CSCI 4830-801:
Introduction to Enterprise Application Security
Fall 2016

Final Project

Due: October 27, 2016

1 Objectives

Project Objectives:

- Learn how to use security algorithms in an application
- Learn how to use existing security libraries
- Learn how to write security code in a popular industry programming language
- Learn how to analyze a security implementation
- Learn how to measure security performance

Skills objectives:

- Perform basic password authentication
  - Using salting and stretching
  - Simulate UNIX `passwd` file
- Encrypt a file
- Decrypt a file
- Ensure integrity of file
- Apply access control
- Only decrypt file if authorized
- Generate log records of each action

2 Project Overview

In this project, you will implement a basic file security system. You will build a system that allows authorized users to encrypt and decrypt files while ensuring that the files are not altered while encrypted. The following sections serve as a specification for this system.

To keep things simple, this system will be console based (so it will just use basic text input to issue commands). Create a directory that we will refer to here as HOME that will contain

a single sub-directory called `data`. The HOME directory will eventually contain a single file called `users`. The `data` sub-directory is where we will store the encrypted data files.

The system will operate in two modes: an administrator mode and a user mode.

2.1 Administrator Mode

When the system starts, it will use the current directory (or the directory given on the command-line) as HOME. It will look in HOME to see if the users file is present. If so, the administrator must have supplied the administrator user name (`admin`) and password to the program when starting (see Section 2.4). Otherwise, the system is in initialization mode and immediately prompts the user to enter an administrator password (for user `admin`). The system should ask for the password twice and confirm that both entries are the same. The system should not echo the password to the console as the user is typing.

The system will then create a new, empty `users` file and will create an entry for `admin`. At this point, the system should exit so that the user is forced to start the system again, but this time with the user name `admin` and the just set password. See Section 5 for more details on how to create the `users` file.

Certain commands are available only to the administrator. These will be marked in the commands table below. If a non-administrator attempts to issue an administrator command, the system should reject the request and log an UNAUTHORIZED message to the log.

2.2 User Mode

User mode is entered when the system is started with credentials that identify the user as anyone other than the administrator.

2.3 Commands

Table 1 lists the commands may be issued by an authenticated user once inside the running system.

In general, the administrator adds users to the system (assigning each an initial password). These users can then log into the system to either encrypt or decrypt files managed by the system. Files that are encrypted are stored in the `data` sub-directory. The code will be a password that will be used to encrypt and decrypt the file (details will be specified later). Note that the system will only decrypt a file if the user had been previously authorized.

A sample administrator session might look like the following[1]:

```
$ fileprotector --user=admin --password=admin-password
> adduser donald iamgreat
Success.
> adduser hillary 1stwomanpres
Success.
> adduser bernie ohsoclose
Success.
> exit
```

[1]The `>` symbol is the prompt that the `fileprotector` program displays to let the user know it is waiting for input. Depending on the language used, the actual syntax to instantiate the program will be different.

Table 1: List of input commands

Command	Permission	Effect	Syntax
`adduser`	Administrator	Causes a user to be added to the system. Each username must be unique in the system.	`adduser` *username password*
`setpassword`	User	Allows the current user to change her password.	`setpassword` *password*
`encrypt`	User	Encrypts and integrity protects a file.	`encrypt` *file-name code*
`decrypt`	User	Checks that the user is authorized to decrypt the file and, if so, decrypts the file.	`decrypt` *secure-file output-file code*
`authorize`	User	Allows a user to access a protected file. Only the owner of the file can successfully authorize other users.	`authorize` *username file-name*
`deauthorize`	User	Removes access to a protected file. Only the owner of the file can successful deauthorize users.	`deauthorize` *username file-name*
`exit`	Any	Exits the system	`exit`

A sample user session logged in as `hillary` might look like the following:

```
$ fileprotector --user=hillary --password=1stwomanpres

> encrypt /usr/hillary/Documents/thanks.txt 2016
Success.
> authorize bernie thanks.txt
Success.
> encrypt /usr/hillary/Documents/goaway.txt 1234
Success.
> authorize donald goaway.txt
Success.
> exit
```

