CMP408 - Mini Project

Modifying CMP408 Application with Spy System Calls

Name - Ethan Hastie

Student ID - 1801853

Table of Contents

1. Introduction

The separation of the operating system into user mode and kernel mode means there is a lower risk of system complications due to harmful processes being executed using a greater privilege level present in the protection ring security model of each system. It provides a higher level of stability and security within the operating system by segregating it into two modes of operation (IONOS, 2020). User mode possesses limited access to resources while kernel mode operates with full privilege and access to the kernel's resources. For programs to access resources offered by the operating system, system calls must be used to initiate requests to the kernel to gain access to resources. System calls are an important interface between the user and kernel modes initiated by user space programs when processes are needed to run in kernel mode. Common system calls include those in file management which are used to open, read, write, and close files. Other categories of system calls include process control allowing for the creation of new processes as well as the termination of these processes. System calls also facilitate communication between systems through networking allowing it to send network packets and receive them:

Figure 1 Examples of Types of System Calls (Pal, 2019)

The cat command on Linux uses four file management system calls, namely *open()* to open files, *read()* to read the content of the file, *write()* to write the and *close()* to close the file. If the *read()* system call were to be intercepted, it could execute malicious code every time that function is called as well as modify the existing information returned from it. The original function could also run alongside this modified call allowing it to spy on user's actions. Other valuable information such as files opened, and network data could be logged without the user's knowledge. This type of attack is called a rootkit, a form of malware, which is designed to bury itself within the computer and mask itself to prevent detection (GoldenOak, 2020). If system calls can be hijacked, then these malicious processes may go unnoticed. Using this method of attack, the victim's machine can be modified with a spy system call using an LKM. An LKM is a module that simply extends the running kernel and is a more efficient process than rebuilding and rebooting the kernel every time new functionality is implemented. This would allow a user to insert an LKM that overrides the address of the system call, modify this with a fake system call that performs an operation the user has specified so whenever that system call is invoked that code will be executed.

This project will involve the creation of a spy system call that logs network data on a Raspberry Pi device that is running the CMP408 demo – a simple application that allows users to control GPIO pins and values from a web browser – to spy on the application and log this data to an DynamoDB held in the AWS cloud. The remaining part of this report proceeds as follows: Section 1.1 details the objectives followed to meet the project aim, Section 2 examines the practical work undertaken for this project and finally Section 3 provides a discussion of the project.

1.1 Objectives

This project aims to demonstrate the usage of spy system calls within the CMP408 demo application. To accomplish the aim, the following objectives must be met:

- Create the malicious system call using an LKM to hijack the connect system call and log network data such as IP addresses and port numbers.
- Using the generated log data, filter the data and send this to a database hosted in AWS.

2. Methodology

2.1 CMP408 Application

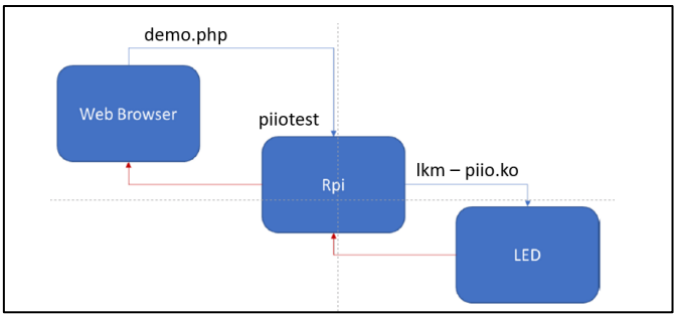

Figure 2 Application Flow of CMP408 Demo

The original CMP408 application used the GPIO pin 23 which could be set using the browser. This has been modified to include two more LED's that act as a traffic light system for when data is sent to the database in AWS. Modifying the Linux connect system call is based on research by Patel, 2008, which involved monitoring system calls using an LKM (Patel, 2008). In addition, it has been modified so that the traffic light LEDs cannot be set within the browser as well as appropriate implementation of input validation techniques on client and server side.

