Password-based user authentication

Niels Jørgensen, Nov 5th, 2025

This note discusses user authentication by means of passwords. As we all know, passwords are widely used for this purpose. The note also discusses the program “PasswordBasedAuthenticator”. The program stores usernames and (hashed) passwords in a password table in a database. The note and the program are intended for teaching at the RUC course “Complex IT Systems” at Computer Science at RUC.

The main discussion concerns cryptocraphic hashing of stored passwords, to counter two challenges:

- we end users have a limited capacity to memorize strong passwords, and
- the system needs to store the (hashed) passwords for comparison when the user logs in.

The preferred solution to protect against guessing or ‘cracking’ of passwords is to store not the password itself, but a hash value of the password. And more specifically, a hash value obtained by applying the hash function iteratively, that is, many times, say a million times. This is to slow down the computation involved in guessing passwords.

The note recommends the following four defensive measures. They are meant for protection against password guessing, in systems that use password-based user authentication.

#1: Passwords must be strong (eg., they must have some minimum length).

#2: Passwords must be hashed (that is, stored using a cryptographic hash function).

#3: Passwords must be ‘saltet’ (when hashed).

#4: The hash function must be iterative, so that attacks are CPU-intensive.

Defenses #1-4 were recommended as far back as in the late 1970s (Morris and Thomsen, 1979). The PasswordBasedAuthenticator implements #2 and #3. Defenses #1 and #4 are left as questions in Assignment 5.

Table of Contents

1. Helicopter perspective: why use passwords?

Passwords are the most widely used means of user authenticaction by online, digital services. Unfortunately, passwords are inherently insecure. This is mainly because a user can only memorize passwords up to a certain length, in particular if the user has to remember a whole set of passwords, perhaps up to 100 or more passwords for different services.

Let's first consider passwords from a helicopter perspective and discuss their purpose and how they may achieve the purpose. In doing so, we are following the 'TRIN model' (a Danish name; 'trin' means step). This is a conceptual model of technologies that is taught at RUCs bachelor program 'humtek' (Bachelor in Technology and Humanities). The model is explained in (Jørgensen, 2018). The first step is an analysis of a technology's 'operational principle' with two suggested questions:

1. What is the purpose of the technology?
2. How does the technology achieve the purpose?

The purpose of passwords is to support user authentication. Without passwords (and with no alternative support for user authentication) a malign user could log on to a system, pretending to be a different user, by simply presenting him or herself as that user. Requiring a password eliminates this simple road to masquerading. A password achieves this purpose because a user may choose a password that is difficult to guess. Even though this observation is trivial, it is good to have a conceptual grip on why a system may choose to use passwords.

User authentication may be defined as *verification of a claimed identity*. In our 'admin' example, a user claims he or she is the sysadmin by entering the username 'admin'; and then the user verifies the claim by entering the correct (but unfortunate) password 'admindnc'.

If a password is the only means of user authentication - as in PasswordBasedAuthenticator - we may characterize the mechanism as 'single factor user authentication'. Many systems employ multiple means of authentication. For instance, users of the Danish MitID system may use a password in conjunction with a mobile phone or a physical device called a 'code displayer' (Danish: kodeviser). A system with two distinct means is termed 'two factor authentication' (2FA). The two factors should be of a different type. We may distinguish between something the user *knows* (a password), something the user *has* (possesses) (a device) and something the user *is* (eg., fingerprints). Using an extra authentication factor is the most secure way of remedying the insecurity of password-based user authentication. But 2FA also makes a system more costly and less user friendly, so the vast majority of digital servicese use passwords only, as in PasswordBasedAuthenticator. Indeed, whether passwords are used as a single factor or as part of a two-factor system, it is important that they are at least reasonably secure.

2. ‘PasswordBasedAuthenticator’

The ‘PasswordBasedAuthenticator’ lets a user register with a username and a password, and then login by presenting the correct password. When a login succeeds, a message is printed to the console saying that “Login succeeded”. That’s all. If an actual login is to happen, the program needs to be integrated into some full system.

‘PasswordBasedAuthenticator’ queries a database with a password table.

C# source files of ‘PasswordBasedAuthenticator’

The program consists of three C# programs, which you may download from the moodle site of the course Complex IT Systems. You may store them in as a separate C# project.

Program.cs. The entry program defines the end-user interface including options ‘register’ and ‘login’.

Authenticator.cs. The program defines methods `register()` and `login()` that may be called from Program.cs. When a user is registered, a record is added to the password table. At login, the password table is queried for data about the user (in particular the hashed password).

Hashing.cs. The program does the cryptographic hashing of passwords.

‘PasswordBasedAuthenticator’, similarly to SQL-Injection-Frontend, uses the library Ngsql, which you need to add to the project.

The database that ‘PasswordBasedAuthenticator’ queries

The C# source file Authenticator.cs defines class Authenticator, and the constructor of this class connects to a database. You need to change the database-related parameters of the string `s` defined in the constructor. Of course, you need to change the username and password for accessing the database. Also, you need to consider which database you want to connect to.

