[redacted]

ENEE457

11/27/2020

Project 4: Build It – Secure Gradebook

Part I: Design

To develop a secure wrapper program that provides both privacy and integrity to a gradebook on the cloud, I designed my gradebook using various nested structures named Gradebook, Student, and Assignment and utilized cryptographic primitive schemes from the OpenSSL library. Each gradebook can contain up to 200 students and keeps tracks of the current student count, current assignment count, and total weight of all assignments. Each student can have up to 40 assignments, and a first and last name up to 20 characters. The two other fields: total_grade and assignment _grade, are used in the gradebookdisplay program for sorting when print_assignment or print_final is called to display the students alphabetically or by their grades from highest to lowest. Each Assignment contains a 20-character name, its points value, the assignment grade for the student, and its weight. To ensure integrity against unauthorized modifications, it also contains a 20-byte SHA1 hash value created from the students first and last name for each grade associated with an assignment. Since the largest class sizes at University of Maryland are around 200 students and the number of assignment would be for only one semester, I made reasonable and valid assumptions to limit my gradebook to 200 students with up to 40 assignments each. My programs verify that these limits are met when adding or deleting a new student or assignment to the gradebook, so a professor, ta, or adversary should not be able break this capacity. Furthermore, the length of input strings and the string characters are checked by a parse_commands() function in each program to make sure that the inputs are valid and don't overwrite past predefined buffers in the structures. For example, assignment names must only be alphanumeric characters, while first and last names of students must be alphabetic characters with no spaces. In all programs, I utilize the AES-128 encryption in CBC mode for encryption and decryption to protect the gradebook data, which follow the OpenSSL sample code example. Thus, each time someone accesses the gradebook locally, the gradebook will be read from and written to an encrypted text file for the cloud.

In the **setup** program, I initialize a new gradebook with the specified name, empty name fields, and all other number fields set to 0. To generate a 16-byte secret key and 16-byte IV for AES encryption, I used RAND_bytes which generates random bytes that are cryptographically strong from the entropy source, /dev/urandom. This secret key is converted to a string of hexadecimal digits and gets output to STDOUT for the tas and professor to use on their local computers with this wrapper program. Then, I encrypt the gradebook with the iv and key and write to a new file. I create a new buffer that concatenate the 16-byte IV with the ciphertext of the gradebook after it. This IV can be viewed publicly because this block will be retrieved in gradebookadd and gradebookdisplay to decrypt the gradebook ciphertext first. Every time we modify the data, a new IV will be generated for encryption of the gradebook before it is uploaded back to the cloud.

The **gradebookadd** program receives the gradebook filename, the key, and a set of inputs from the command line. In the main function of this program, parse_commands() is called with the command line arguments and parses the options using getopt_long_only from the opt library. This functions validates the order of these commands, their string length, and checks whether they come from the set of allowed options is {-N, -K, -AA, -DA, -AS, -DS, -AG, -AN, -FN, -LN, -P, -W, -G}. Flags are set to see if an option and its corresponding argument has been correctly received. For example, if the action "Add-Student" via "-AS" is received, the function also makes sure that the -FN *student-first-name* and -LN *student-last-name* are also received. Inputs are also checked based on the project specifications, such as the *student-first-name* containing only alphabetic characters (a-z, A-Z) in upper and lower case or *assignment-points* being a non-negative number. I also created a CmdLineResult structure that stores and organizes the arguments, which will be used later in the program to query and modify the gradebook. If any of the command line arguments are found to be invalid, the program will print "invalid" and exit with error code 255. Otherwise, the program will proceed by reading in the data from the gradebook file. The first 16 byte block retrieved from this data is the IV, while the rest of it is the ciphertext of the gradebook. The input key string is validated by calculating its sha1 hash and checking if it matches the sha1 hash value that was stored in the file. The key input is converted from hex digits back into a 16-byte unsigned char block. Using this key, the ciphertext data is decrypted with the IV. Depending on the action argument provided, this program will either add/delete assignment, add/delete student, or add a grade. If an assignment

with the specified name does not exist, or a student with the same first and last name already exists for add-student, or the student is not found for delete-student or add-grade, then an error occurs. For add-grade action, if an existing student with a graded assignment already exists, the student's grade will just be updated. Once the gradebook has been modified, I first generate a new random IV and encrypt the gradebook with this new IV. Then, I concatenate and store the new IV block with the ciphertext of the gradebook in a new buffer using memcpy(). The buffer data is written back to the gradebook file, overwriting the original contents with the new information. As a result, the updated gradebook is re-encrypted on the cloud, and a ta or a professor can retrieve its data again.

