COMP 448: Computer Security

Fall 2019

Project 2: Secure Password Storage for Password Authentication

Date Released: 10/24/2019

**Assignment:** Secure password storage is crucial for the security of password authentication. Many password theft incidents nowadays occur due to improper password storage. In this project, we implement a password store similar to the Linux operating system password file (/etc/shadow). The password store supports salted passwords hashed using the SHA-512 cryptographic hash function.

The Linux operating system stores user passwords as salted password hashes. The Linux password file is located in /etc/shadow and is only accessible by the root user. Each password entry contains the following fields:

* username
* salted password hash
  + password hash type
    - 1 = MD-5
    - 5 = SHA-256
    - 6 = SHA-512
  + encoded password salt
  + encoded salted password hash
* number of days since user last changed password
* number of days until user may change password
* number of days until user must change password (password expiration)
* number of days to warn user before password expires
* number of days to disable user account after password expires
* number of days user account has been disabled

Each password entry is on a separate line. The fields in a password entry are separated by a colon (:). In addition, each sub-field of the salted password hash field begins with a dollar sign ($).

The encoded salted password hash is computed as follows:

Encoding ( Hash ( password || Encoding ( salt ) ) )

where “||” denotes concatenation, “Hash” denotes the hash function determined by the password hash type, and “Encoding” denotes a custom encoding algorithm used by Linux.

A sample password entry looks as follows:

testuser:$6$p3hJzy6d$I/UjcNIa.VWlnJQJqXttVXA02.C9YqFp9Q2Ngm53ORbP3TP6cQjJPvis0/518gjtPrH4FDIF5q7doOA7RQhTJ.:17862:0:99999:7:::

In this entry, the hashed password is “testpass”:

* username = testuser
* salted password hash
  + password hash type = 6 (SHA-512)
  + encoded password salt = p3hJzy6d
  + encoded salted password hash = Encoding ( SHA-512 ( “testpass” || “p3hJzy6d” ) ) ) = I/UjcNIa.VWlnJQJqXttVXA02.C9YqFp9Q2Ngm53ORbP3TP6cQjJPvis0/518gjtPrH4FDIF5q7doOA7RQhTJ.

For this project, we will implement a simplified version of the Linux password file with the following fields:

* username
* salted password hash
  + password hash type = 6 (SHA-512)
  + Base-64 encoded password salt
  + Base-64 encoded salted password hash

The implementation needs to use SHA-512 and the Base64 encoding (see more information below) instead of the Linux custom encoding. The remaining, number-of-days, fields in the Linux password file should not be implemented. The project must compute the salted password hash in the same way as Linux:

Base64 ( SHA-512 ( password || Base64 ( salt ) ) )

You need to use the ‘:’ and ‘$’ delimiters the same way as in the Linux password file to delimit fields and sub-fields. For example, a password entry for user ‘alice’ and password ‘S3cret!Pazw’ might look like this:

alice:$6$362f5EAV8ws=$sh2Nmqjgj62M2VDg2EDmcfTB52cJmJLqR5o6GTrFYgpHHJSJ3kRSxx+UKkJKCAP1sg6keqmRNSIrG+SOFTQQ4g==

To help you with this assignment, you are provided with four files:

* Makefile: you can use this file to build your project. It compiles your code and generates the ‘pass\_store’ executable. To use this Makefile, simply run: make
* main.c: this file contains the main method that runs the project. It reads the command line arguments that the user specifies and calls the proper function in pass\_store.c. You are not allowed to make any changes to this file.
* pass\_store.h: this file contains the declarations of the functions that you need to implement. Each function is fully commented, and the parameters are fully listed. You are not allowed to make any changes to this file.
* pass\_store.c: this file is the bulk of your project. Fill in all of the empty functions:
  + pass\_store\_add\_user: adds a user account to the password store
  + pass\_store\_remove\_user: removes a user account from the password store
  + pass\_store\_check\_password: checks the password of a user in the password store
  + ﻿\_\_pass\_store\_load: accessory function that loads the password store from file
  + \_\_pass\_store\_save: accessory function that saves the password store to file. Setting the ‘append’ flag appends new user accounts to the password file, instead of replacing the password file.
  + Make sure that you close any open files, allocate memory correctly, and deallocate all the memory locations used.

Here is the sequence of commands used in a sample run of the project:

* make
* ./pass\_store add-user alice
* ./pass\_store check-password alice
* ./pass\_store remove-user alice

By default, ./pass\_store uses a file “passwords” for the password store.

**SHA-512**

You can use the OpenSSL API to compute SHA-512 hashes:

#include <openssl/sha.h>

unsigned char \*SHA512(const unsigned char \*d, size\_t n, unsigned char \*md);

For more information, see the SHA512() manual page:

<https://www.openssl.org/docs/man1.0.2/crypto/SHA512.html>

OpenSSL also provides the macro SHA512\_DIGEST\_LENGTH equal to the size of the SHA512 hash output in bytes (64).

**Random Number Generation**

You can use the OpenSSL API to generate random data:

#include <openssl/rand.h>

int RAND\_bytes(unsigned char \*buf, int num);

For more information, see the RAND\_bytes() manual page:

<https://www.openssl.org/docs/man1.0.2/crypto/RAND_bytes.html>

**Base64 Encoding**

Base64 encoding is commonly used to encode binary data as human-readable, printable ASCII text. It uses the characters A-Z, a-z, 0-9, +, and /. Base64 encoding is used in PEM-encoded cryptographic data (keys, certificates, etc.), the HTTP protocol, and other applications. In order to encode 256 possible values of a binary byte into the 25+25+10+1+1 = 62 characters in the Base64 character set, each three bytes of binary data are encoded as four Base64 characters. Padding is added to pad the Base64 encoding to multiples of 4 bytes.