Regarding choice of database, this note assumes you are connecting to a database called ‘passwords’ which a single table named ‘password’. You may define the table using the SQL statement in Figure 1.

```
create table if not exists password (
  username varchar(50),
  salt char(16),
  hashed_password char(64),
  primary key (username)
);
```

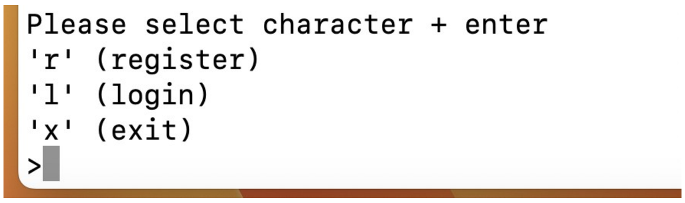
Figure 1: Definition of the table named password.

As an alternative to connecting to this simple database, you may connect to any database of your choice, but please keep in mind that the note and the program ‘PasswordBasedAuthenticator’ assumes that passwords etc. are stored in a table as defined in Figure 1.

Table ‘password’ in Figure 1 is defined to be as simple as possible. The username is a primary key. This simplifies the program that accesses the database. For instance, we do not want there to be multiple instances of the same username. It is not allowed to add to the table a record with a username that is already ‘taken’, because this would violate the primary key constraint. Thus this case is handled automatically by the database. In a practical application, you may prefer to identify a user by a number, and store the number in the password table, but we omit this for simplicity.

3. The two user interfaces of ‘PasswordBasedAuthenticator’

The first user interface of PasswordBasedAuthenticator is the end-user interface discussed already, for an end-user to register and login. The main program, Program.cs, always returns to the interface shown below in Figure 2.

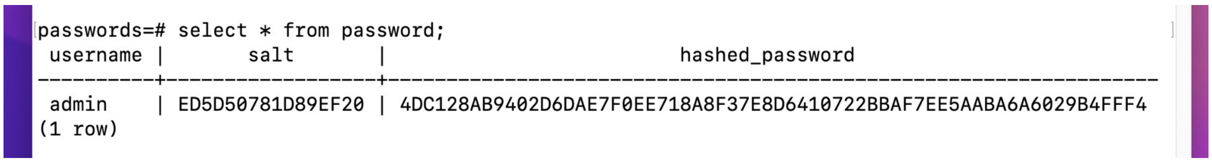


```
Please select character + enter
'r' (register)
'l' (login)
'x' (exit)
>
```

Figure 2: The end-user interface of PasswordBasedAuthenticator.

When the user selects ‘r’ or ‘l’, the program calls methods register() or login() as defined in Authenticator.cs.

The second interface is the SQL interface. This is intended for a developer or a system administrator. This interface is started in the usual way of starting an SQl client connected to database ‘passwords’. The contents of the only table in the database is shown below in Figure 3.



```
passwords=# select * from password;
 username |       salt       |                          hashed_password
----------+------------------+------------------------------------------------------------------
 admin    | ED5D50781D89EF20 | 4DC128AB9402D6DAE7F0EE718A8F37E8D6410722BBAF7EE5AABA6A6029B4FFF4
(1 row)
```

Figure 3: Using a database client to show the contents of the passwords table directly. In this case, the table contans a single row, with username ‘admin’ and a salt and a password hash.

The data in the second and third columns of the ‘admin’ record in Figure 3 are hexadecimals 0, 1, 2, .. 9, A, B, .., E, F. Hexadecimals will be discussed shortly.

The ‘admin’ record is created when the following method call is invoked from Program.cs:

```
register(“admin”, “admindnc”);
```

In Program.cs, the main road to register a user is via the end-user interface (Figure 2), asking the user to supply input. But of course, the register() method may also be called with program-defined parameters. Let’s look at the program-defined registration of user ‘admin’ in more detail. This careless systemadministrator is the note’s main example.

The first parameter of register() is the username; the second is the password. The password ‘admindnc’ is unfortunate, mainly because it contains the username as a substring. (Hopefully not a common choice for system administrators.) Moreover, let’s pretend that ‘dnc’ represents ‘Democratic National Committee’. This is the leadership body of the Democratic Party in the United Stated. The DNC’s webserver was hacked in 2016. Subsequently a total of 27,000 emails were leaked to various websites, including Wikileaks. Some of the leaked emails indicated that the committee had been biased in the 2016 race for presidential nomination between Hillary Clinton, Bernie Sanders and others. The committe, which was supposed to be neutral, had favored Clinton and worked against Sanders. A theory is that the attack was conducted by Russian, state-sponsored hackers, and that a possible motivation was to harm Clinton because of her ‘hawkish’ stand on foreign policy vis-a-vis Russia.

When register(username, password) is called, a salt is generated; then a hash value is computed using the username and salt; and finally the salt and the hash value are stored separately in the password table. The salt is generated randomly, so the salt you will see is different from the salt in Figure 3.

Hexadecimals for storing salt and hash values

Salt and hashes are bit strings, or equivalently, integer numbers. This subsection explains how PasswordBasedAuthenticator stores bitstrings/numbers in the password table as strings of characters, where each character is a hexadecimal diigit. (And then we discuss what hashes and salts are actually used for.)

