bday1234 9dahgfdaf
Angel kidfdfdfdf
Myangel123 234hjadg31
It's easy enough to store this table backwards: (Rainbow table)
9dahgfdaf → Bday1234
Kidfdfdfdf → Angel
234hjadg31 → myangel123

Next idea: Salt a password:
Bob's password is myangel123…now, when I go to store it for the first time, I also generate a random string, let's say e8cz0o. Now, add this string to the password and calculate its hash
H(myangel123e8cz0o) = 8rgyxtndfu
Now, in my table on the server, I store:

User	Hashcode	Salt
Bob	8rgyxtndfu	e8cz0o
Alice	892htdadbe	km451q

Alice nothingtoseeHear add the sale km451q calculate
H(nothingtoseeHearkm451q) = 892htdadbe

Question: why is this better?

Answer: Now, my adversary can't just precompute H(myangel123) and H(nothingtoseeHear)…they have to find each salt, and add that to each precomputed word, and even if they did this for one salt, their lookup table would be invalid for all but one of the passwords. So, now, they would have to make up a new lookup table for each user, which goes against why the technique was so valuable to begin with.

Before 1000 users, store one table of 1000000 hashed values…maybe 50 users have passwords that I have in my table…

Now, 1000 users, let's say I make a table with one of the salt values…I have a 1/20 chance of obtaining that ONE password.