A sample user session logged in as `bernie` might look like the following:

```
$ fileprotector --user=bernie --password=ohsoclose

> decrypt thanks.txt /usr/bernie/Documents/thanks.txt 2016
Success.
> decrypt goaway.txt /usr/bernie/Documents/goaway.txt 1234
Unauthorized.
> authorize bernie goaway.txt
Unauthorized.
> deauthorize bernie thanks.txt
Unauthorized.
> exit
```

A few notes:

Table 2: Command-line arguments

Parameter Name	**Description**	**Usage**
user	Supplies the user name of the person accessing the system.	`--user=`*username*
password	Supplies the password that authenticates the given user.	`--password=`*password*
home	Specifies the directory that will act as HOME (i.e., that contains the `users` file and the `data` sub-directory). When not supplied, defaults to the current directory.	`--home=`*directory-path*

- Once a file is encrypted (and stored in the `data` sub-directory) it can be referred to without a full file path (i.e., just the basic file name). This can be a problem if you encrypt files with the same name from different source directories.
- The *code* can be any string. It is used to generate a key that is used to encrypt the file (see Section 6).
- This is NOT a truly secure system. There are a number of flaws that make this system unusable in the real world. This system is just for you to gain experience in understanding the basics of what goes into a secure system. See Problem 2 in Section 8.

2.4 Command-line Arguments

Table 2 shows the parameters that can be entered at the command-line when starting the program.

When no user and password parameters are supplied, the system should behave as if it is in initialization mode (see Section 2.1).

Note that in a secure system, passwords are *never* entered on the command-line. This is because most UNIX systems display the entire command-line as part of the output to `ps`[2], so any user on the system would be able to see passwords entered this way. More secure solutions include simply prompting for the password once the program begins or reading the password from a protected configuration file.

3 Cryptography Packages

There are a number of packages that perform the actual cryptography that you will need for this project - you are not being asked to implement any of the cryptographic algorithms yourself. Indeed, one of the goals here is to learn how to use an existing library to perform the cryptographic primitive operations.

The following sections discuss options available for the popular languages Java, Python and C++.

[2]The UNIX command that shows process status information.

3.1 Java

The interfaces for the Java cryptographic functions are in both the `java.security` and `javax.crypto` packages. While implementations for `java.security` are built-in to the JDK, implementations (providers) for the `javax.crypto` package are not. You will need to install a provider. The two most popular are in the following table (both are free):

PROVIDER	URL
Bouncy Castle	`http://www.bouncycastle.org/java.html`
Sun (Oracle)*	`http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html`

*Note that the Sun JCE is included in the JDK, but in order to get the key sizes needed, you need to installed the Unlimited Strength Jurisdiction Policy files that is included in this URL. Following the README instructions to understand how to install the policy files.

3.2 Python

Python is nice because cryptographic packages are easily found, for example, via `pypi.org`. There appear to be many to choose from. Here are some (in no particular order):

PACKAGE	URL
cryptography	`https://cryptography.io/en/latest/`
PyOpenSSL	`https://github.com/pyca/pyopenssl`
PyCryptodome	`https://pycryptodome.readthedocs.io/en/latest/src/introduction.html`

While you should do your own research, on first blush, the `cryptography` package looks like a good one to use.

3.3 C++

Some choices for C++ are in the following table:

PACKAGE	URL
Crypto++	`https://www.cryptopp.com`
Libgcrypt	`https://www.gnu.org/software/libgcrypt/`
OpenSSL	`https://www.openssl.org`

For C++ developers, it would appear that Crypto++ may be the easiest to use, but I have no personal experience with it.

4 Logging

Create a module that can be used to audit all security operations. You can use any logging library with which you are comfortable including rolling your own simple logger. Logging messages do not need to be fancy for purposes of this project, but they should each include at least the following information in each record:

- User identifier of the user that caused the action to be taken
- Timestamp including: year, month, day, hour, minute, second and millisecond
- A status (SUCCESS, INFO, FAILURE, UNAUTHORIZED) indication depending on the result of the action (see table below)
- A text description of the action including any variable data
  - If an action failed, the message should include the non-sensitive parameters along with why the action failed (e.g., "User login failed because user name XX was not known").

