# CMSC 414
# PROJECT 4
# DOCUMENTATION

Manan Bhalodia
Joseph Nyangwechi

# Format of log File

1. First 32 bytes is the hashed token.
2. Second 32 bytes is the hashed plain text.
3. Next byte is a new line.
4. The rest of the bytes starting from the 66th byte are the encrypted text.

# Logappend

## Steps to Achieve a Valid Log File in Program

### 1. Parse options based on command line arguments using while loop

a. Variables stored outside the while loop so that cmd line arguments don't have to be given in a certain order

b. Variables

i. **Char ***Token - token passed in by user

ii. **Unsigned char** Md_value - hashed token value

iii. **Unsigned int** md _len - length returned by hash function after hashing token

iv. **Int** Timestamp - timestamp passed in by user

v. **Char** Ptype - person type: 'E' for employee or 'G' for guest

vi. **Char** Name - name of person entered by user

vii. **Int** Direction - direction of person: 1 for arrival or 2 for departure

viii. **Char** Dtype - Direction or either 'A' or 'L'

ix. **Int** Room - room the person enters, -1 if entering the gallery

### 2. Pick up positional argument for log path

a. If no log file is found -> create a new log file

i. Important Variables

1. **Char** Outstring - value of event string to be added to log
2. **Unsigned char** hash_value - hashed log value
3. **Int** sizetest - size of outstring

ii. Check if person is entering a gallery, if not then invalid

1. Room == -1 && direction == 1

iii. If entering gallery then prepare outstring

1. Use snprintf to get length of formatted output string, store in sizetest
2. Use malloc for outstring to allocate sizetest + 1 bytes of memory
3. Use sprintf to store formatted hash into outstring

iv. Write to file / hash log contents / encrypt text
1. Fseek to start of file
2. Fwrite from the start of the file, the hashed token (md_value)
3. Hash the outstring and store into hash_value and add null terminator at set value 32 since hashed value is always 32 bytes
4. Fwrite hash_value and add new line character
5. Create ciphertext by using cbc 128 bit encryption
6. Fwrite cipher text to log file

b. If existing log file is found
i. Go to beginning of file using fseek and then use fseek again to go to end of file and then use ftell to get size of entire file
ii. Malloc a array, buffer, with length of entire file; used later to get contents from
iii. Create two char arrays with size length buffer - 65 to just store everything after the two hash values
iv. Use one of the char arrays too fread length buffer - 65 bytes from the log file, this is the cipher text
v. Use decrypt with this ciphertext array to decrypt and store plaintext in other char array that was unused
vi. Fseek back to beginning of file and use fread to read entire file into buffer
vii. From the buffer get the first 32 bytes which is the hashed token and then store it
viii. Then use memcmp to compare this extracted hash token to the hash token that was created using user input
ix. If same hash then get the next 32 bytes which is the hashed log
x. Compare the hashed log to the current extracted log for integrity check by hashing the extraction and using memcmp
xi. If valid and hashes are the same then tokenize the plaintext log splitting by the new line character.
xii. Take the array at each pointer from the tokenized log ang tokenize again inside a while loop that stops when the first tokenized pointer reaches NULL
xiii. Inside the while there is another while loop which is used to get all the information such as timestamp, E or G, name etc. for every line
xiv. In this loop if a name that matches the user input name then the last event involving that person is stored.
xv. This stored event is then compared against the user input event to see if the user input event it valid.
xvi. If the user input event is valid then check if room id is present or not

xvii. Format user defined input based on format string and store in array called outstring

xviii. Now check if user defined direction of where the person is going in the gallery is valid by checking the last event of that person with that same name and employee or guest signal.

xix. If valid then place event in the end of the buffer that contains the old plaintext event logs and then encrypt and fwrite to the log file.

