SE448: Information and Cyber Security	**Recitation 7–8**
Semester 2 5785	**15–22 May 2025**
Lecturer: Michael J. May	**Kinneret College**

Managing Passwords

Many information systems and applications expect users to choose passwords to identify themselves. The passwords are used during the authentication step in the system, but often a key element is ignored in the system design: where are the passwords stored? Storing passwords in a plain text file or a regular database is convenient, but then appropriate measures must be taken to secure the password repository.

Standard best practices for managing password repositories include protecting the repository in general (*i.e.* protecting it with a password, encrypting it, making it inaccessible to any users) along with including protection of the password file contents itself. This way if the repository protections fail and the contents are leaked to an attacker, all is not lost. Common content protection mechanisms include encrypting or hashing the passwords before storage.

In this recitation we'll develop a simple password management tool which stores usernames and passwords in a plain text file. The use of a plaintext file is solely for convenience. In a real system, the password information would be kept in a protected directory or database. We'll first develop a simple version of the tool which just stores the password information in plaintext. Then we'll add hashing to the passwords to make them one way summaries. Finally, we'll add *salt* and use the libc `crypt` function that will ensure that even identical passwords appear different in their hashed summaries.

1 A Simple Password Manager

We'll first describe the simple password manager tool. It will serve as a first iteration which we will improve in the following stages.

1.1 Password File

The tool starts up by parsing the password file stored from the previous use of the tool. The file is called "password-file.txt" and is stored in the local working directory.

The file is formatted as follows:

UserName;Password;Role

We parse the file based on the assumption that the file is correctly formatted. Since we identify users by the username, we don't allow multiple users with the same username. If two lines in the file have the same username, the second one is just ignored.

1.2 Add a User

To add a user, we pass the following parameters to the tool:

-operation "add"

-user The user name to add (no spaces allowed)

-password The user's password (no spaces allowed)

-role The name of the role the user will have (no spaces allowed).

The tool will then add the user to the password file with the provided information. If an existing username is chosen, the existing one is overwritten with the new data. The tool then outputs a success message:

User added

If the addition fails, output a failures message:

User was not added.

1.3 User Login

We can test the password file by using the login operation. The tool uses the password file to examine whether the given credentials are acceptable. To login a user, we pass the following parameters to the tool:

-operation "login"

-user The user name to add (no spaces allowed)

-password The user's password (no spaces allowed)

The tool will check if the user and password combination are valid. If they are found in the file, the tool outputs a success message:

User [user] with role [role] successfully logged in.

For example:

User Dan with role Realtor successfully logged in.

If the user name and password are not valid, the tool outputs a failure message:

User [user] login failed.

```
artimus;ifsh45df;Student
barty;^college;Student
cindy;^college;Student
Michael;RegularPassword;Lecturer
```

Figure 1: Sample password file contents

For example:

> User Dan login failed.

A sample simple password file is shown in Figure 1. Note that barty and cindy have the same password - that's something we'll come back to later.

1.4 Build system

As with previous projects, we will use the Gradle build system to manage the dependencies and code. Use the starter project on Moodle to begin with a valid project with a Kotlin Gradle build settings file.

1.5 What to do

Your first tasks are to do the following. Start with the empty simple password project and fill in the TO DO sections:

1. User addition - add users to the users hashtable and the saved password file
2. Login - check the provided user name and password against the hashtable

2 Hashing Passwords

If we look at the password file for the tool we developed in the previous section, we see an obvious problem: the passwords can be read by anybody who gets a hold of the password file. A sample file is shown in Figure 1. This means that administrators or anybody who can access the password file can masquerade as anybody else. If the data is stored in database, we have the same problem.

A better idea would be to obfuscate the password file contents in a manner which prevents others from discovering the password while preserving the ability to verify them. This can be done easily by computing the hash digest of the password before it is stored in the password file. We first convert the password to a byte array using UTF8 and then use SHA-512 to compute a hash digest of the password before it's stored in the password file in a Base64 encoded string. The user interface is unchanged, but as shown below the password is stored in a digest format which can't be reversed. This means that if someone steals the password file, it won't be possible to recover the passwords easily.

After hashing the input passwords, the password file from Figure 1 will be as shown in Figure 2.

```
artimus;OrRFsihbQfSMXkpSX5tYW/y8dRA3IauekD+swx8uTnm0NIwKePe98W6FycWy0EpPgwZnf
6nOsMT4XNsXUuJ8wQ==;Student

barty;+UfrFdwH7B7ao3loSJCEOOhUZVB4wN+5leFRbTX/SN6wvBSLU1Y0ZeuR9PD+oFW8ymtGkfB
f7C8t15ne/J23jw==;Student

cindy;+UfrFdwH7B7ao3loSJCEOOhUZVB4wN+5leFRbTX/SN6wvBSLU1Y0ZeuR9PD+oFW8ymtGkfB
f7C8t15ne/J23jw==;Student

Michael;UhX2+KAwmCwmKYb+AZJZLR++jOlT9y4j0VRGaf7cQkRgIszqI7J/r56aspAzXrfhBnLa2
4wzuJeIksuY4n7iPg==;Lecturer
```

Figure 2: Hashed password file contents

2.1 What to do

Your next tasks are to do the following. Update the password tool you created to do the following differently:

1. User addition - hash the provided password with SHA-512 before you save it to the users hashtable
2. Login - check the provided user name and password after you hash it with SHA-512

3 Salted Hashed Passwords using crypt

The password file we developed using the hash digest removes the problem of the password file being readable, but still leaves us with a corner case: if two users select the same password they will appear the same in the password file. Such a case can be seen in Figure 2 where the users barty and cindy chose the same password. This reveals information to barty or cindy if they manage to get a copy of the password file. In general, popular passwords will appear as identical hashed values, giving a strong hint where to attack.