The salt and password hash are bitstrings like ‘01001101....’. The salt is a 64 bit string. The hash is a 256 bit string. These strings are represented as strings of hexadecimals, as seen in Figure 3. The salt ‘ED5D50781D89EF20’ is a string of 16 characters, where each character, say E, is a hexadecimal digit. Recall from the SQL create table statement Figure 1 above that the salt field was defined to consist of exactly 16 characters. The motivation for using this representation in the prototype PasswordBasedAuthenticator is, on the one hand, that hexadecimals are human readable; and on the other hand, they are not overly expensive in terms of memory usage.

Let’s look at the ‘E’ in the admin’s salt. ‘E’ is one out of the 16 hexadecimal digits: 0, 1, 2, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. One hexadecimal digit can represent four binary digits; this is because four binary digits have a total of 2^4 =16 different possible values. Since ‘E’ is the second-largest, it represents the string of four binary digits ‘1110’. Of course, instead of ‘binary digit’, we normally use the term ‘bit’.

Hexadecimals are relatively convenient to read. The 64 bit salt is represented by 16 characters (because 64 = 4*16). If we were to display a salt in the console in raw, binary form, it would require 64 characters. But that’s the major part of a full line in the console, so that would take up too much space. Also it is easy to take a quick look to check whether two salts, stored as hexadecimals, are identical; or check whether a salt appears to be random, rather than appears to be a fixed salt with repeated patterns, such as ‘ABABAB..’. In the Postgres database system, there is also a low-level format for bit string storage. If the prototype evolves into a production system with many users, this format may be chosen to reduce memory load.

4. Passwords must have some mimimal length (Defense #1)

A system using password-based user authentication should impose password requirements. The requirements are imposed when a new user registers. If the user attempts to register with a password that does not meet the requirements, the password should be rejected and the user asked to define a compliant password instead.

Password requirements protect against password dictionary attacks, which are explained in this section.

Assignment 5 has a question about requiring that passwords have some minimal length.

While there is a general consensus that there must be *some* password requirements, there are diverging opinions about what these requirements should be. I like the recommendations of the Danish Council for Digital Security (Rådet for Digital Sikkerhed, 2022).

- Passwords must be at least eight characters long.
- At login, there must be an upper limit on the number of wrong passwords a user can try. (For instance, at most 10 wrong passwords).

This is a fairly short list. One may ask, why does the Council not recommend a requirement that passwords contain special characters? (Such as '!', '$' etc.) And both numbers and letters? And upper case as well as lower case chararacters? And why does the Council not recommend a requirement that passwords are changed at regular intervals, say, yearly? I believe the answer to all these questions is the same: All these additional requirements make password memorization more difficult. This introduces a risk of the user making password-related mistakes. Mistakes may include to forget the password, to reuse the same password for different services, or to glue to the screen a piece of paper with the password written on it.

The Council does, however, suggest some additional measures. This includes performing an automated check, at the time of registration, of a new password, for instance against an existing list of weak passwords.

Also the Council discusses an alternative to its own "ten times you're out" rule. The Council suggests that a system may replace the rule with a time delay before each new password can be entered.

It is interesting that many Danish password-based systems are in conflict with the Council's recommendations. At RUC, a user account provides access to email and other services. A password may contain *at most* eight characters, it must include various special characters, and a new password is required every year.

Threat scenario

The note’s main approach to discuss the strengths and weaknesses of defenses #1-#4 is to discuss a series of attacks on ‘PasswordBasedAuthenticator’. Before we can discuss attacks, we need to make some further assumtionps about the context or system where PasswordBasedAuthenticator is used, and exactly what information our imaginary attacker has access to.

The system complies with the recommendations of the Danish Council for Digital Security. In particular,

- the lower password length limit is set to eight characters, and
- a user is allowed up to ten failed logins, whereafter the account is closed (temporarily).

Also we assume the following about the user accounts:

- the system has a total of 50,000 user accounts,
- and the users of half of these accounts have defined a passwords consisting of exactly eight characters, such as our unfortunate system administorator admin, with password ‘admindnc’.

Regarding what information the attacker has access to, we assume

- the attacker has acccess to all usernames,
- also, in a so-called offline attack, the hacker has access to the password table.

Figure 4: Tthreat scenario - assumptions about the context in which PasswordBasedAuthenticator is used.

With reference to these assumptions, let’s clarify how the note uses the concepts of weak, strong etc. passwords:

- weak passwords: passwords that are rejected by PasswordBasecAuthenticator,
- minimal passwords: eight character passwords, as in 25,000 of the system’s user accounts,
- other normal passwords: say, nine-ten character passwords,
- strong passwords: passwords that are even longer and may be based on passphrases.

Attack types

There are two fundamentally different types of password-guessing attacks on online digital services systems that use passwords for user-authentication.

- Online attacks. In an online attack, the attacker attemps to log in with some username and a guessed password. If the attempts fails, the attacker tries another password, and so on.
- Offline attacks. In an offline attack, the attacker has a copy of the password table. That is, the attacker has a copy of every password record. In the case of our password table, a record includes the username, the salt and the hashed password (as can be seen from Figures 1 and 3).

