Sabbir Ahmed

CMSC 421: Project 2 Final Design Document

May 13, 2018

1 Introduction

This project implements a new version of the Linux kernel that adds functionality to support a simple intrusion detection system (IDS). This system will operate by logging the system calls made by a process in the kernel, while analysis and intrusion detection will be done in user space. This assignment is designed to teach a simple method of intrusion detection, as well as to reinforce the idea of how user space and kernel space interact through the use of system calls.

An intrusion detection system is a computer program that attempts to identify (and thwart) attacks that might be performed on the system by attackers. There are several time-tested approaches to the development of an IDS. The project will keep track of the system calls made by a monitored process and check for abnormalities in the sequences of system calls made. When an attacker breaks into a process, they will need to make system calls in order to attempt to access the resources of the system that are under attack. As the system calls that the attacker will perform will likely be different than those performed by a process that is not under attack, it follows that by monitoring both healthy and broken processes, it is possible to develop a scheme to identify those that might be under attack for further action to be taken.

2 Objective

The project will compare sequences of system calls made by a monitored process to known good sequences using the hamming distances between them.

2.1 Kernel Space Requirements

The kernel-space program will instrument the system call dispatcher of the Linux kernel with code that logs each time a system call is made. Built-in system calls such as `ptrace` are not allowed to trace the usage of system calls to generate the logs for the project.

2.2 User Space Requirements

The analysis of the logs will be handled by the user-space program, which may be implemented in any supported programming language. The 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. If a particular system call is made in the window, the bit for that system call will be set to 1. The bit arrays will then be measured for their hamming distance with the example of a healthy system call sequence for a process.

3 Design Approach

The final design of the project adopted a much simpler approach than what was previously assumed. The methods to achieve the requirements became apparent after extensive research.

3.1 Kernel Space

The kernel-space program will trace the system calls and dump them on logs consisting of the `pid` and the system call numbers of each processes spawned by a program. The log was intended to also contain the corresponding timestamps of the execution of the system calls, but the kernel-space did not appear to have nanosecond resolution of their timestamps. Initially, the kernel-space program intended to create the log files in N chunks specified by the user to pass on to the user- space program for analysis. This parameter will be removed as an option to the user and instead be converted to a constant. The constant N number of system calls per log files would provide a smooth communication method with the user-space program.

Initially, `ptrace` appeared to be the only method of tracing system calls. Two different approaches were proposed

in the initial design. The first approach suggested inserting breakpoints in the implementations for each system calls in the kernel. The second and more probable approach suggested a stripped down version of `ptrace` that behaved identically. After extensive research, a third approach was considered that merged aspects of both of the prior approaches. The conditional jump instruction, `jnz`, was updated to an unconditional jump, `jmp`, inside the system call entry point section in `entry_64.S`. This update forces all the system calls to save their system call number to registers before disabling the interrupt request. The system call number is then accessible in the register `ORIG_RAX` in `common.c` which can be provided as a macro to external files.

The kernel-space program will require the user to provide the programs to be traced. The program will fork its own copy of the process, and gather the process IDs and the system call numbers. The `proc` file system will be utilized to provide a convenient interface between the kernels- space program and writing log files accessible to the user space. The common directory shared by the entire IDS, `idsdata`, will be used to store the log files.

3.2 User Space

The user-space program was developed before the kernel-space program. The program was tested using dummy data simulating log files created by the kernel-space program. An object oriented approach was taken to develop the interface with the user and the kernel-space logger. The user would be required to initialize the user-space program after the kernel-space program has been initiated to run concurrently. The user-space program would wait for the logger to generate a chunk of the system call logs, and then automatically label the first sequences of system calls as “healthy” sequences. The healthy sequences get stored in a separate file so that it may be read back into memory for analysis of the specified program at a later time.

The user-space program is written in Python 2. The user-space program executes a C wrapper for the kernel-space system call. The program provides feature options via the command line to the user. The options allow the users to utilize the the detection system for the list of programs, window sizes or other analysis parameters as needed.

4 Implementation

Not all the requirements of the project were completed. The implementation for the user-space program mostly followed the design approach initially planned. Portions of the program were required to be modified due to the incomplete kernel-space program it accompanied. Development for the kernel-space program involved programming in user-space for simulation in the same programming language (C). Once sufficient progress was made in the user-space version, the source code would be copied over and translated into the kernel-space program with kernel libraries. This method proved unsuccessful, since the translation process did not have enough time as the development neared the deadline.

The only requirement successfully implemented in the kernel-space program was creating a reference to the current system call in the architecture's register. The system call is labeled `sys_ids_log`. The kernel compiles and packages successfully using the kernel-space program.

The other requirements that were not successfully implemented in the kernel-space were forking processes from the programs listed by the user to trace, and writing to special filesystems using `proc` instead of normal filesystems. These requirements were implemented in the C wrapper user-space program which invoked the `sys_ids_log` system call.

The user-space program, labeled `ids.py`, therefore had to modify its design approach to compensate for the missing functionalities in the kernel-space. The initial design approach suggested both the kernel and user-space programs utilize the log files concurrently with mutexes to prevent any reader-writer issues. The current version of the `ids.py` design invokes the `sys_ids_log` system call before analyzing the sequences, removing the need for synchronization locks. The current version of `ids.py` also serves as the convenient high-level interface to the system call from the user.

4.1 Usage

To use the IDS, the user-space program, `ids.py`, provides various options for tracing processes. The current version of `ids.py` does not support tracing processes by their process identifier (PID). The user must provide the

processes to be traced as command line arguments. Additional optional features are provided to the user:

- The window size for the sequences of system calls to be analyzed for their hamming distances is also provided as an option to the user. The default value for the parameter is $k = 5$
- The minimum threshold for the hamming distance at which the IDS will mark a sequence as "unhealthy" is set to the default value of $t = 3$. The feature is provided to the user via `ids.py`
- The user also has the option to trace their processes from any working directory, given the absolute path l provided is valid

The command line arguments of `ids.py` are as follows:

```
python2 ids.py -p=<PROC1> -p=<PROC2> ... -p=<PROCN> [-k=K] [-t=T] [-l=PATH]
```

An example usage of the IDS with simple parameters are as follows:

```
python2 ids.py -p="ls -l" -p="pwd" -k=5 -t=4
```