For example, the Base64 encoding of the string “testuser” is dGVzdHVzZXIK. The Base64 encoding of the string “testus” is dGVzdHVzCg==, where “==” are padding characters.

We introduced the OpenSSL BIO API in Project 1. In particular, BIO filters allow applications to transform data in some way. One of the BIO filters provided is BIO\_f\_base64(), which encodes/decodes data to/from Base64. BIO filters, sources, and sinks, can be chained so the output of one BIO object is redirected into the input of another BIO object.

For example, we can Base64 some data and write it to a memory buffer by chaining a Base64 BIO filter to a memory buffer BIO sink as follows:

// sample buffer with data to Base64 encode

uint8\_t buf[24];

﻿// Base64 filter

BIO \*b64\_bio = BIO\_new(BIO\_f\_base64());

// Memory buffer sink

﻿BIO \*enc\_bio = BIO\_new(BIO\_s\_mem());

// chain the Base64 filter to the memory buffer sink

﻿BIO\_push(b64\_bio, enc\_bio);

// Base64 encoding by default contains new lines.

// Do not output new lines.

﻿BIO\_set\_flags(b64\_bio, BIO\_FLAGS\_BASE64\_NO\_NL);

// Input data into the Base64 filter and flush the filter.

﻿BIO\_write(b64\_bio, buf, 24);

BIO\_flush(b64\_bio);

// Get pointer and length of data in the memory buffer sink

﻿char \*data\_ptr = NULL;

long data\_len = BIO\_get\_mem\_data(enc\_bio, &data\_ptr);

// Finally, free the BIO objects

BIO\_free\_all(b64\_bio);

Similarly, we can Base64 decode some data from a memory buffer by chaining a Base64 BIO filter to a memory buffer BIO source as follows:

// sample buffer already populate with data to Base64 decode

uint8\_t buf[32];

// output buffer

uint8\_t out\_buf[24];

// Memory buffer source

﻿BIO \*enc\_bio = BIO\_new\_mem\_buf(buf, 32);

// Base64 filter

﻿BIO \*b64\_bio = BIO\_new(BIO\_f\_base64());

// Chain the memory buffer source to the Base64 filter

﻿BIO\_push(b64\_bio, enc\_bio);

// Base64 encoding by default contains new lines.

// This Base64 encoded data doesn’t have new lines.

﻿BIO\_set\_flags(b64\_bio, BIO\_FLAGS\_BASE64\_NO\_NL);

// Extract decoded data from Base64 filter into output buffer

﻿int num\_read = BIO\_read(b64\_bio, out\_buf, 24);

// Finally, free the BIO objects

BIO\_free\_all(b64\_bio);

For more information, refer to the OpenSSL manual pages for the BIO API functions:

<https://www.openssl.org/docs/man1.0.2/crypto/>

To debug your Base64 encoding/decoding code, you can use the Linux base64 command line tool to encode and decode data. See the base64 Linux manual page for more information.

**Rubric:** To receive credit, submit a .zip file via mygcc containing the all of your code.

Points will be awarded based on the following:

|  |  |
| --- | --- |
|  | Points |
| Task 1: The project allows the application to add a new user to the password store. | 20 |
| Task 2: The project allows the application to remove a user from the password store. | 20 |
| Task 3: The project allows the application to check a user’s password. | 20 |
| Task 4: The application correctly loads the password store from file. | 20 |
| Task 5: The application correctly saves the password store to a file. | 20 |

**A program that does not compile or link will not be graded.** You are not allowed to make any changes to ab.h or main.c files. Your code should compile with the provided Makefile.

**Your code must compile, link, and execute in the Linux environment using the class VMs: no compile equals no credit. Also, if your code crashes, no partial credit will be given**

The following criteria will be used to grade your submission:

* Does the code compile?
* Does the code function according to the problem specification?
* Is the code readable and well-formatted? Is it well-documented and clear?

**Extensions will not be granted for technology-related issues.** Leave yourself enough time to complete the assignment, submit the assignment using mygcc, and contact the instructor if you run into problems.

**Project Policies:**

* Assignments must be submitted electronically via my.gcc. Be sure to upload your files correctly the first time.
* This project is a group project. Students are expected to keep the same team for the whole semester.
* 20% of the grade will be weighed with the peer evaluation. Students are expected to turn in the peer evaluation form posted on mygcc at the end of the semester. If a student works on his or her own without a team, then 20% of the grade will be deducted from the project grade.

**Academic Integrity Policy:**

* Each team is expected to work on its own. Members of each team can work together, discuss ideas, look at each other’s code, and share files among each other.
* Students belonging to different teams should not discuss, share code that directly bears on this project, or look at each other’s code. Any instances of this will be considered a violation of the academic integrity policy of this course and will be reported to the SFRC committee.
* Use or possession of past solutions and similar solutions from online resources is strictly prohibited and is considered a violation of the academic integrity of this course.
* You may use online resources to look up how to use a function or a system call, but you may not copy code from online resources. Any copied code (whether cited or not) is considered a violation of the academic integrity policy of this course.