In addition to password-guessing attacks, there are also various 'social engineering' type of attacks. In the 2014 'Celebgate' email phishing case, hackers sold or published almost 500 private fotos of celebrities, mostly women. 'Phishing' is a deliberate misspell of fishing, indicating that something fishy is happening. One approach was to obtain a password to a victim's iCloud account (where the fotos were stored) by contacting the victim and pretending to be an Apple sysadmin. Four attackers received 9-18 months sentences in the US (English Wikipedia, 2022b).

To protect against social engineering type of attacks, a set of user guidelines should be in place, including 'never give away your password'. However, this note focuses on online and offline attacks and on protective measures that are technical in nature, such as a minimal password length.

An offline attack is straightforward, if passwords are stored in their original form (not hashed). The attacker can simply read both the username and the password. (Hashing of passwords is discussed in the next section.)

An online attack that targets a particular account, such as the account of user admin, is difficult. An online attack is less difficult if the password is weak, such as 'admindnc', but still somewhat difficult. The reason is that guessing even weak passwords typically requires more than ten guesses. And after ten (failed) guesses we assume the account is temporarily closed.

Password dictionary attack

A password dictionary attack is an online atttack. It does not attack a *particular* account; rather it tries to guess the password of *some* account. Any account will do. More specifically, a password dictionary attacks targets accounts with the most easily guessed passwords. In our threat scenario, those are among the accounts with eight character passwords. Our scenario (cf. Figure 4) is that 25,000 accounts have such passwords, out of a total of 50,000 accounts.

Password dictionary attack
Type: online
Target: any user account
Method: intelligent password guessing using a password dictionary

A password dictionary is a collection of likely passwords:
pizza, pasta, admindnc, ..

The idea behind using a password dictionary is, of course, that some passwords, such as ‘admindnc’, are much more likely than others, such as ‘s/b(#=X?’.

A password dictionary may be composed from previously leaked passwords, English dictionaries or collections of ordinary English texts.

Attack #1 uses a password dictionary to attain a form of *intelligent* password guessing, rather than a *brute-force* guessing approach. A brute-force approach would be to try all combinations of eight characters. There are 95 different printable characters. Therefore the total number of passwords is 95*95* .. * 95 = 95^8. This number is 6,634,204,312,890,625 and somewhat larger than 2^{52} (which is 4,503,599,627,370,496). The size of this number is so large that the probability of guessing a pasword is very low, so that in an online attack, any account we are attacking will soon be closed.

Let’s assume the attacker has access to a good password dictionary. We assume that the password dictionary gives a probability of $1/2^{18}$ for succesfully guessing an eight character password using a single guess (a random word from the dictionary). The number $1/ 2^{18}$ is approximately

$$1/(250,000) = 0.000,004 = 0.000,4\%$$

Password entropy

The concept of ‘password entropy’ may be useful if we want to estimate the difficulty of using a password dictionary to guess a normal password. An eight character password such as ‘admindnc’ has an estimated entropy of 18 bits.

A password entropy of 18 bits means that the difficulty of guessing the password is approximately the same as that of guessing an arbitrary string of 18 bits. The number 2^{18} equals 262,144. This means that there are 262,144 possible values of such bitstrings. Unfortunately, this number is much, much smaller than 8^{95}.

The above sketch of a password entropy analysis is based on a framework suggested by NIST in (Burr *et al.*, 2006). NIST is an abbreviation of National Institute of Standards and Technology, a US agency. The major assumption is that a password character is not chosen completely at random. Rather the character is random only in the same (limited) sense that a character in a normal English text is random.

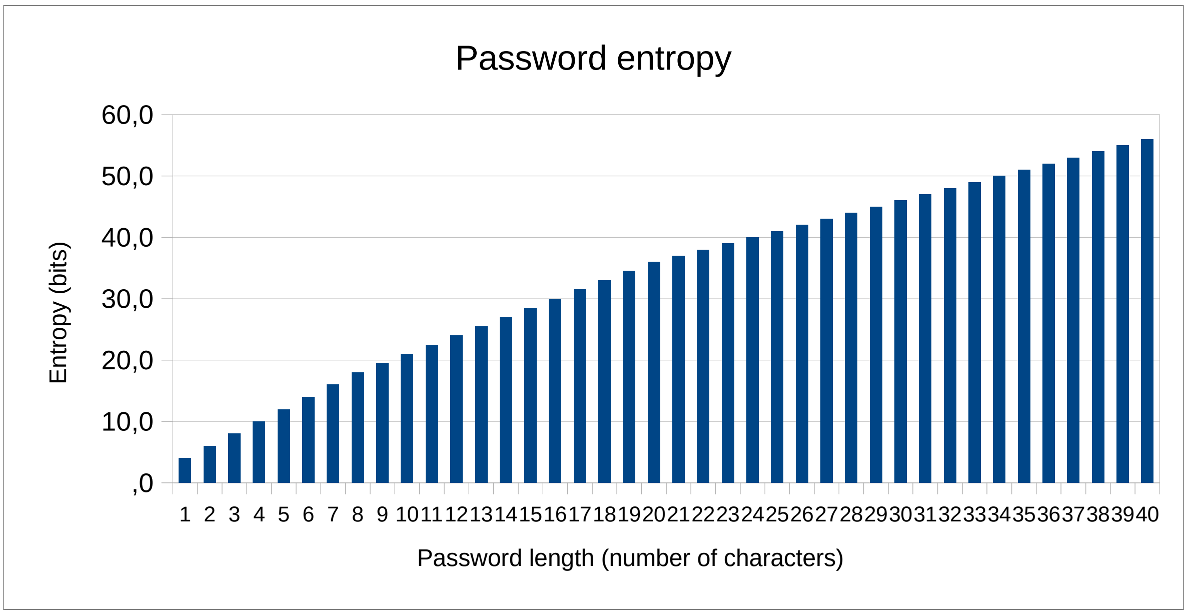


Figure 5: Password entropy. A password of eight characters has an entropy of 18 bits.