The following table gives guidelines on which status to attach to a logging message:

Status	**Indicator**
SUCCESS	A requested security operation was allowed and succeeded.
INFO	This message just contains information that may be useful to someone viewing logs to determine state of the system. This should never contain sensitive information.
FAILURE	A requested security operation was allowed but did not succeed for some reason. For example, a login attempt failed because the password was incorrect.
UNAUTHORIZED	A requested security operation was not allowed because the user did not have permission to perform the action.

You may also output debugging messages (with status DEBUG), but these should be turned off (through configuration) at the time you turn in your project[3].

While you are free to roll your own logging mechanism, it is encouraged to use one of the following popular packages as they are common in industry (especially Apache log4j):

Java	**C/C++**	**Python**
`java.util.logging` Apache log4j[6]	Apache log4cxx[4]	`logging` module[5]

We want to get the logging mechanism working first since you will want to add logging messages throughout the rest of your project, both for formal auditing of security actions and to help you debug any problems.

Note that logging packages have their own logging levels. To keep things simple, you can map your SUCCESS and INFO messages to the logger's equivalent INFO level. FAILURE and UNUAUTHORIZED messages should be mapped to the logger's equivalent ERROR or SEVERE level. But be sure to include the status level from the above table in the text part of the log message.

[3]Specifically, this means you can (and probably should) leave your debug logging code in your program, but your logging configuration file should be sure to only output messages of level INFO or higher.

[4]`https://logging.apache.org/log4cxx/latest_stable/usage.html`

[5]`https://docs.python.org/3/howto/logging.html#logging-basic-tutorial`

[6]`http://logging.apache.org/log4j/2.x/`

5 Authentication

To perform authentication, we will emulate a simplistic version of the UNIX password system. A user is authenticated by providing a user name and a corresponding password.

You should familiarize yourself with how the UNIX `passwd` file is formatted. We will name our file users and we will use a much simpler version. Each user's information is kept on a single line in the text file. Each line (a record) is a series of fields, each separated by a colon and terminated by a newline, that has the following format:

username:secured-password:date-last-updated

The *username* is a string that identifies the user and can be any sequence of numbers, letters and certain special characters, but never a space. We will just use basic names for our purpose, but it doesn't really matter what you use.

The *secured-password* has a format that is taken from UNIX's shadow password file. In UNIX, passwords are no longer stored in the same file as usernames and user data. They are now stored in a shadow file that only root has access to. Look at `http://www.yourownlinux.com/2015/08/etc-shadow-file-format-in-linux-explained.html` for details on how the shadow file is formatted.

For our purpose, we will format *secured-password* as:

5salt$hashed-password

The 5 indicates that we are using SHA-256 to hash although you can use SHA-512 (and then 6) instead. The *salt* is an encoded[7] 128-bit random string. The *hashed-password* is the result of prepending the *salt* to the user's input password and then performing 10,000 rounds of hashing (thus both salting and stretching).

Finally, the *date-last-updated* can be any string representation of the date and time the record was last updated (or, if new, the date and time it was first inserted).

When the system starts, a username and password is passed in via the command-line. The system should immediately authenticate the user. If the username is not known or the password is incorrect, a single FAILURE message should be written to the log and a message should be displayed on the console indicating that authentication failed. The program should then immediately exit.

If authentication succeeds, a single SUCCESS message should be written to the log and a Success message displayed on the console. The user is then given a prompt where she can perform actions as specified in the Commands table (see Section 1).

5.1 Process of Generating the Hashed Password

To generate the hashed password, perform the following[8]:

1. Generate a cryptographically secure random number (actually any randomly generated string of bits). You will need 128 bits. This is your salt.

[7] Anywhere this document refers to "encoded" binary strings, you can feel free to choose any valid way to encode raw bits into an ASCII encoded string of characters. Examples include base 64 encodings and hexadecimal strings.

[8] This is not quite the correct algorithm as defined by PBKDF2. That algorithm uses HMAC and rehashes the password in each iteration along with new salt which is the previous round's output XOR'd with the previous round's salt input. See `https://en.wikipedia.org/wiki/PBKDF2` for a reasonable explanation of the details.