The **gradebookdisplay** program will accept the gradebook filename, the key, and a valid inputs from the command line. Similar to gradebookadd, a parse_command() function will parse the commands and validate them by checking if they are a subset of specified allowed options from {-N, -K, -PA, -PS, -PF, -AN, -FN, -LN, -A, -G} with corresponding values needed for an action. These actions are print-assignment, print-final, print-student, and each have their own function in the program. If both -A and -G are specified, then an error occurs. Once the format of the input strings is checked to be valid, the program proceeds by doing the action specified, does the query from the gradebook, and displays the information on the terminal for the user. This program reads the entire data from the file and extracts the first 16 bytes for the IV. The remaining part is the ciphertext of the gradebook. The input key string is converted from hex digits to its original 16 unsigned char byte format. With the IV and key, the program runs decryption on the ciphertext to obtain the original gradebook data. For example, when print_final() is called to print out the final grades for all the students, the total grade of each student is calculated based on their grade for each assignment out of the assignment points multiped by that assignment's weight. The students and their final grades are either displayed alphabetically by last name, then first name, or numerically from highest to lowest if "-A" or "-g" option is supplied. In order to sort the students, I copied them to another list and applied quicksort "qsort()" on the list with a comparator function. The comparator will compare two students based on their total_grade or assignment_grade, returning a value to define the order of these student elements. If an assignment with specified name is not found for print-assignment or the student is not found for print-student, an error occurs, and program exits and prints "invalid".

```
4   typedef struct _Assignment {
5     unsigned char sha[20];
6     char name[20];
7     int points;
8     int grade;
9     float weight;
10  } Assignment;
11
12  typedef struct _Student {
13    float total_grade;
14    int assignment_grade;
15    Assignment assignments[40];
16    char firstname[20];
17    char lastname[20];
18  } Student;
19
20  typedef struct _Gradebook {
21    int student_count;
22    int assign_count;
23    float total_weight;
24    Student students[200];
25  } Gradebook;
```

Figure 1: Gradebook structures format in data.h

Part II: Vulnerabilities

Vulnerability #1: Adversary inspects the gradebook file itself to learn facts about the names of students, assignments, or grades of individuals

If the sever is compromised, the adversary cannot view the contents of the gradebook file because the gradebook structure in the file is encrypted using a 128-bit (16 byte) secret key stored locally and a random initialization vector IV. Using the OpenSSL library, I implemented AES-128 encryption and decryption in CBC mode for setup, gradebookadd, and gradebookdisplay programs to implement privacy. Thus, if an adversary tries view the text file with a hex editor, the characters in the file will be random. The gradebook file has two parts where the first 16 bytes is the IV and everything after it is the encrypted gradebook data itself. The IV remains public because it does not provide any information about the gradebook to an attacker. It is necessary for retrieving and decrypting the rest of the file contents. I chose CBC mode because it is not deterministic when running encryption on the same plaintext and key. Each time an authorized user accesses the gradebook to modify its contents, the gradebook is decrypted with this IV and the key, which is an input string of hex digits but converted back to a

bytes. Once modifications have been made, gradebook add will generate a new random IV that is cryptographically strong via RAND_bytes() and entropy. Then, it will be used to encrypt the gradebook with the same key. When the gradebook is written back to the file, the new IV is concatenated with the encrypted gradebook data, so it can be stored securely on the cloud again. Since AES encryption uses a block size of 16 bytes, it will add padding to the plaintext in order to be a multiple of 16 bytes and fill a new block. In my program, I also had to account of this discrepancy when encrypting new data to output to file or reading the file's data into a buffer for retrieval and decryption. Thus, the ciphertext of the gradebook in the file on the server is actually a little bit longer that the actual size of the gradebook. The encrypt() and decrypt() functions are implemetned in data.c. In "setup.c", I use encryption to create a ciphertext of the initialized gradebook in line 88. In "gradebookadd.c", I utilized decryption in line 700 to retrieve the data, and use encryption in lines 734, 747, 761, 781, and 794 for each different action (add-student, add-assignment, etc.) that modifies the gradebook. In "gradebookdisplay.c", decryption is used in line 489 so that an authorized user, such as the ta or professor, can view student's information.

