COMSW 4181: Security I **(Due: 10/14/21, 1 PM EST)**

Homework Assignment #1

Instructor: Suman Jana *Name: , UNI:*

Course Policy: Read all the instructions below carefully before you start working on the assignment, and before you make a submission.

- You have a total of three late days that you may choose to use in any of the assignments without any penalty. Any submission after the deadline will be considered as a late day and if it exceeds 24 hours after the deadline, then it will be considered as two late days. You may use the late days in separate assignments or together. Once you have used all of your late days, you will not receive any credit for late submissions.
- Due date (window) and time: 10/14/21 from 1 pm to 1:15 pm. After that submissions will be counted late. It does not make any difference if it's only 10 minutes or 23 hours.
- You are not allowed to discuss the solutions to homework problems/programming assignments with your fellow students.

Submission: You should create your own file `UNI.zip` with all the source code and written component. Skeleton code in C++ will be provided. Please note this homework must be completed in either C or C++. If you provide Python, Java, Rust, etc. it will receive a 0.
Google cloud: All the assignments are designed to run under Ubuntu 20.04 LTS. To solve the environment issues, we use google cloud for the running environment. The detailed setup can be found in the following google docs: `https://docs.google.com/document/d/1GE2Gc8WNp0sox5DieQZbHkHP-ZkowfRTj3jHCElydQY/edit?usp=sharing`
The following packages will be installed in our VM by default:

- build-essential
- cmake
- curl
- git
- openssl
- valgrind
- unzip

You are welcome to use other packages to finish your assignment, but you will need to provide a **setup.sh** script, and we may not be able to help as much with a different package.

Problem 1: Adversarial Thinking (10 points)

(a) Which of the following best describes the difference between MAC and digital signatures? Please submit `UNI-hw1.pdf` in which you briefly describe the two methods and choose one of the options.

(1) While a digital signature can only be verified using the secret key for signing the message, message authentication codes can be verified only based on the message.

(2) While a digital signature can only be verified using the public key of the signer, message authentication codes can be verified only based on the message.

(3) Digital signatures can be verified based on only the message, while message authentication codes can only be verified using the secret key that was used generate them.

(4) While a digital signature can be verified using the public key of the signer, message authentication codes can only be verified through the secret key that was used to generate them.

Problem 2: Encrypted File Store (90 points)

The goal of this assignment is to build an encrypted file store. The stored files are each encrypted; the archive as a whole is integrity-protected.
The executable file will be called *cstore*. The command **MUST** accept the following options:

```
cstore list archivename
cstore add [-p password] archivename file
cstore extract [-p password] archivename file
cstore delete [-p password] archivename file
```

Do NOT implement any additional commands! You are responsible for checking arguments if they are both valid and in the correct order.
The command you build **MUST** be named cstore, and it must accept exactly this syntax. This will be taken care of in the Makefile and skeleton code provided.
Multiple files may appear on each command. If no password is given, it should be read from the keyboard. Use of the *getpass()* library routine is encouraged but not required. (For reasons we'll discuss later in the semester, the -p option is not particularly secure. However, it's useful for testing.)
To convert a password to a cryptographic key, iteratively apply SHA2-256 at least 10,000 times. (Again, this isn't the best way. You may implement PBKDF2 if you wish, but it's not required and there's no extra credit.)
A password can be capped to be at most 12 characters long. If it is longer, throw an error as described next section.
The attacker's goal is to steal one of the files, or to corrupt the archive. Therefore, the files must be encrypted and the archive must be integrity-protected. However, it must be possible to list the archive without supplying a password.
I suggest download AES and SHA2-256 source code from `https://github.com/B-Con/crypto-algorithms` by Brad Conte (brad@bradconte.com), though you're welcome to use any other open source implementation you find. (If you do, document the location.) I've compiled and tested those files on Linux. If you use them:

1. You may only use the AES key setup routine aes_key_setup and the ECB encrypt/decrypt routines (aes_encrypt and aes_decrypt)
2. The aes.h file is (skimpy) documentation of the parameters to each routine. You may find it useful to look at aes_test.c for sample calling sequences
3. If you need any other modes of operation, you must implement them yourselves