2.2 Modifying the Connect System Call

To write the modified *connect()* system call, the system call table within Linux that holds the all of the addresses of the different system calls needs to be located. Depending on if it's a 32-bit or 64-bit machine, system call addresses may be different. The kernel version of the Rasberry Pi used was Linux 5.4.51. In some earlier versions of the kernel, the address of the system call table was exported but was patched in recent versions because once an attacker obtained this symbol, they could perform harmful operations (GoldenOak, 2020). One method may be to brute force the memory range of the kernel that system call table belongs to. On 32-bit systems, this range is from *0xc0000000* to *0xd0000000*. An LKM can scan this entire memory range and return the address of the table, although this is only reliable if the system call table is exported (Patel, 2008). In this case, a special library was used to store the base of the system call table called *kallsyms.h*. Kernel versions before 5.7 are affected by a vulnerability that allows the system call table address to be used by utilising the *'kallsyms_lookup_name()'* function to return the address of the system call table in the *'kallsyms.h'* library. Upgrading the kernel version would render this method useless (TheXcellerator, 2021). In addition, the read/write status of the system call table should be changed to read only to prevent it being written to. This function can then be used to store the addresses of the other system calls and modify the system call table to point to the new code of the modified system call:

```
long this_will_be_fake_connect_call(*aargs) {
        # modified code here
        return real_connect_call(*aargs)  # optional
}
```

Figure 3 Pseudocode of Modified Connect System Call

In this case, the modified code accepts the arguments of the original system call and once the modified code has run, the original system call will be invoked. The connect system call is responsible for initiating a connection on a socket and accepts three arguments (man7.org, 2021). The modified code of the system call

calls two other system calls: getsockname() and getpeername(). Both perform the same function except the former retrieves the address of the source socket while the latter obtains the address of the peer socket. Along with this, user ID's as well as port numbers can be extracted can be logged to a log file. Once the module has been unloaded from the machine, the system call table is reverted to its previous state for safety.

2.3 Sending the Log File to AWS

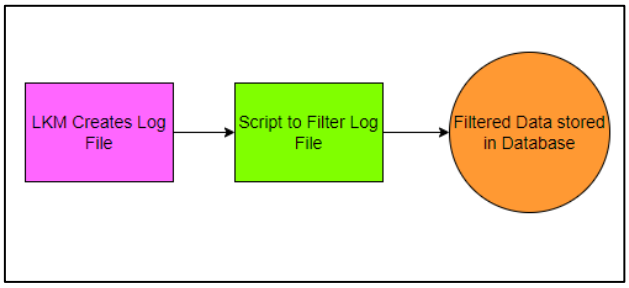

Figure 4 Process of Sending Data to DynamoDB

Once the LKM generates the network log files, the data can be extracted from the log file and placed in the database (Patel, 2008). The Python script extracts the entries in the log file such as IP addresses and port numbers and utilises AWS SDK Boto3 to send this data to DynamoDB. The script is modified to set the traffic lights on and off for when data is sent to AWS. Data held in DynamoDB is automatically encrypted at rest using AWS KMS. In addition, the Boto3 library uses secure HTTPS requests to interact with DynamoDB, so data is protected while in transit (Maksimov, 2021). When data is stored in the database, it can be viewed as shown below:

Table 1 Columns in Database

Table ID	User ID	System Call	Source IP Address	Source Port Number	Destination IP Address	Destination Port Number

3. Conclusion

The objectives of the project were achievable and by limiting the scope of the project to certain system calls it reinforces this. All aspects are addressed concerning the hardware, software, and cloud. The implementation of the hardware built upon the CMP408 demo by adding more LEDs to indicate AWS connectivity. The creation of the LKM to modify the system call was well considered and was the most integral feature of the project. The database setup within AWS using DynamoDB as a managed service rather than setting up a separate EC2 instance and hosting a database server on this platform, made it most efficient for the project. This approach minimised the setup involved in the cloud and for future operations can scale to store more data.

One issue during development was that the system calls did not extract the correct network data. It wasn't sure why this was happening so this issue should be fixed in the future. Modifications could be made to the software aspect because log files must be manually extracted and uploaded to DynamoDB. Improvements could be made to enhance the monitoring feature within the application by periodically uploading data to the database. The AWS element could be enhanced with the inclusion of a web portal, pulling data from DynamoDB, and displaying results generated from written queries. In addition, timestamps could be extracted when the system call is invoked allowing for more accurate monitoring and allowing log files to be organised according to timestamp. The list of system calls could also be expanded as well as including the modification of file management system calls, such as *read()* and *open()*.

