CMSC 421 Spring 2018

Project 2 Final Design

Introduction

In Linux, the Intrusion Detection Scheme is a patch to the Linux kernel and used to enhance the kernel's security. In our project, we need to create an IDS that keeps monitoring the system calls made by a certain process. We will keep a log file of the system calls the process makes within a certain period. The IDS needs to be able to identify any attacks that could harm the operating system. The first step I take is to keep track of all the system calls that a process is running, and that process is always being monitored. However, storing all the system calls would be very inefficient as it takes up too much space and time to do so. Therefore, the program should instead detect a sequence of system calls that are often used by a process. This sequence is then used to compare with any new system calls made by the same process. As a result, the Linux-based custom operating system must prevent an attacker from unauthorized access at any time by monitoring what system calls are made during each process.

Implementation

I will build a discriminator to distinguish acceptable and unacceptable behavior. The discriminator shall have a monitor and a detector, where the detector's job is to detect the sequence of system calls that a process is making and the monitor is to keep monitoring the newer system calls the process is making. The detector looks at the level of privileged processes which is done in the kernel space. 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 kernel mode, the IDS still needs to record potential changes that can be made to the privileged processes done by the root user. 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 determining the maximum hamming distance in which process can still be non-infected.

Before implementing the functions in kernel space and user space, I have created four different global variables in the kernel space using extern. They are int user_pid, unsigned long *syscall_array, int pid_position, and int log_switch. There are 5 system calls implemented in detector.c in the kernel space:

Asmlinkage void setPID(int pid);

//This system call gets the pid from the user space and puts it in the global variable user_pid for future comparison with the current kernel process pid

Asmlinkage int getPID(void);

//This system call returns the pid that the user has entered. This is important because in user space we want to keep track of the same process (pid), meaning that we have to be constantly making sure that the user pid is the same and not altered.

Asmlinkage void sysarray_init(void);

//This system call initializes the array that stores all system calls made by the process by dynamically allocating memory in the kernel space. It initializes current position to 0.

Asmlinkage unsigned long *get_sysarray(void);

//This system call obtains the system call array for logging purposes.

Asmlinkage void set_switch(int on_off);

//This system call gets the switch signal from user space and turns on or off storing system calls made by the same process as desired.

Asmlinkage void num_syscalls(void);

//This system call returns the total number of system calls invoked by the current process we are looking at. This will help to define the window size needed in user space to monitor the system calls.

The detector module in kernel space consists of multiple different system calls that are functions used to set or use the pid from the userland. The user pid is passed to the kernel to compare with the pid running on the current process. There are two system calls involve in this process. They are setPID and getPID. The system call setPID obtains the pid input chosen by the user from the userland and sets it to the global variable user_pid. Then, in common.c (/arch/x86/entry), I include the <asm/current.h> to get the current process pid. And, I will include an if statement to only compare the system calls made by the current process if the current process pid matches the pid entered by user. If the pid matches, the system call will be stored in the syscall_array. Before starting storing the system calls, I need to make a change from a conditional branch to an unconditional one at line 250 in entry_64.S, so that the assembly code used for responding to system calls makes a call into the C code. By doing this, the system calls can be constantly updated corresponding to the current process in common.c. And, the log_switch is a signal coming from the userland to decide whether or not the array should be continuing storing system calls. The switch can absolutely resume when it wants to, and the decision is made by the user in userland. The global variable pid_position is used to keep track of how many system calls the process has made and can be a good indicator of the current position in the system calls log. When having all these variables needed to monitor the process, the monitoring module can be done in user space.

The monitor C program in user space will start off prompting the user to enter the desired process by displaying all the available processes at current time. The monitor will then start making a setPID system call to set the desired user pid, and the kernel space will start capturing system calls created by the current process. The kernel space will make sure that system calls only made by this desired process will be stored. In monitor.c, In user space, procfs is used to create virtual files to log the system call sequences. And, when a desired length is reached the first sequence will be stored as an uninfected sample. Then, the log file will be passed to user space and used to compare with subsequent system call sequences. So, in monitor.c, everything could theoretically be done in just one main function as it is in user space. There are several parts needed to complete this task. First, there would be a section of code responsible for getting the user input for the switch signal and the pid needed to track. Then, the following section will be making the system call setPID in order to pass that pid information to the kernel space. Once that is done, the procedure in kernel space will be the same as described before. As the user space program starts, it will keep monitoring the same process. If the user turns off the monitor, the kernel space should immediately stop capturing system calls of that specific process. In case there is something unusual occurred in the system call sequence, the monitor module will identify it and kill the process, which is the main purpose of have an IDS. The monitor module shall select about 2000 system calls as a sample log file and use it to determine whether or not the newer system call sequence(s) is/are valid. As procfs offers virtual log files, there will be no “physical” text files to be saved in the directory.