Similarly, see sha256.h and sha256_test.c for how to use SHA2-256. The style—calling sha256_init() to initialize the state, using sha256_update() for each block of additional text to be hashed, and sha256_final() to produce the final output—is the way every cryptographic hash function I've seen operates.
If you need cryptographically strong random bytes, read them from /dev/urandom. Note that since its output is constantly changing, you may see non-reproducible errors. You may, if you wish, build in some test scaffolding; if you do, you must document it.
You must submit:

1. Source code
2. A single Makefile to compile everything.
3. A short document explaining any design choices. You must explain which AES mode of operation you used and why. You must also explain how you have chosen to integrity-protect the archive

You may optionally submit your test scripts. We will provide some tips on testing/fuzzing/code coverage for C++ code shortly.
Note your archive will be a file, do NOT create a directory structure. Essentially consider it something like a *zip* file, except compression is not a requirement. Efficiency is not especially important when implementing this program.

More details on implementation

The primary goal of the assignment is to give you experience with implementing integrity and encryption. So, to simplify some edge cases:

(add)
Some edge cases you don't need to worry about:

- If the file is empty, please ignore it.
- If you are adding a file to an archive that exists, please go to the error case.
- An archive should be created if you run "cstore add <archive> <file>" and <archive> does not yet exist.
- If at least 1 file is empty or doesn't exist, you can just create an *error.txt* file, as described in the **error** subsection.

(list) List the members of the archive. Assuming the following commands are executed:

```
cstore add -p security1 security_archive hello_world.txt HW1.pdf HW2.pdf
cstore list security_archive
```

Please create a *list.txt* that would have the following content:

```
hello_world.txt
HW1.pdf
HW2.pdf
```

- Please over-write the *list.txt* if it already exists with new list contents
- Please don't add any other metadata to the contents of *list.txt*, I will be using list to verify archive changes.
- You don't need to worry about the order of the files listed. You are welcome to sort it, but there is no extra credit for this.

You can assume file character names will not exceed 20 characters long. If it is longer, you should throw an error.

(extract)

- Decrypt the named files and write them in the current working directory
- When the file is extracted, please delete the file as well from the archive
- If at least 1 file doesn't exist, you can just create an *error.txt* file, as described in the **error** subsection.

(delete)

- The files should be deleted from the archive
- Ensure that when you run "list", the file is that was deleted is no longer listed as well.
- If at least 1 file doesn't exist, you can just create an *error.txt* file, as described in the **error** subsection.

(Errors) It is critical that your code does not crash, and your code will be expected to have **no memory leaks**.
If you have an error, e.g. your archive got tampered, archive doesn't exist, please do the following:

- Create an *error.txt* file.
- The *error.txt* file will contain something explaining the cause of the error

Frequently Asked Questions

These were questions pulled from Fall 2020 Piazza relating to this homework. Hopefully, these provide a lot more clarifications.

a. Running HMAC on an encrypted file requires an authentication key in addition to the key used to encrypt the file. Suppose I only have one key, and that I've already used it for the encryption. Is there a conventional/secure way to generate an authentication key from it (say, by hashing it)?

The usual way to do that is to concatenate something else and then hash again. The short answer is that after you convert the password to a key, as specified in the assignment, you use that key as an input to a "key expansion function". The simplest thing to do is to hash that key together with strings indicating the purpose of the new keys. Here, you might calculate H(k || "integrity") and H(k || "confidentiality"). But the two strings could just as easily be "I" and "C", they're not secret, and the invocation of H will mix them in thoroughly.

If you want the gory details, see https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Cr2.pdf and in particular the definition of "FixedInfo" on p 19.

b. In the homework, what is meant by "Multiple files may appear on each command"? Would it be possible to get an example of a command we'd need to support containing multiple files in the command?