2. Prepend the salt to the user's input password. This is your initial input.
3. Using SHA-256, generate a hash of the initial input.
4. Using the output from (3) above as new input to SHA-256, hash again.
5. Repeat step (4) 10,000 times using the output of the hash as the input to the next hash.
6. Store the encoded salt and the encoded result of step (5) in the `users` file as described above.

5.2 Process of Authenticating a User

To authenticate a user, perform the following:

1. Look up the username in the `users` file. If the username does not exist, return FAILURE.
2. Decode the corresponding *salt* from the corresponding record and then perform steps (2)-(5) as described in Section 5.1.
3. Decode the corresponding *hashed-password* from the `users` file and compare that result with the result from step (2)[9].
4. If the values are the same, then authentication has succeeded, return SUCCESS.
5. Otherwise, return FAILURE.

5.3 adduser

The `adduser` command creates a new record in the `users` file. Only the administrator can perform this action (if anyone else tries, log an UNAUTHORIZED message to the log and console).

It is an error to add a user with a username that already exists, so be sure to check for this case. If that happens, issue a FAILURE message to the log and console.

5.4 setpassword

The `setpassword` command allows the current user to change her password. This results in an updated record in the `users` file where the *secured-password* section is updated as well as the *date-last-updated* field.

6 Data Protection

This system allows a user to specify a file on the file system and then store an encrypted version of the file in the `data` sub-directory. Then, an authorized user can request to decrypt the encrypted version by simply knowing its name and having the appropriate security code.

In order to track file authorizations and file integrity digests, we will create another database (file) called `files`. This database will reside in the same directory as the `users` database file. It will have the following format:

[9] You could also just encode the result and compare it directly to the already encoded *hashed-password*. It is essentially the same amount of work either way

file-name:integrity-value:code-salt:owner:authorized-users

When a file is encrypted (via the `encrypt` command), a new entry is added to the `files` database. The entry will have 4 fields, each separated by colons as shown above.

- The *file-name* field will be the name of the file (without any path). This is the name that will be used during the decrypt operation and the name as stored in the `data` directory.
- The *integrity-value* field will be the encoding of the digest (using SHA-256) of the plaintext file.
- The *code-salt* field will be the encoding of 16 bytes of salt that will be used to create the key material along with the encryption code when applied to the PBKDF2 algorithm.
- The *owner* is the user name of the user who originally encrypted the file (and caused this record to be created).
- The *authorized-users* field is a comma-separated list of user names who have been authorized by the owner to be able to decrypt the file. Each time the `authorize` or `deauthorize` command is used for this file, this field must be properly updated.

6.1 Process of Encrypting a File

When a user executes the `encrypt` command, the following steps should be performed:

1. Open the target file for reading. If the file does not exist, generate a FAILURE message.
2. Open an output file in the `data` sub-directory whose name is the same as the root name (i.e., name without the path) of the target file. If the output file cannot be opened, generate a FAILURE message.
3. You will need to convert the input *code* (i.e., file password) into a secret key that can be used by the AES algorithm. Choose a 16-byte salt value for this file. Apply the PBKDF2 algorithm that will convert a character array containing the password plus the salt into suitable key material that can be passed to AES.
4. Read the target file. While reading, encrypt the input data stream to the output file. Use AES (128 or 256 bit) with CBC mode and PKCS #5 Padding[10],[11]. At the same time, be computing the digest of the file using SHA-256. This way, you should only have to read the original file one time.
5. Close the target file and the output file.
6. Create the `files` entry record as described above. The *integrity-value* should be the encoded version of the digest computed in step (4). The original set of *authorized-users* should just be the owner (the current user). The *code-salt* should be the salt value you computed when creating the key material.
7. Update the `files` database file by adding this new record to the file. Note that it is not important where in the database you add the record.
8. Generate a SUCCESS message.

[10] In Java, for example, this would be passed to `Cipher.getInstance()` as `AES/CBC/PKCS5Padding`.

[11] You can also choose to use no padding, but in this case, you will have to apply the padding yourself since block ciphers require certain sized block sizes.