# LOGREAD

## Outline

- Parse user inputs
- Checks that the token given is valid by hashing the given token and comparing the hashed token with the stored hashed token. I hash using sha256.
- Then if token is valid, I use the token as the key  to decrypt the encrypted text stored in the file.
- I decrypt using aes-128-cbc encryption method for openssl.
- I then read the decrypted text and parse it to implement the all logread functions.
- The logread only implements the current state of the gallery or prints all the room a user has visited.
- To print the current state of the gallery.
    1. I get the decrypted plain text and read it line by line.
    2. I keep updating a data structure of linked nodes that keep track of the current state of the logfile.
    3. After reading all the lines. I print the current state of the gallery in the format specified in the project description.

# 4 considered attacks

**NOTES:

- Programs use crypto libraries from Opensll
- All hashing is done using SHA 256
- All encrypting / decrypting uses AES 128 CBC

# 1. Confidentiality

a. To break confidentiality an attacker can just view a log file that is outputted if it is in plaintext. The attacker could also use log read if there was no token stored in the log file that was used to match the user inputed token.
b. To combat this we first encrypted our log events and printed the ciphertext into the file. We also made sure that before we stored our secret token, we hashed this so that the attacker cannot easily get the value and decrypt using this value.
c. Lines in code:
    i. We encrypt the plaintext first if its a new file and are logging an event for the first time, this starts from line 199 and goes up to line 202.
    ii. logappend.c:
        1. If it's a file that has already been created, the lines of code where the encryption takes place varies based on the state of the log:
            a. Lines 462 to 467
            b. Lines 504 to 508
            c. Lines 557 to 561
            d. Lines 597 to 601
            e. Lines 634 to 638
        2. We decrypt the file at line 264 of logappend.c
        3. We hash the user defined token at line 88 and we compare it to the stored token in the file at line 254
    iii. Logread.c:
        1. We use the validate log function in the func.c file (line 211) to do the decryption of the log
d. NOTE: When we encrypt the contents of the log, we use a constant initialization vector which is bad practice as an attacker can see if the same log file was created.
    i. What we would have liked to do instead is to generate a random IV and store that random IV in the log file so that we can use it to decrypt the file and avoid having the same encrypted log files.

# 2. Authentication

a. To break authentication an attacker could just read the log file if it was stored in plain text and get the token and use the token to start either writing to the log or reading to the log
b. To stop this we hashed the token using openssl's sha256, before we put it in to the file and we use the user provided token and hash it to make sure it matches with the hashed token in the file.
c. Code in logappend.c: hashed at line 88 and compared it at line 254
d. Code in Logread.c:
    i. We validate the log file using a validate_token function we created that is stored in a helper file called func.c.

1. This function starts in line 194 of that file and used in line 169 of the logread.c file

## 3. Integrity

a. We also ensure integrity by checking whether file has been modified everytime we do logread or logappend. We ensure a file has not been changed by:
    i. Decrypting the encrypted part of the file i.e is all the bytes after the first 65 bytes of the file. After decrypting we obtain the plain text that has the log entries.
    ii. We hash the plain text and compare it against the hash of the plain text stored in the file. The hashed plain text stored in the file is 32 bytes long and stored in the file starting at 33rd byte to 64th byte.
    iii. If the two hashes match then we have verified that the plain text wasn't changed. If the two hashes don't match then the file was changed.
b. Lines of code
    i. validate_log function in line 172 of logread.c file.

## 4. Buffer Overflow

a. An attacker can implement a buffer overflow attack by providing inputs that are greater than the size of the allocated space for an array.
b. We prevent buffer overflow by ensuring that every time we write to a variable/buffer we specify the number of bytes that should be written to the buffer. We make sure that the number of bytes to be written to the buffer doesn't go beyond the size of the buffer.
c. For example in line 86, 109, 117 of logappend.c we use strncpy instead of strcpy.

## 5. Format String vulnerability

a. An attacker can exploit a format string vulnerability by providing a format string as the argument to the printf statement, doing so leaks the stack.
b. To ensure that this vulnerability has not been introduced in our code we made sure that all printf statements contain the same number of format specifiers as arguments.
c. Lines of code in logread.c file:
    i. 538
    ii. 554
    iii. 567