Sure here are some examples

```
cstore add security_safe contacts.txt vouchers.docx
cstore extract security_safe contacts.txt vouchers.docx hello_world.c
cstore delete security_safe contacts.txt vouchers.docx README.md
```

security_safe would correspond to the archive name, everything else are some files e. g. contacts.txt, hello_world.c, etc.

c. If we are using HMAC for integrity check, are we expected to implement HMAC ourself from scratch or we can use open source? Thanks

If you want to use an HMAC, you are expected to implement it yourself. But recall that you are given a crypto library to use (including AES and SHA2 functions).

d. Just wanted to clarify the behavior of the API: How are archives expected to be created in the first place? If I run "cstore add archive f1 f2 f3" I presume this should create n archive that ultimately has those 3 files encrypted? Then if I run "cstore add archive f4" should it add the file f4 (encrypted) into the existing archive?

Yes. Each archive has one password and can store multiple files. You should be able to add a file into the archive once it has been created. An archive could be created if you run "cstore add <archive> <file>" and <archive> does not yet exist. The purpose of the assignment is in the cryptographic aspect of it and not in creating the best archive program.

e. when adding to or deleting from an existing archive we need to validate the password used to ensure it was the same as the one used when creating the archive. To do this would it make sense to use the password to re-run HMAC and validate that the output matches the output we had stored for the integrity check?

Consider a scenario where you are logging in to your account on a social media platform. When you sign up, the password you enter is stored after hashing (if they follow good security habits). When you log in, the password you enter is hashed and compared. It is never a good idea to store passwords in plain-text. Hopefully this analogy helps clarify your doubt.

f. Should we have integrity checking for the list command? If so, what are some ways to run HMAC without taking in the password to generate the key, when generating the HMAC key?

No, no integrity check on the list command.

g. **Let's say I were to iteratively apply SHA to a password "hello" three times. Which of these gives the intended result? 1. SHA(SHA(SHA("hello"))) 2. SHA("hellohellohello") I'm pretty sure it's Option 1 but I just want to make sure. Thanks! Does that mean we have to call both sha256_init and sha256_final every time?**

   Yes use Option 1 and yes you have to call sha256_init and sha256_final everytime.

h. **What exactly does integrity protection refer to and how can we do that? Is that mean we should make some hash functions on our cipher-text? How could we prevent our files from being modified by others who don't have correct keys?**

   Integrity check is NOT preventive. It does not prevent an attacker from tampering with your file, but if she does, you can detect that there is something wrong.

   Integrity protection is done by computing a MAC (message authentication code) on the ciphertext and then concatenate the code to the ciphertext. code = MAC(k, ciphertext) The final product is (ciphertext, code).

   We check the integrity of a (ciphertext, code) pair by checking if code and MAC(k, ciphertext) match.

i. **I have a question about the AES encryption here. I know SHA is used to convert a passport to a cryptographic key, so what does AES do in this scenario? And one more question is that if "add" is to add a new file, which I presume is empty, and there is no command about editing a file, and "extract" is to decrypt a file, does that mean we need to prepare a encrypted file in the archive?**

   AES is a block cipher that you use (coupled with an appropriate mode) to encrypt a file.

   Your cstore add should be able to add an existing file (possibly non-empty). You don't have to implement an editting command.

   Yes. The files should be encrypted (and their integrity be protected).

j. **Can we use the aes_encrypt_cbc() function in aes_test.c? Or must we implement our own CBC mode? I feel it is hard for me to come up my "own" implementation after reading the source code.**

   As per the instructions, you may only use aes_key_setup, aes_encrypt, and aes_decrypt, so no you're not allowed to use aes_encrypt_cbc. That being said, once you can do AES in ECB mode, implementing CBC mode should be near-trivial. Just implement it yourself using code you wrote yourself and understand

k. **If the user passes in an unproperly formatted archive file, can we simply let the behavior be undefined? (i.e. crash, memory leaks, etc. are all fair game in this case)**

   Also, can we let the behavior be undefined if the user passes in command-line arguments not according to the format specified?

   Please cover those cases with errors and/or exits. Any memory leaks or crashes will be counted as bugs.