4. References

Patel, B., 2008. *User Activities Monitoring System Using LKM.* California: California State University. [Online] Available at https://csu-csus.esploro.exlibrisgroup.com/esploro/outputs/graduate/User-activities-monitoring-system-using-LKM/99257830873801671

GoldenOak, 2020. *Linux Kernel Module Rootkit — Syscall Table Hijacking.* [Online]
Available at: https://infosecwriteups.com/linux-kernel-module-rootkit-syscall-table-hijacking-8f1bc0bd099c

IONOS, 2020. *System calls: What are system calls and why are they necessary?.* [Online]
Available at: https://www.ionos.com/digitalguide/server/know-how/what-are-system-calls/

man7.org, 2021. *syscalls(2) - Linux manual page.* [Online]
Available at: https://man7.org/linux/man-pages/man2/syscalls.2.html

Pal, T., 2019. *System Calls in Operating Systems - Simple Explanation.* [Online]
Available at: https://technobyte.org/system-calls-in-operating-systems-simple-explanation/

TheXcellerator, 2021. *Linux Rootkits: New Methods for Kernel 5.7+.* [Online]
Available at: https://xcellerator.github.io/posts/linux_rootkits_11/

5. Bibliography

Clarke, J. & Dhanjani, N., 2005. Intercepting System Calls. In: *Network Security Tools.* 1st ed. Sebastopol: O'Reilly, pp. Part 2, Section 7.2.

Joshi, A., 2019. *Python and DynamoDB.* [Online]
Available at: https://amit-joshi-79576.medium.com/python-and-dynamodb-aef04d6e6001

Maksimov, A., 2021. *Introduction to Boto3 Library.* [Online]
Available at: https://hands-on.cloud/introduction-to-boto3-library/

ruinedsec, 2013. *Intercepting System Calls and Dispatchers.* [Online]
Available at: https://ruinedsec.wordpress.com/2013/04/04/modifying-system-calls-dispatching-linux/

Salzman, B., Burian, M. & Pomerantz, O., 2003. Chapter 8: System Calls. In: *The Linux Kernel Module Programming Guide.* Grande Prairie: Peter Jay Salzman, pp. 50-54.

Trail of Bits Blog, 2019. *How to write a rootkit without really trying.* [Online]
Available at: https://blog.trailofbits.com/2019/01/17/how-to-write-a-rootkit-without-really-trying/

Wilinski, R., 2020. *DynamoDB Python Boto3 Query Cheat Sheet [14 Examples].* [Online]
Available at: https://dynobase.dev/dynamodb-python-with-boto3/

6. Appendix

```
pi@raspberrypi:~ $ hostnamectl
   Static hostname: raspberrypi
         Icon name: computer
        Machine ID: 50c23bc931bf42a4970095ea2aaf14e6
           Boot ID: eab4ebf848624fd6aca7b494bf9a4951
  Operating System: Raspbian GNU/Linux 10 (buster)
            Kernel: Linux 5.4.51+
      Architecture: arm
```

Figure 5 Kernel Version

```
pi@raspberrypi:~ $ sudo cat /proc/kallsyms | grep sys_call_table
c00091a4 T sys_call_table
```

Figure 6 Read/Write Status of System Call Table Demonstrating that it's Available to Use Anywhere with T Value Set

```
// store the original addresses of the system calls - for safety and
// invoking them with new functions
printk(KERN_INFO "storing original addresses of system calls\n");
orig_getsockname = (void *)sctable[__NR_getsockname];
orig_getpeername = (void *)sctable[__NR_getpeername];
orig_connect = (void *)sctable[__NR_connect];
getuid_call = (void *)sctable[__NR_getuid];
```

