Password Hashing

Security

Otago Polytechnic
Dunedin, New Zealand

The problem

- It's clear that we need to encrypt users' passwords.
- We can use a one way hashing function to do this, since we don't ever need to decrypt it.
- Instead we hash the supplied password in the same way that we hashed the stored password
- If the two hashes match, then we authenticate the user.

Naive hashing is not enough

- If we have a database of hashed passwords, it's still a bit too easy to break.
- This is especially true if our users have bad passwords.

It's too easy to find passwords

- Suppose we want to see if anyone is using the password "password".
- We hash it and find that it hashes to `E201065D0554652615...5BC8EDCA469D72C2790E24152D0C1E2B6189`
- Now we scan the password database to look for a match.

The solution: add salt

- We solve this problem by adding *salt* to the hashed password.
- The process for generating a salted password is:
  1. Start with the supplied password, e.g., `foo`.
  2. Generate some random bytes that we can represent as a string, e.g., `salt`.
  3. Prepend the salt string to the password and hash the combined string. Our hashed password is thus

     ```
     hashedpw = hash("saltfoo")
     ```
  4. Now in our password store, we save the username, the salt, and the hashed password.

Verifying the salted passwords

The verification process is:

1. Get a username/password combination to verify.
2. Retrieve the salt and hashed password from your user data store.
3. Hash the supplied password with the salt from the saved password.
4. Check to see that the hashes match.

Conclusion

Do you see why this makes our password database harder to crack?