6.2 Process of Decrypting a File

When a user executes the `decrypt` command, the following steps should be performed:

1. Search the `files` database for the specified *secure-file* given with the `decrypt` command. If no record for the *secure-file* is present, generate a FAILURE message.
2. Upon finding the record, search the *authorized-users* field for the user name that matches the current user. If the current user's user name is not in the list, generate an UNAUTHORIZED message. The owner is always an authorized user.
3. Extract and decode the *integrity-value* field.
4. Extract and decode the *code-salt* field.
5. Open the *secure-file* in the `data` sub-directory. If unable to do this, generate a FAILURE message.
6. Open the *output-file* given in the command. If unable to do this, generate a FAILURE message.
7. Generate the secret key that will be used to decrypt the file using the provided *code* and the *code-salt* field extracted in step (4). You will use same PBKDF2 algorithm as you did in step (3) of the encryption procedure, but using the *code-salt* as the salt instead of generating new salt.
8. Using the provided *code* as the key, decrypt the input cipherstream and write the plaintext output to the output file. After decrypting, be sure to also compute the digest value of the plaintext.
9. Close the input and output files.
10. Compare the computed digest value with the value extracted in step (3). If they are different, generate a FAILURE message *and delete the output file* (as it is corrupted OR the code was incorrect).
11. Generate a SUCCESS message.

6.3 encrypt

The `encrypt` command should execute the steps defined in Section 6.1. Note that the *code* is the key that should be used to encrypt the file.

6.4 decrypt

The `decrypt` command should execute the steps defined in Section 6.2. Note that the *code* is the key that should be used to decrypt the file.

7 Authorization

This system allows the owner of a file to determine what other users may see the files contents (via decryption). At the time of a decryption request, the system checks if the requesting user is authorized to perform the operation. This is done by checking to see if the user is included in the authorized users list for the given file. Authorization checks result in either approval, in

which case the operation is performed, or denial, in which case the operation is not performed and the user is informed via an UNAUTHORIZED message.

7.1 authorize

The `authorize` command adds a user name to the list of authorized users for the named file. This operation only succeeds if the user executing the command is the owner of the named file. Authorization occurs as follows:

1. Read the `files` database to find the record whose *file-name* matches the given *file-name* argument. If the record is not found, generate a FAILURE message.
2. Check that the owner user name matches the current user's user name. If not, generate an UNAUTHORIZED message.
3. If the *username* supplied to the `authorize` command is not already in the *authorized-users* list, then add it. Otherwise, do nothing.
4. Generate a SUCCESS message.

7.2 deauthorize

The `deauthorize` command removes a previously authorized user from the list of users allowed to decrypt a file. The command performs the following steps:

1. Read the `files` database to find the record whose *file-name* matches the given *file-name* argument. If the record is not found, generate a FAILURE message.
2. Check that the owner user name matches the current user's user name. If not, generate an UNAUTHORIZED message.
3. If the username supplied to the `deauthorize` command is present in the *authorized-users* list, then remove it. Otherwise, do nothing.
4. Generate a SUCCESS message.

8 Completing the Project

In order to get full credit for this project, you need to complete the following:

1. You will submit all your source code via Moodle.
   (a) If you have a build script or make file, you should also submit those.
   (b) You should include any external configuration files (e.g., for logging).
   (c) You must submit a step by step guide on how to build and run your program.
   (d) Be sure your code is reasonably commented with emphasis on non-obvious sections of code.
2. Write a single page (submit to Moodle) that addresses the following questions:
   (a) Why would this system not be considered secure? What are the flaws in the design and/or implementation?

(b) What improvements could be made to secure this kind of system? Improvements can range from a new design to changes in implementation details.

3. Measure two different symmetric key algorithms. The base project uses AES with either 128 or 256 bit keys and CBC mode. Choose at least one other algorithm (you can vary the actual algorithm, the key size and/or the encryption mode) and then measure the time it takes to encrypt and decrypt the same input. The input file should be at least 5 MB in size (or big enough to show a difference on your machine). You should report your results in a table (and submit to Moodle) that compares algorithms with speeds for both encryption and decryption. A sample table is shown below:

Algorithm	**Encryption**	**Decryption**
AES/CBC/PKCS5Padding (128)	*time-to-encrypt*	*time-to-decrypt*
Blowfish/CBC/PKCS5Padding (128)	*time-to-encrypt*	*time-to-decrypt*