Implementation and evaluation of the password dictionary attack

To carry out attack #1, the attacker may proceed as follows

```
Implementation of the password dictionary attack (pseudocode)

for i=1 to 10 do {
   for each username u of the total of 50,000 usernames do {
      select a random eight character password p from the password dictionary
      try login(u, p)
   }
}
```

The attack iterates ten times over all usernames. This is the maximum number of failed password guesses that our system allows, for a particular account, before the account is closed.

In total, the attack involves 500,000 attempted logins, each with one password guess. If the attacker can do a total of ten password guesses in a second, the total time required for the attack is about 1.5 hours.

To evaluate the password dictionary attack, as indicated by the above entropy-based analysis, we may estimate the probability of guessing a single account's password as $1 / 2^{18}$ = 0.000,4%. This is if the account is one of the 25,000 accounts that have an eight character password.

As the next step, using ten guesses on each account, the probability of success is approximately ten times as high, that is, 10*0.000,4% = 0.004%.

Finally, the probability of cracking *some* account is approximately

25,000 * 0.004% = 10%

In other words, under the given assumptions, there is a probability of approximately 10% that the password dictionary attack succeeds in guessing - not a particular, targetted account - but some user account.

The main conclusion is that a system using password-based user authentication may from time to time experience that some user account is hacked. The accounts that are the most vulnerable to this attack are the accounts that use the most simple passwords - while still complicant with the system's mimimal passwords requirements.

The individual user may protect her or himself by choosing a strong password. A good approach is to remember a sentence such as "I am a system administrator at the Democratic National Committe with a weak password", and either use the full phrase as the password, or the first letter of each word (IaasaatDNCwawp).

The assignment question about password requirements is important in practice for several reasons. One of them is that none of defensive measures #2-#4 to be discussed below protect against password dictionary attacks.

The threat scenario defined above in Figure 4 includes the assumption that the attacker has access to all usernames. On the one hand, a system will normally not make usernames public. On the other hand, usernames may be relatively easy to get a hold of, because they may be the same as, or closely related to, the real names of users, or their email addresses.

What costs and risks does the attacker face? The attacker risks being identified. The system may try to identify the Internet address of any computer sending huge amount of failed passwords. To mitigate this defense, an attacker may employ a network of ‘bots’. A bot is another person’s computer on which the attacker has been able to install malware, to bring the computer under the attacker’s control. The name ‘bot’ is intended to resemble the name ‘robot’. A bot might be able to do atttemped logins, and also secretly report back to the attacker in case of success.

While some form of password requirements are a must, one should be aware that there are limitations of the approach. It may not be advisable to require very long passwords, such as 12 or 15 or more characters, since then some users might be tempted into unsafe password practices, as discussed above.

With some kind of password requirements in place, users are forced to define somewhat longer and/or more complex passwords, such as

pizzamaker
pastaeater
admindncweak
..

This significantly increases the entropy of passwords and implies that the risk of succesfully guessing them is reduced (although not entirely eliminated).

5. Passwords must be hashed (Defense #2)

Passwords must also be hashed. The passwords that 'PasswordBasedAuthenticator" stores in the password table are hashed using a strong cryptograhic hash function.

Requiring passwords to have some minimal length (defense #1) and also hashing them form the core mechanisms in password-based user authentication. The additional protective mechanisms that we will discuss later (defenses #3-4) can be seens as safe-guarding these core mechanisms; they serve to make the core mechanisms work as intended.

This section introduces cryptographic hash functions and motivate their importance by arguing that there is always some risk that the password table, and so the user's passwords, are leaked. If passwords are stored in hashed form, they provide no clue to the actual passwords. For user authentication, the stored password hash, which was computed at registration, is compared to a hash of the password provided at login. If the hashes are the same, we assume that the passwords are the same, and so the user is logged in.

Strong cryptographic hash functions

One property of a strong cryptographic hash function is that it is a 'one-way' function. Consider the function sha256. The PasswordBasedAuthenticator hashes passwords using sha256. The admin user whose record is Figure 3 above has the password admindnc. The sha256 hash of this is

sha256(admindnc) = B94EAB632F442F573DED7C5902DC12EB9A2DB87046F39106106CF2A3B90A96D3

The number B94E.. is the function value, or function output, when the function sha256 is applied to the password admindnc. We use the term 'hash value' or 'hash' to refer to B94E...

