Project : Secure Log File

May 9, 2018

Enock Gansou

Kevin Wittner

Overview

In this project, we implement a secure log to describe the state of an art gallery: the guests and employees who have entered and left, and persons that are in rooms. The log will be used by two programs. One program, logappend, will append new information to this file, and the other, logread, will read from the file and display the state of the art gallery according to a given query over the log.

1. System design

In the following sections, we describe the overall design of our secure log file system.

1.1. Log file format

The unencrypted text in the log file has one line per employee or guest who has visited the art gallery. Therefore, the entire history of a single person in the gallery is recorded on a single line. The general format of a line in the log file is as follows:

[E or G] [Person's Name] [History]

The first character of each line is either 'E', signifying that this line corresponds to a person who is an employee, or 'G', meaning that the line is for a guest. This character is followed by a single space character, then the person's name, exactly as it is provided by the user. Another single space character separates the person's name from their entire history in the gallery. A person's history is composed of multiple entries. The format of each history entry is:

[1 or 0]R[Room ID]T[Timestamp];

The first character in an entry is a 1, meaning the person entered a room, or 0, meaning the person left a room. After the first character is the letter 'R', which is followed by the room ID that the person entered or left. If the room ID is -1, then this means the person entered or left the gallery. Finally, this is succeeded by the letter 'T' and the timestamp, which is always a nonnegative integer. Every entry except the last is terminated by a semi-color character.

1.2. Logappend Program

The logappend program is responsible for processing every input specified by a user. If the input violates any rules as defined by the project requirement, "invalid" is printed. Otherwise, the data is processed. To accomplish this, the program always makes sure of the validity of the user's input and then, it chooses between two different means of processing: file creation and file update.

During the file creation phase, the program creates a file as the file name is specified for the first time and the file does not exist. Since the input is already validated, the program must then

make sure this first task is that an employee or a guest enters the building (1R-1T...). Once this is verified, the format string as defined above is then created, encrypted, and saved to a file along with the TAG and a random IV. For example, let's assume we want employee John to enter the gallery at time 1. The corresponding content of the file will look like this:

[Encrypt("E John 1R-1T1\n")][TAG][Random IV]

During the file update phase, the program identifies that the file has already been created. It then decrypts the file to a string using the random IV of the encryption process and makes sure the TAG matches. If the TAG does not match, "invalid" is printed. The string must hold all information about any employee or guest who visited the gallery. Based on the data we obtained, we should evaluate whether an input is valid to be added.

As we update the data, we can either update an existing individual information or add a new individual information. Adding a new individual information follows the same rules as discussed previously. For any new guest or employee, he or she must enter the gallery first. Updating existing individual information is the one which requires more work. After the specific guest or employee is matched via a specified input, we obtain his or her most recent information in the history. Based on this information, we decide to update our data or "invalid" is printed. To update the data, four conditions need to be met. Assuming the current time is always bigger than the previous one saved, we have:

- If the individual previously entered the building, he can leave the building or enter a room.
- If the individual previously entered a room, he must leave the same room.
- If the individual previously left the building, he can only enter the building.
- If the individual previously left a room, he can leave the building or enter a room.

If an update is made, the data (string) is again encrypted using the same technique as discussed before with the TAG and the random IV and added back to the same file. For example, let's assume we want that John, from the previous example, enters room 0 at time 2. We also want a guest Jack to enter the gallery at time 3. The content of the file will then look like this:

[Encrypt("E John 1R-1T1;1R0T2\nG Jack 1R-1T3\n")][TAG][Random IV]

1.3. Logread Program

The logread program is responsible for reading the encrypted log file, decrypting it, processing the information in the log file, and displaying this information to the user. It accomplishes this by first parsing the command-line arguments, recording which flags were provided and recording any user input such as the key and employee or guest name.

Next, logread decrypts the entire log file. Decryption is done in GCM mode using 256-bit keys. To handle the decryption of large files, decryption of the ciphertext is done in blocks of 1024 bytes. Any integrity violations are also checked during decryption using a tag appended to the ciphertext.

If the -S command-line option is provided, then logread prints out the entire state of the log file. It does this by iterating through each line of the decrypted log file, recording information about

each employee or guest in the gallery. For each line, logread first grabs the last entry in that person's history and parses the room ID, timestamp, and whether the person entered or left the room. From this information, if the person hasn't left the gallery then they are added in sorted order to a linked list of employee or guest names. Furthermore, if their last entry is entering a room, then that person's name is added to a linked list of room nodes, where each node contains a room ID and a linked list of all the people in that room. These linked lists are printed in the correct format after the entire log file has been processed.

