CMSC 421 Spring 2018

Project 2 Initial Design

Intrusion Detection System

The IDS needs to be able to identify any attacks that could harm the operating system. The initial step I would take is to keep track of all the system calls that a process is running, and that process is always being monitored. The Linux-based custom operating system must prevent an attacker from unauthorized access at any time.

I will first build a discriminator to distinguish acceptable and unacceptable behavior. The discriminator shall have a monitor which looks at the level of privileged processes. When the user invokes any processes that require privileges to run, the monitor will record all the requests as a log file. Even if it is in user mode, the IDS still needs to record potential changes that can be made to the privileged processes. This is to detect irregularities that could possibly allow an intruder to break into the kernel and cause harm to the disk. The monitor will first capture the sequences of the system calls. I will implement a function to measure anomalous behavior independent of the trace length based on the hamming distances between sequences. By determine the maximum hamming distance in which process can still be considered to be non-infected.

My implementation of calculating and compare the hamming distance would be something as follows:

```
//code written in c, then converted to kernel c after testing

//define structure to hold sequence data and user id

Struct sequence {

        unsigned int **seq;

        unsigned int id;

} sequence;

//When there is a new system call, a sequence is passed to the sequence table. If the sequence is good, the function will return -1. If there is something wrong with the hamming distance, the function will return other errors. The sequence size should be uniform throughout the whole table. Otherwise, an error should be returned.

unsigned long sequence_detector(unsigned int *new_sequence, unsigned int id)

{

        //check whether user id is correct
        If (this->id != current_uid().val){

                return -EPRM;

        }

        //calculate the hamming distance and compare it with the sequence passed in

        //add new sequence to sequence table

}

//When there is nothing in the sequence table, there is really nothing to compare with.
Instead, we initialize the sequence table with the sequence of the current system calls
being invoked. This initialization is also used to determine the sequence size that shall
be used throughout the whole table.

unsigned long sequence_init(unsigned int *sequence, unsigned int id)

{

        this->id = id;

        if (this->seq == NULL) {

                sequence **temp = malloc(10 * sizeof(sequence));

                temp[0] = sequence;

        }

        Else {

                Return -1;

        }

}
```

The implementation above is to instrument the system call dispatcher of the Linux kernel with code that logs each time a system call is made. The log entries should contain the process ID and the system call number at a minimum, we may include any other information that you deem appropriate, such as user/group IDs, timestamps, etc.

Then, I will be making a separate user-space process that runs in the background and monitors these logs, checking for abnormalities in them. At a minimum, my user-space process should construct a bit array for each process under monitoring showing which system calls have been run in a window of the last k system calls where k is a constant of my choosing – which should be chosen to demonstrate a working system. If a particular system call is made in the window, then I should set the bit for that system call to 1. Any system calls not made in that window would then have values of 0.

//I will then build a monitor to examine each system call that gets passed in. The function monitor will look through each system call and check its validity. If the system call is made in the window, set 1. If the system call is not made in the window, set 0.

```
unsigned long monitor(char *sys_calls, unsigned int id)

{

        //for instance, system calls are O,R,M,W,C

        Int I;

        Unsigned int *sequence;

        for (i=o; i<sizeof(sys_calls); i++) {

                if (!strcmp("//some system call",sys_calls[i]) {

                        sequence[i] = 1;

                };
                else {

                        Sequence[i] = 0;

                }

        }

        If (sequence_init(sequence, id) == -1) {

                sequence_detector(sequence,id);

        }

}
```

The functions above are basic structures for my discriminator. There could be other functions that need to be in order to satisfy the requirements and maximum the performance.