You may note that the hash B984.. is different from the hash value shown in Figure 3. The reason for this is that PasswordBasedAuthenticator also uses salting, that is, defense #3 as discussed in the next section.

The 'one-way' property of sha256 is that if you know the hash, and you don't know the password, you can not infer the password from the hash. It is easy to compute B94E... from admindnc; but it is practically impossible to compute admindnc from B94E.. In this context, the meaning of 'practically impossible' is that there is no approach to computing the password from the hash which is significantly better than guessing the password: guess a password, compute the hash, and see if the hash is correct.

The one-way property of hash functions provides a defense against leaked password tables. The one-way property is not a sufficient defense, and needs to be supplemented with, at least, defenses #3 and #4.

The full set of requirements that must be satisfied by a strong cryptographic hash function is listed below. The requirements are discussed in, eg., Schneier (2015).

A strong cryptographic hash function *h* must satisfy:

1. The input *p* to *h* may be of any size.
2. The output *h(p)* has a fixed, small size.
3. Given an input *p*, it is easy to compute *h(p)*.
4. Given an output *v = h(p)*, it is practically impossible to find *p*.
5. There is no known pair of values *p* and *p'* such that *h(p) = h(p')*

Item 1 discusses input size. In the case of PasswordBasedAuthenticator, inputs might be, say eight characters as in 'admindnc'. Input may also be a longer password, and we certainly want all parts of the password to influence the hash output value.

Item 2 says that the regardless of the input size, the output size is always the same. For the hash function sha256, the output size is always 256 bits.

You may observe that items 1, 2 and 5 seem to imply a contradiction. Item 1 and 2 mean that there are many more different input values that there are output values. As a consequence, some inputs must have the same output values, contradicting item 5. The contradiction is resolved because we interpret item 5 in a special manner: For a strong cryptographic hash function, while there do exist pairs *p* and *p'* such that *h(p) = h(p')*, nobody knows the values of *p* and *p'*. In cryptology, it is common to say that if this is the case, the hash function *h* has *no collisions.* Perhaps it would be better to say that there are *no known collisions*.

If a cryptographic hash function is well-designed, collisions are the more difficult to find, the bigger the hash value is. The cryptographic hash functions sha256 and sha512 are similar in operation, but their output sizes are 256 and 512 bits, respectively. It is likely that eventually, collisions will be found in sha256, and then years later, collisions will also be found in sha 512. If a cryptographic hash function has collisions, it should not be used for password hashing.

Password tables leaks

Is it realistic that a password table is leaked?

Morris and Thomsen in (1979) describe an incident where a message was sent to all users of a system and which contained the password database. The database stored users' actual passwords, so all passwords were leaked to everybody.

Of course this was an error. But errors do happen. There are many examples of password table leaks from, eg., social media (English Wikipedia, 2022a).

In many organizations, several system administrators have access rights to the password database. Thus there is a risk that a person with privileged access makes a careless mistake, or even takes some deliberate, malign action, so that an attacker gets access to the password database.

Therefore it is commonly acknowledged that offline attacks (where the attacker has acccess to the password table) are a realistic threat. As a consequence, a system using password-based user authentication should employ these two design principles:

1. The password database should be considered a confidential, highly valuable asset, and access to it should be restricted, so that offline attacks are unlikely.
2. Even though access should be highly restricted, the password database should be hard to use in an offline attack, even if is leaked.

Regarding item 2, how about protecting the password table by encrypting the password, instead of hashing the password? This would be a good approach if we could assume that the attacker does not know the encryption key. However, the approach of using encryption instead of hashing to protect passwords is generally rejected, because of the risk of leaking of the encryption key. If a password table can be leaked, so can an encryption key. Similarly to a password table, several system administrators and programs would need access to the key, implying a leakage risk of the key as well.

The word list which can be downloaded from crackstation.net (mentioned above in Section 4) is composed from, among other sources, leaked password tables.

6. Passwords must be salted (Defense #3)

The notion of ‘salting’ a password refers to adding a salt to the password before it is hashed.

Salting is already implemented in PasswordBasedAuthenticator. Figure 3 above showed that a record of the password table has the format (username, salt, hashed_password), for instance

(admin, ED5D.., 4DC1..)

A salt is a number. The value ‘ED5D..’ of the salt field is the hexadecimal representation of that number. Each user gets a salt when the user registers. The user does not select the salt; rather the salt is generated by the system. The salt should be generated randomly. Then the salt is ‘added’ to the password, and the two, taken together, are hashed. Thus the password table records the username along with the salt and the hash of the salted password, sha256(password + salt)

The salt is used everytime the password is hashed. That is, the salt is used both at registration (when the stored hash is computed) and at login (when the user provides her or his password, and the hash is computed again, and finally compared against the stored version). That’s why the password table must store the salt - the salt is needed at login, so that the salt can be added to the password before hashing. Because obviously, the salt that’s added to the password at login-hashing must be the same salt that was added at registration-hashing.

Intuitively, salting makes a password hash more ‘complex’. Specifically, salting protects against rainbow table attacks, a kind of offline attacks (where the password table has been leaked). Rainbow tables and a rainbow table attack are explained below in this section.

