Sabbir Ahmed

CMSC 421: Project 2 Initial Design Document
April 15, 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.

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

Development in the preliminary stages began with research into the topics involved in the project. Understanding the objectives and scope of the project to identify the requirements was the initial step taken for the project.

The next step in development involved outlining the list of tasks necessary to complete the project within the milestone deadlines. After considering several approaches, it was decided that the user-space program will be completed and tested for its functionality first, concurrently while conducting research on the Linux system call dispatcher.

3.1 Kernel Space

Initially, the logging of the system calls were perceived as a non-trivial but moderately difficult task that would not require much time committed. After preliminary research, it was evident that without the built-in system calls such as `ptrace`, logging system calls would require much greater effort.

The kernel-space program will trace the system calls and dump them on a log consisting of the `pid`, the system call numbers, and their corresponding timestamps.

Due to the extensive research required to build the kernel-level logging system, the development of its counterpart user-space program would be completed first. Research on plausible methods to achieve the logging of the system calls would resume simultaneously.

Two approaches are currently being considered to influence the flow of the research. Regardless of the approach chosen to develop the functionality in the kernel space, extensive research on the system call dispatcher is required.

3.1.1 Trivially Insert Breakpoints to the System Call Implementations

A potential but not very likely approach would involve fully understanding the implementation of all the system calls that would be used in the logging system. Once their implementation is understood, breakpoints would be added to suitable areas to pause the system call while it indicates to the log of its activity. This approach was being considered seriously, until it was apparent that it would not be feasible with the number of system calls involved. This would also increase complications when attempting to bind system calls. Adding breakpoints may involve modifying numerous assembly files, such as inserting `jmp` instructions to the writing system calls to write to the log.

Even though this approach appears very unlikely, it is not being completely ruled out because of the potential uses of parts of its ideology, such as using the system calls table to identify implementations of each system calls.

3.1.2 Reverse Engineer and Create A Stripped-Down Version of `ptrace`

The more probable approach involves fully understanding `<linux/ptrace.h>` and simulate it for the project. The built-in system call offers several functionalities that may not be required for the logging system. Creating a stripped-down version of the system call would most likely allow the system call logging functionality for the IDS.

3.2 User Space

The user-space program will be implemented in Python 2 without any third-party libraries. Time-permitting, the implementation will be translated to C to trade off the lines of codes for speed.

The program will continually parse the log file shared with the kernel- space program while properly utilizing mutex locks to avoid any synchronization issues. The user-space program will begin parsing the logs only after it has been closed by the kernel-space program. Once the lines of system calls for a process are properly parsed, they will be converted to their binary vector representations. The binary vectors will then be analyzed for their hamming distances, and the results will be logged in a separate file.

Additional Python 2 and shell scripts will be added to assist in properly formatting the outputs and managing the process as it runs in the background.