A better solution is to include a salt value for each password. That will ensure that even identical passwords will look different in the password file. Doing so requires us to modify the password file format to include a salt value for each user. A good standard for encrypting passwords using hash functions and salt is the libc `crypt` function that we mentioned in class. You can read about it at the `crypt(3)` man page at: `http://man7.org/linux/man-pages/man3/crypt.3.html`. We'll use crypt with SHA-512, a much more secure choice than using the default DES implementation of crypt.

For ease of use, we'll use the Apache Commons Codec implementation of crypt which you can download from `https://commons.apache.org/proper/commons-codec/` or by including the following Gradle dependency (in the dependencies section of the Gradle file):

```
implementation("commons-codec:commons-codec:1.15")
```

Among other use functionality, Apache Commons Codec offers the `Crypt` class the implements the crypt utility in Java in a way that is compatible with crypt(3). You can read the Javadoc for the `Crypt` class at: `https://commons.apache.org/proper/commons-codec/archives/1.15/apidocs/index.html` We will use the `public static String crypt(String key, String salt)` override for our tool.

Once we have added the use of crypt and salt, we'll need to store the resulting passwords and salts in the password file. We'll use the following format:

UserName;HashedPassword;Role;Salt

3.1 crypt and Salt

The crypt tool uses the `$` as a delimiter between the three parts of the password string and returns the salt as part of the hashed and salted password. For instance, the string:

```
$6$1/FdkRRhmKLXVZF3$CD.gmeLAkWoJCKaHXpdXKImDnwU2ovXh9psz0WpVM.X3xcnJsknNKsQ.VK
O3ta14hBn3mJ5NfqIR955a9G1MO0
```

has three parts:

- The 6 which is the code to use SHA-512 as the hash function for crypt (and not the default DES). You can see all of the codes for the hash functions that crypt supports at: `https://commons.apache.org/proper/commons-codec/archives/1.15/apidocs/org/apache/commons/codec/digest/Crypt.html#crypt-java.lang.String-java.lang.String-`
- The The salt value `1/FdkRRhmKLXVZF3`. It is 16 characters long, the maximum allowed with SHA-512 in crypt.
- The hashed and salted password CD.gmeLAkWoJCKaHXpdXKImDnwU2ovXh9psz0WpVM.X3xcnJsknNKsQ.VKO3ta14hBn3mJ5NfqIR955a9G1MO0

Since crypt uses the $ as a delimiter, you can provide the whole string above to crypt as the salt and it will ignore the parts are the not required for the hashing and salting (*i.e.* the third part).

3.2 Salt Generation

While crypt will generate a random salt for you if you want, it's best to generate your own and provide it to crypt. We'll use a `SecureRandom` to generate 16 random bytes and encode them using Base64 to generate a salt string.

The tool must select a new random salt value for each user on addition. The user doesn't need to select a salt value - it's chosen automatically as a randomly generated 16 byte value when performing the Add method. The complete contents of the salted and hashed password file can be seen in Figure 3.

The salt value is stored automatically in the password file so the user doesn't need to remember it. When the user performs the Login method, he only needs to provide the password. The tool gets the appropriate salt and checks it against the value stored in the file.

After implementing the salted hash using crypt, we can see in Figure 3 that even two users with the same password (barty and cindy) have different hashed password values, hiding the fact that they have the same original password.

```
artimus;$6$uasAmtc0Aj3iPsPK$RM4ReH.KoisfkPFux5t2Tuta1sQhmc/MUl0hh57j3eKm4SpIc58uf0putPu
TkOpFZZ2mEKlAaUfIwStw55yKl/;Student;$6$uasAmtc0Aj3iPsPKEXFI/w==

barty;$6$Xmgt2ZKvuISVX8qM$/F69.qGfgw9uIb/KL6jgXtWQujbteqSTWTJFx7UU/3LcLMEaNVxHzqtVUdIQZ
r8GXJFZ.SvNYrtVbvXNMqLZa0;Student;$6$Xmgt2ZKvuISVX8qMnp3W+A==

cindy;$6$PELfDxREPhQLRwSO$KGYpDkR4xLNwNMN5SfrK8v7UChHDXEpatf4XZfEOGkCHosMLIbHBt9DPDkAMP
wTvE67pKR8P5992VusAeEAGe/;Student;$6$PELfDxREPhQLRwSO1tGvXg==

Michael;$6$1/FdkRRhmKLXVZF3$CD.gmeLAkWoJCKaHXpdXKImDnwU2ovXh9psz0WpVM.X3xcnJsknNKsQ.VKO
3ta14hBn3mJ5NfqIR955a9G1MO0;Lecturer;$6$1/FdkRRhmKLXVZF3RUDGrw==
```

Figure 3: Salted and hashed password file contents

3.3 What to do

Your next tasks are to do the following. Create a copy of your hashed password tool and modify it to use the crypt salted hashing method.

1. Download Apache Commons Codec or include it in the project using Gradle.
2. User addition - compute a hashed and salted value using crypt. Be sure to first generate a salt with SecureRandom. Save it to the users hashtable. Update the file saving and loading routine as needed.
3. Login - check the provided user name and password using crypt and the salt. Take the salt from the users hashtable entry and the provided password as the input.

Note that in order to run the tool from the command line, you'll need to put the Apache Commons Codec JAR file in the same directory as the executable JAR file and use the class path command to make the tool run. See the sample test files for an example.