Password salting in PasswordBasedAuthenticator is implemented in method hash() of Hashing.cs. The first two lines of code of the method generates a salt.

```
byte[] salt = new byte[salt_bytesize];
rand.GetBytes(salt);
```

Figure 6: Source code for generating a salt in method hash() from Hashing.cs

The source code shown above in Figur 6 firstly defines a byte array with 8 bytes. This corresponds to 64 bits. Then values for the byte array is assigned via a call to rand.GetBytes(). The variable rand points to a *pseudo random number generator*. This generates 64 bits that are ‘pseudo random’.

A scheme for password salting should be designed with two major objectives:

1. The salt should be so big that it is impossible to construct a rainbow table (see below) that includes password hashes for all possible salt values.
2. The salt should be randomly generated.

The objective that salts are random means that salts shouldn’t be generated in some predictable way, say, a fixed order 0, 1, 2, etc. The pseudo random number generator used in PasswordBasedAuthenticator is claimed to be cryptographically secure in the sense that it is impossible to predict anything about the next salt that it generates, even if you know all the previous salts. Secure random number generator is designed for, among other things, providing input for the generation of encryption keys. In the context of encryption keys, it is crucial that an attacker is unable to predict the numbers generated; because if the attacker has any such information, this may guide the attacker in guessing an encryption key. However, for password salts, I am not aware of any succesful password cracks that are based on predicting salt values, so the use of a cryptographically secure random generator is not absolutely mandatory. However, since many programming languages, including C#, provide one, and since their use incur no significant cost (in terms of time or memory), it is good practice to use one. With a 64 salt bit length, salts are long enough (and perhaps longer than they need to be). But again, the amoung of space used for 64 bit salts is insignificant, so we'll opt for the long salt.

Rainbow table attack

Password salting protects against rainbow table attacks.

Rainbow table attacks are offline attacks. They work on data from a leaked password table. They attack password tables whose records contain only pairs (username, password_hash) - that is, a password table that does not use salting.

```
Rainbow table attack
Type: offline
Target: any user account
Method: intelligent password guessing using a rainbow table

A rainbow table is a table of pairs (password, password_hash)
where password is selected from a very large password dictionary
(password1, password1_hash)
(password2, password2_hash)
(password3, password3_hash)
..
```

The passwords (and their hashes) stored in a rainbow table are passwords hoped to be defined by some users of the system that the attack wants to target. We assume that the system employs defense #1. Therefore a simple password dictionary, such as the one considered above in the discussion of the password dictionary attack, will not suffice (in most cases). Instead we assume the attack selects all passwords from some very big dictionary with, say, a billion passwords. A very large password dictionary can be downloaded from http//www.crackstation.net. Also, there are programs that generate likely passwords by picking a word from a base list and then add simple prefixes or suffixes such as -

pizza1234, pastaeater, admindncweak, ..

The crucial idea behind a rainbow table is to build the table in advance of the attack. The table is built by hashing all passwords in the (very big) dictionary, and add pairs (password, password_hash) to the table. In our case, the table would use the hash function sha256.

Implementation and evaluation of the rainbox table attack

From the attacker’s point of view, the benefit of a rainbow table is that while the construction of the table may require considerable time, the attack itself is very fast. A rainbow table attack may be implemented as follows:

```
Implementation of rainbow table attack (pseudocode)

The attack uses a rainbow table that has already been built

for each username u of the total of 50,000 usernames do {
  fetch the user’s hashed password h in the leaked password table
  search the rainbow table to see if h is the hash value of a known password
}
```

A rainbow table attack is fast because searching through the hash values already recorded in the rainbow table is very fast. Similarly to the dictionary attack, it is unlikely that the password for any *particular user* is found. But since the password hashes of all user accounts are looked up in the attack, there is a significant probability that password of *some user* may be guessed.

A rainbow table may be extended so as to store hash values computed from different hash functions, say, values from ten commonly used hash functions. Each record would then have the format

(password, sha256_hash, sha512_hash, ..)

The construction of such a rainbow table requies ten times the time and ten times the memory, compared to a rainbow table for a single hash function. The construction of a “multi hash function” rainbow table is a practical possibility with comtemporary CPU and memory hardware, even when using passwords from huge password dictionaries.

Here is why password salting protects against rainbow table attacks. Password salting prevents building the rainbow table in advance of the attack. Consider our system administrater with username ‘admin’ and the record with the salted password -

(admin, ED5D.., 4DC1..)

Supporse our ‘admin’ user has passsword ‘sysadmin’ and the password is contained in our very large password dictionary. However, we cannot compute the hash value without the salt. We may only begin atttacking the ‘admin’ account once we obtain the leaked password table with the salt ‘ED5D..’. With the salt, the attack may begin. For each password in the (very big) password dictionary, the attack now needs to compute the hash of (password + ED5D..) and compare with the stored value ‘4DC1’. This process is time consuming. While the ‘delayed’ attack remains somewhat dangerous, it is crucial that salting eliminates the option of precomputing a rainbow table.

Alternatively, an attacker insisting on precomputing a rainbow table with all possible salt values, will face both time and space challenges that are impossible to solve. This is because the rainbow table needs to compute and store the hash values of astronomical numbers of pairs (password, salt). Recall that with a 64 bit salt, there a 2^{64} different salt values.