Figure 7 Grabbing the Original Addresses Held Within the System Call Table

```
// modify the system call table to original addresses that
// we took in module1_init()
printk(KERN_EMERG "Modifying System Call Table To Original Addresses\n");
sctable[__NR_getsockname] = orig_getsockname;
sctable[__NR_getpeername] = orig_getpeername;
sctable[__NR_connect] = orig_connect;
sctable[__NR_getuid] = getuid_call;

printk(KERN_EMERG "Results written to %s%s.\n", LOG_FILEPATH, LOG_FILENAME);
```

Figure 8 Reverting System Call Table to Original Dresses

```
/**
 * this will be the modified connect() system call
 * note: also runs the original
 */
asmlinkage int new_connect(int fd, struct sockaddr __user *buff1, int flag) {
```

Figure 9 Modified System Call Function

```c
/**
 * function to write the network logs
 */
int write_logfile(char *buff) {

    char f_path[120];
    strcpy(f_path, LOG_FILEPATH);
    strcat(f_path, LOG_FILENAME);

    struct file *fp_w;
    loff_t pos;
    int ret, count;
    mm_segment_t fs;

    // either create or append to file
    fp_w = filp_open(f_path, O_RDWR|O_CREAT|O_APPEND, 0644);
    if (IS_ERR(fp_w)) {
        printk(KERN_INFO "failed to open %s\n", LOG_FILENAME);
        ret = -ENODEV;
        goto out1;
    }
```

Figure 10 Function to Write Log File

```
pi@raspberrypi:~/Desktop/Assessment $ cat test.log
*1000*Connect*28.126.13.195*49177*143.33.127.22*49153
*1000*Connect*60.126.13.195*0*208.53.1.192*49153
*0*Connect*0.0.0.0*58202*156.126.13.195*55181
*0*Connect*48.109.202.208*49165*244.80.29.192*49933
*1000*Connect*0.0.0.0*0*96.80.11.207*53003
*1000*Connect*252.13.166.192*49178*52.126.13.195*55114
*0*Connect*165.64.0.0*8487*255.255.255.255*32767
*0*Connect*236.153.33.192*49933*36.126.13.195*49933
```

Figure 11 Data Held in Log File

```
# initialise GPIO pins for red and green leds
# green - GPIO7
# red - GPIO20
green_led = LED(7)
red_led = LED(20)

red_led.on()

print("Red LED ON\n")

# use existing table
dynamodb_client = boto3.resource("dynamodb")
table = dynamodb_client.Table('CMP408_Network_Logs')

# open log file
with open("test.log", "r") as logfile:
    for line in logfile:
        mylines.append(line)

logfile.close()

# get the data seperated by the * symbols
for line in mylines:
    if not line:
        continue

    columns = [col.strip() for col in line.split("*") if col]
    data.append(columns)
```

Figure 12 Python Script (send_aws.py)

```
for row in data:
    uid = row[0]
    sys_call = row[1]
    sip = row[2]
    sp = row[3]
    dip = row[4]
    dp = row[5]
    table.put_item(
        Item= {
            # primary key is random (1 - 999,999)
            'Table ID': random.randrange(1,999999),
            'UID': uid,
            'System Call': sys_call,
            'Source IP Address': sip,
            'Source Port Number': sp,
            'Destination IP Address': dip,
            'Destination Port Number': dp
        }
    )

red_led.off()

print("Red LED OFF\n")

sleep(5)

green_led.on()

print("Green LED ON\n")
print("Data Sent to Cloud\n")

sleep(5)

green_led.off()
```

Figure 13 Python Script (send_aws.py)