Vulnerability #2: Adversary uses user input to cause a buffer overflow or use formatted strings to read parts of the gradebook

An adversary could cause a buffer overflow by trying to get the gradebookdisplay or gradebookadd to accept user input and overwrite data outside buffer bounds. For example, if the attacker puts a string on the command line larger than the program has accommodated space for it, a strcpy() function will continue to copy until the NULL byte is reached. This may corrupt other data on the stack. My gradebook wrapper program aims to protect against these vulnerabilities by first checking if the sizes of input strings are valid for my predefined buffer sizes and copying them to these buffers using strncpy(). Since I use the opt library to read the command line arguments as options, all inputs to program will be null terminated strings. Input that represents a student's grade or an assignment points or weight are validated and converted to integer/float types using atoi() or atof(), so it stops reading any non-numeric characters. I also utilize isalpha() and isalnum() functions from ctype library to verify that student names are alphabetic characters (a-z, A-Z), assignment names are alphanumeric characters, and filenames are alphanumeric characters (including underscore and period). Furthermore, I check if the input key is a string of 32 hex characters. By only allowing certain

input characters, the gradebook cannot store format specifier strings like “%s” in the gradebook. This also defends against a buffer overread attack because executing printf() with a format specifier will cause it to look for other arguments on the stack, which are not provided. The only program that utilizes printf() is gradebookdisplay to output information from the gradebook. In the parse_commands() function in “gradebookadd.c”, I utilize a combination of these methods to protect against these vulnerabilities for each different action in lines 111-129, lines 140, lines 153-166, lines 176-182, lines 192-198, lines 211-224, lines 238-252, lines 261-275, and 284-298. Similarly, these protections are also implemented in mygradebook display in its parse_commands() function to check the validity of input strings from the command line.

Vulnerability #3: Adversary modifies the gradebook information without the key

Since the gradebook is on the insecure cloud server, an adversary can directly access and modify the gradebook file to change student’s information and assignment scores in the gradebook. If the attacker was a student in the class, he/she could try to increase his/her own grades for assignment or switch grades with other students in the class. This attacker may also change the weight and points for assignments. As a result, the ta or professor may never find out these changes have occurred when they download the file locally to modify/view data. To implement data integrity for my gradebook wrapper program, I utilize the SHA1 function from the OpenSSL library to create a 160-bit (20-byte) hash value. For each assignment, the function takes in a concatenated string consisting of the student’s first name and last name. Once I obtain the sha1 hash value of this string, I store it as a field inside the assignment. When a user wants to add or modify a grade for an existing student, the gradebookadd program will calculate the sha1 hash value of the inputted first and last name for a student. Then, this new hash value is compared to the stored hash value for the assignment that we are accessing grades for. If the student names were not tampered by an unknown user to change grades, then the compared hash values should be equal. In my gradebookadd program, I prevent these types of modifications from occurring and detect it if someone tries to make unauthorized changes in lines 444-457 in the add_Student() function and lines 643-665 in add_Grade() function. Since two students with the same first and last name are not allowed in the gradebook, two strings for different students should not hash to the same value in the gradebook. However, hashes are not the most cryptographically scheme due to the possibility of a collision. Thus, if I had more time to

implement this project, I would have implemented a more secure MAC algorithm for each student name and associated grade for the assignment instead of the sha1 hash value.

Vulnerability #4: Adversary causes a user running gradebookdisplay or gradebookadd to accept a fake gradebook file even though an authorized user, such as a ta or professor, did not modify it

Another vulnerability is that the adversary modifies the content of the gradebook file and it is still accepted by the wrapper program. An authorized user who has downloaded the file assumes that the last person who modified or views the files data was also an authorized user. However, they are oblivious that an adversary could have modified the file while it was stored in the cloud server. While I did not have time to implement this functionality in my program, I would have used a type of secure MAC algorithm to reach the goal of authenticated encryption. For the MAC algorithm, I would have generated another random key and stored it at the end of the gradebook file. In my gradebookadd program, once I have modified the gradebook, I will encrypt this data to obtain the ciphertext. Then, I run the MAC algorithm on the ciphertext with the MAC key to generate a tag. The IV for encryption/decryption, the ciphertext of the gradebook, the MAC tag, and the MAC key will all be stored in the gradebook file, which is uploaded to the server. Thus, authorized users can verify that this tag is correct and are guaranteed to know that the gradebook file was not modified on the server. If the tag is not accepted, then they know that an adversary must have modified the gradebook, and the file is now corrupt.