7. The hash function must be iterative (Defense #4)

The final defense that a password-based system for user authentication must implement is that the hashing of passwords (with salts) must be iterative.

The motivation for this is simply to increase the time required to obtain the final hash of a salted password. This is to slow down online attacks on the users' passwords that rely on guessing a password, computing the hash, and then comparing with the stored hash. If we increase the time required to do a hash with a factor ten, this also increases the attack time with a factor of ten.

How far should we go - increase by a factor ten? A factor 100? The general advice is to make the time required to hash a user password *long*, but not so long that it annoys the legitimate user when registering or logging in. The legitimate user will experience the hashing time as a delay. Perhaps an acceptable delay is a couple of tenths of a second, or at most a half second or a full second.

Assignment 5 includes a question about implementing an iterative hash function. Indeed the hash function used in 'PasswordBasedAuthenticator' is already iterative, but only uses one iteration. This can be seen from the definion in Hashing.cs of the function hashSHA256().

```
private string hashSHA256(string password, string saltstring) {
   byte[] hashinput = Encoding.UTF8.GetBytes(saltstring + password);
   byte[] hashoutput = iteratedSha256(hashinput, 1);
   return Convert.ToHexString(hashoutput);
}
```

In the function body, the function iteratedSha256() is called, but with a parameter value of 1, indicating a single iteration only. It is technically straightforward to increase the value. The challenging part of the question is to strike a balance between security (a slow hash function) and usability (not annoying the legitimate user).

In iterative hashing, the final hash value is computed as

   hash(hash(... hash(input) ..))

where the input is the salted hash. One may observe that to arrive at the final hash value, it is necessary to first compute all the intermediate hash values. This is a reasonably safe approach. However, rather than defining iteration by hand as a programmer, it is recommened to use an established standard for iterative hashing, such as PBKDF2 (for Password-Based Key Derivation Function 2). The library Microsoft.AspNetCore.Cryptography.KeyDerivation contains an implementation of this standard. The name PBKDF2 indicates that the standard is intented not only for hashing of passwords, but also for derivation of a key for encryption.

8. Discussion

The core mechanisms presented above for secure password-based user authentication include having some minimal password requirements (defense #1), hashing of passwords to mitigate the risk of password table leaks (defense #2), and salting and iterative hashing (defenses #3 and #4) to increase the work required for various password-guessing attacks.

Iterative hashing increases the computing time required to guess a password. It makes password guessing ‘CPU-intensive’.

Researchers and other security experts have argued that password-based systems must also consider forcing password-guessing to be ‘memory-intensive’. This is because recent developments have made cheap hardware chips available that are dedicated to cryptographic hashing. The market for such chips has evolved in response to demands from, among other actors, businesses in the BitCoin community. A central mechanism in BitCoin is a competition to guess a value having a certain hash value, with the incentive of a financial reward.

As a reaction to the threat posed by chips dedicated to hashing, new hashing algorithms have been proposed with the purpose of forcing password guessingto be both ‘CPU-intensive’ and ‘memory-intensive’. This is memory in the sense of RAM (Random Access Memory). A password hashing competition was held in 2013-2015 to compare various new algorithms, including Argon2 (Aumasson, 2019).

9. Exercises

Exercise 1. (Building a rainbow table.) Select a couple of likely passwords. Consider, for instance, the passwords ‘admindnc’ or ‘hclinton’ or passwords of your own choice. Compute the sha256 hash of the password. For the computation you may use the online tool at https://miraclesalad.com/webtools/sha256.php.

Exercise 2. (Rainbow table). Check whether the rainbow table of https://crackstation.net/ contains the password hashes you computed in Exercise 1.

References

Jean-Philippe Aumasson et al Aumasson, 2019. “Password Hashing Competition,” at https://www.password-hashing.net/, accessed 5 November 2023.

William E Burr, W Timothy Polk and Donna F Dodson, 2006. “Recommendation for Electronic Authentication,” at http://csrc.nist.gov/publications/nistpubs/800-63/SP800-%0A63V1_0_2.pdf.

English Wikipedia, 2022a. “2012 LinkedIn hack,” at https://en.wikipedia.org/wiki/2012_LinkedIn_hack, accessed 29 October 2002.

English Wikipedia, 2022b. “2014 celebrity nude photo leak,.”

Niels Jørgensen, 2018. *Digital signatur. En eksemplarisk analyse af en teknologis indre mekanismer og processer,* at http://webhotel4.ruc.dk/~nielsj/research/publications/indre-mekanismer.pdf.

Robert Morris and Ken Thomsen, 1979. “Password security: A case history,” *Communications of the ACM*, volume 22, number 11, at https://dl.acm.org/doi/pdf/10.1145/359168.359172.

Rådet for Digital Sikkerhed, 2022. “Anbefalinger om passwords,” at https://www.digitalsikkerhed.dk/wp-content/uploads/2021/03/Sikrepasswords-Anbefalinger-om-passwords.pdf, accessed 25 October 2022.

Bruce Schneier, 2015. *Applied Cryptography*, Indianapolis, Indiana: Wiley.