

## Assignment 4

Assigned: 12/11/2020

Due: 22/11/2020

### Access Control Logging

For this assignment, you will develop an access control logging system using the C programming language. The access control logging system will monitor and keep track of every file access and modification that occurs in the system. So, each file access or file modification will generate an entry in a log file. This log file will be inspected by a separate high privileged process.

You will use “LD\_PRELOAD”, which instructs the linker to bind symbols provided by a shared library before any other library. This way, you will override the C standard library functions that handle file accesses and modifications (fopen, fwrite) with your own versions in order to offer the extra functionality you are requested.

***Important: This assignment will be required for your next assignment (Assignment 5). This means that if you skip this assignment, to fully implement Assignment 5, you will have to implement both assignments 4 & 5.***

#### Step 1: Access Control Logging tool

As already discussed, you are requested to develop a shared library, named “logger.so”, that overrides the C standard I/O library using the LD\_PRELOAD. Specifically, your own versions of *fopen* and *fwrite* will collect and log the needed information for each file access, before continuing with the standard I/O operation. The log file should be named “file\_logging.log”. The log file must be stored somewhere, where it can be accessible by all users. Each log entry should contain the following information:

1. **UID:** The unique user ID assigned by the system to a user (hint: see *getuid()* function).
2. **File name:** The path and name of the accessed file.
3. **Date:** The date that the action occurred.
4. **Timestamp:** The time that the action occurred.
5. **Access type:** For *file creation*, the access type is 0. For *file open*, the access type is 1. For *file write*, the access type is 2.
6. **Is-action-denied flag:** This field reports if the action was denied to the user with no access privileges. It is 1 if the action was denied to the user, or 0 otherwise.

7. **File fingerprint:** The digital fingerprint of the file the time the event occurred. This digital fingerprint is the hash value of the file contents (hint: You can use the md5 hash functions: [https://www.openssl.org/docs/man1.1.0/man3/MD5\\_Init.html](https://www.openssl.org/docs/man1.1.0/man3/MD5_Init.html) ).

Events that must be logged:

1. File creation: Every time a user creates a file, the log file must be updated with information about the creation of the file. Make sure to modify the `fopen()` function in a way that the *creation* of a file can be distinguished from the *opening* of an existing file.
2. File opening: Every time a user tries to open a file, the log file must be updated with the corresponding file access attempt information. For this case, `fopen()` functions need to be intercepted and information about the user and the file access has to be collected.
3. File modification (write): Every time a user tries to modify a file, the log file will be updated with the corresponding file modification attempt information. This means that `fwrite()` functions need to be intercepted and information about the user and the file access has to be collected. Every `fopen()/fwrite()` function should create a new entry in a log file.

## Step 2: Access Control Log Monitoring tool

Develop a log monitoring tool, named “`acmonitor.c`”, which will be responsible for monitoring the logs created by the Access Control Logging tool (Step 1). This log monitoring tool will:

1. Parse the log file generated in Step 1 and extract all incidents where malicious users<sup>1</sup> tried to access multiple files without having permissions. As an output, the tool should print all users that tried to access more than 7 different files without having permissions.
2. Given a filename, the log monitoring tool should track and report all users that have accessed the specific file. By comparing the digital fingerprints/hash values, the log monitoring tool should check how many times the file was indeed modified. As an output, the log monitoring tool is expected to print a table with the number of times each user has modified it.

## Step 3: Test the Access Control Logging & Log Monitoring tools

Develop a simple tool, named “`test_aclog.c`”, that will be used to test and demonstrate the above tasks. The “`test_aclog.c`” tool has to create/open/modify files, in a way that will create the conditions that the

---

<sup>1</sup> For this assignment, a “malicious user” is the user that tries to access multiple files without having the permission. For Step 2, when we refer to a “malicious user” we refer to the user that tries to access more than 7 different files without having the permission.

“acmonitor.c” tool searches for. For instance, you should try to open files without having the permission to do so (Step 2.1), and modify specific files (Step 2.2).

Executing the “test\_aclog.c” tool with your custom fopen() and fwrite() functions preloaded, will create the required access control log file entries in “file\_logging.log”. Then, use the log monitoring tool to get the relevant reports (Step 2).

### Tool Specification

The Access Control Log Monitoring tool (Step 2) will receive the required arguments from the command line upon execution.

#### Options

-m	Prints malicious users
-i <filename>	Prints table of users that modified the file given and the number of modifications
-h	Help message

#### Notes

1. If no appropriate option was given, the Access Control Monitoring tool has to print the appropriate error message.
2. You need to create a Makefile to compile your library and programs (you must submit it with your source code) or you can use the one provided to you.
3. You are also provided with a corpus to build your own source code.
4. You need to create a README with your name, your login and a short description (1-2 lines) of your implementation.
5. You must submit the following files: README, Makefile, logger.c, logger.so, acmonitor.c, test\_aclog.c
6. You should place all these files in a folder named <AM>\_assign4 and then compress it as a .zip file. For example, if your login is 2020123456 the folder should be named 2020123456\_assign4 you should commit 2020123456\_assign4.zip.

7. The tool's corpus provided with this assignment is just an example. Feel free to define your own functions or change the signatures of the given functions (except for `fopen()` and `fwrite()`). You can even re-design it from scratch. However, the options defined in the "Tool specification" section must remain as-is.
8. Use the tab " $\Sigma \cup \xi \eta \tau \eta \sigma \eta$ " in courses for questions.

### Example

You need to use `LD_PRELOAD` to instruct the linker to load *your implementation* of `fopen/fwrite` before any other library. Example below:

```
#include <stdio.h>

int main()
{
    printf("Calling the fopen() function... \n");

    FILE* fd = fopen("test.txt","r");
    if (!fd) {
        printf("fopen() return NULL\n");
        return 1;
    }
    printf("fopen() succeeded\n");

    return 0;
}
```

Figure 1: Example code in file `main.c` that calls the `fopen()` function, checks if it returns a pointer to FILE value (successful call) or NULL (failed call), prints a message and returns.

```

#define _GNU_SOURCE

#include <stdio.h>
#include <dlfcn.h>

FILE* fopen(const char* path, const char* mode) {
    printf("In our own fopen, opening %s\n", path);

    FILE* (*original_fopen)(const char*, const char*);
    original_fopen = dlsym(RTLD_NEXT, "fopen");
    return (*original_fopen)(path, mode);
}

```

Figure 2: Our custom implementation of `fopen()`. This custom `fopen()` function prints a message and then calls the original `fopen()` from the C standard library using `dlsym` with the `RTLD_NEXT` flag (finds the next occurrence of a function in the search order after the current library).

```

$ gcc main.c -o main

$ ./main
Calling the fopen() function...
fopen() succeeded

```

Figure 3: By compiling and executing the `main.c` we get this output. Without the use of `LD_PRELOAD`, the original `fopen()` function from the C standard library is called.

```

$ gcc -Wall -fPIC -shared -o myfopen.so myfopen.c -ldl

$ LD_PRELOAD=./myfopen.so ./main
Calling the fopen() function...
In our own fopen, opening test.txt
fopen() succeeded

```

Figure 4: We compile the “`myfopen.c`” source file in order to get the “`myfopen.so`” file. To execute “`main`”, we first preload our custom `fopen` function using the following command “`LD_PRELOAD=./myfopen.so ./main`”. As occurs from the output, the message “`In our own fopen,...`” shows that the custom `fopen` was executed.