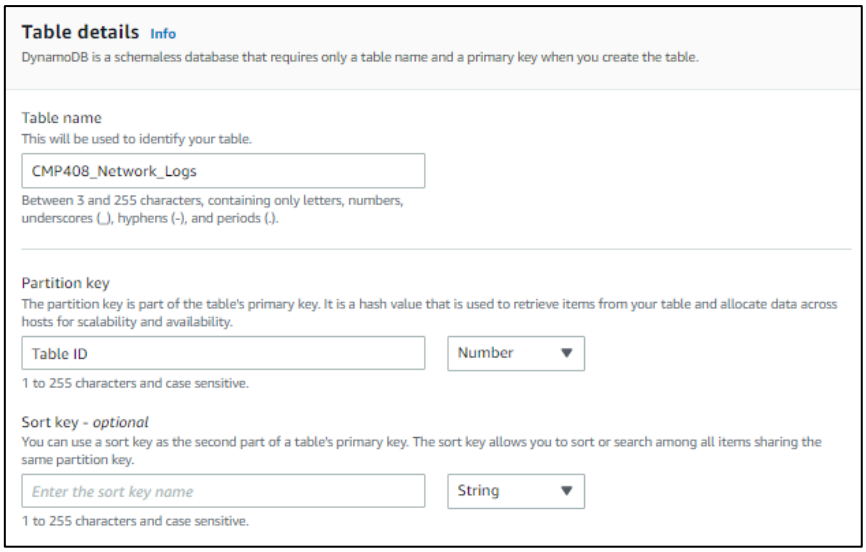

Figure 14 Table Details

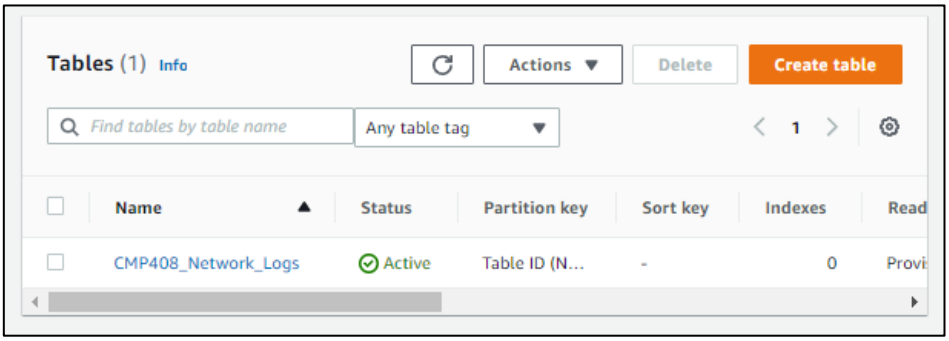

Figure 15 Created Table – CMP408_Network_Logs

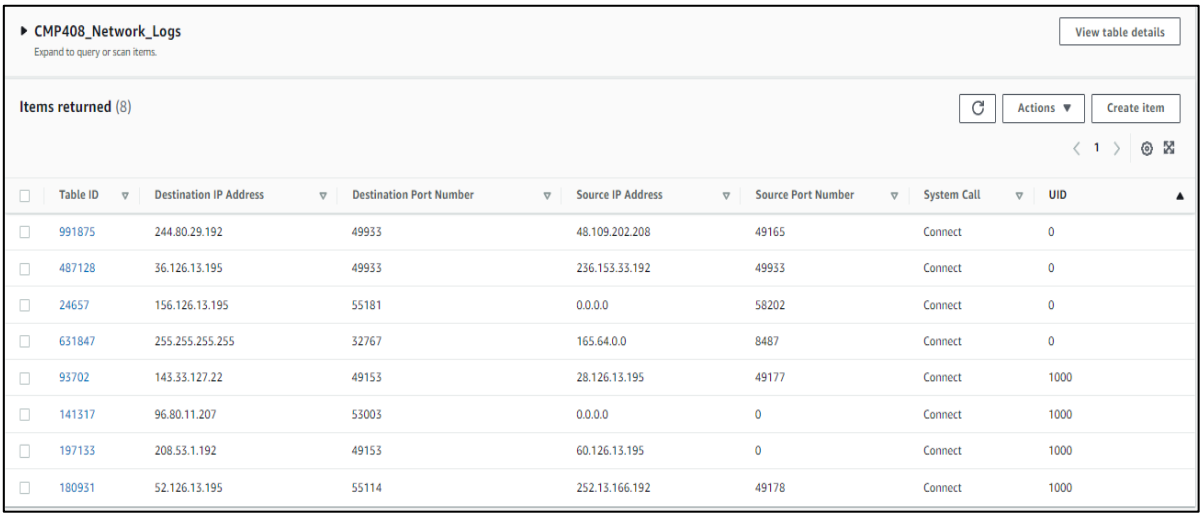

Figure 16 Data Stored within CMP408_Network_Logs Table