If the -R command-line option is provided, then logread finds the line in the log file corresponding to the person's name that was provided as input. Once it finds the person's history, it iterates through each entry in their history. If the entry corresponds to entering a room, then the room ID is printed out in the correct format.

If the -T command-line option is provided, logread finds the line in the file corresponding to the employee or guest that was provided, iterates through that person's history, and sums the total time the person spent in the gallery. This is done by keeping track of when a person entered and left the gallery, and subtracting the two times.

2. System security

This section describes five specific attacks that we considered when designing and implementing the secure log file system and the measures that our implementation takes to defend against the attacks.

2.1. Buffer Overflow attacks

A malicious user could attempt to break the system by providing very large inputs or writing large amounts of data to the log file in hopes of overflowing an in-memory buffer. To prevent this attack, in both logappend and logread we allocate memory for the decrypted log file text (on line 168 of logread) using malloc instead of a char array on the stack. The number of bytes to allocate for the buffer is determined by seeking to the end of the file and using ftell to return the file's size. By doing this, our program will always have a buffer large enough to store the entire log file.

Our system's encryption and decryption processes are also designed to mitigate against buffer overflows. The decrypt function in logappend and logread decrypts the ciphertext in blocks of 1024 bytes. Since this can be done an arbitrary number of times, a log file of any size can be decrypted by our system.

2.2. Integer Overflow attacks

A malicious user could provide very large numbers as input to our system in an attempt to cause incorrect behavior or a program crash. This would happen if the numbers or system is storing or computing exceed the maximum value that can fit in the data types of the variables we use.

To mitigate against this attack, in logread we always use the largest data type appropriate for the program's variables. For example, in the function print_time_spent, the variable time_spent (line 376) is of type unsigned long. Since the largest valid timestamp is 1,073,741,823 and the maximum value that can be represented by an unsigned long is 4,294,967,295 on a 32-bit machine, time_spent will always be able to correctly store the amount of time a person spent in the gallery. Another example of this is line 276 of logread, in which the variable used to store the length of a person's name is an unsigned long. Since the project description does not provide a limit for the size of a person's name, we use the largest data type possible to compute the person's name length.

2.3. Confidentiality attacks

For this project, confidentiality is one of, if not, the most important factor to consider in designing a secure log. To encrypt the message to the file, we use GCM mode which provides both integrity and confidentiality to our plaintext (more about integrity in 2.4). Yet, we think a good encryption/decryption is done if good parameters, namely the key and the iv, are passed into the encryption/decryption mechanisms.

To encrypt the plaintext, we make sure to use a random IV every time. We design a mechanism which provides a random IV of length 12 used for encryption. This IV is then appended to the end of our file to be used in the decryption process. This process guarantees a different ciphertext every time we encrypt since the IV is randomized. We therefore reinforce confidentiality by preventing an attacker to make assumption about the plaintext.

Moreover, we decided to reinforce the key provided in command line. Instead of just padding the key that was provided through command line, we decided to first append some "salt" to the key (which may be helpful while preventing dictionary attack). We then hash this string and choose the first 32 bytes to be our key. We realized that an attacker can use different keys as long as he makes sure the first 32 bytes are the similar if we directly use the key. Yet, different keys are "very likely" to provide different hash values ensuring, so hashing is safer to use.

2.4. Integrity attacks

Since users of the secure log file system have access to the log file itself, a malicious user could attempt to modify the log file without using logappend. If a malicious user was able to do this, then they could modify the data in the log file, causing logread to output incorrect information.

To defend against this attack, our system includes a message authentication code (MAC) tag with the encrypted log file. This is done in the encrypt function of logappend. After encrypting the plaintext using GCM mode, we use the OpenSSL EVP library to generate a tag (line 339) and write it to the end of the encrypted log file (line 342). The tag is then verified during the decryption process of logread, in the decrypt function. In this function, we set the tag on line 137 then use EVP_DecryptFinal_ex to verify that the decryption succeeded and the tag was valid. A return value of 0 indicates that the tag was invalid, which causes logread to print "integrity violation" and exit on line 202. By using OpenSSL's tagging features, we ensure that the log file has not been modified before logread reads from it.

2.5. Format string attacks

In logread, measures are taken to prevent an attacker from executing a format string vulnerability attack on our system. Every time a string is printed to standard out, a format string is always passed to printf to correctly display the results. The best example of this is on line 61 of logread.c in the function print_person_list. On this line, the current name is printed to standard out using the following call: printf("%s", current->name). This mitigates format string attacks because it is possible that a malicious user could provide a format string as a person's name when using logappend. In this case, if "%s" was not used (e.g. printf(current->name)), then the format string in current->name could potentially reveal undesired information about the log file to an attacker.