Name: Trevor Philip
Student Number: NL10252
Course: CSMC 421, Spring 2018
Date: 5/9/2018

Final Design Documentation for Project 2

Disclaimer: A few things changed from the initial design, but overall, the idea is similar.

Introduction

The purpose of this project is to develop an intrusion detection system for the Linux kernel. An intrusion detection system, or IDS for short, is a system by which the sequence of system calls, recorded and logged by the kernel, are analyzed and abnormalities in the sequences of these calls are detected. If an abnormality is detected, then a process can be declared as being under attack. For the userland IDS, two components are required: a logger and a log-analyzer. For the kernel, three new system calls will be required for toggling the kernel in-memory logging on and off and reading that memory to userspace.

The kernel toggler will require additional code to be added to the Linux kernel (specifically in the system call dispatcher) to log system calls in kernel space, and the analyzer for the logs of the system calls will be done as a separate user space program running in the background.

The choice of programming language for the kernel space program will be C, using the C90 standard, as this is the language and standard used by the Linux kernel as a whole and will need to be compiled with the kernel. The user space program will be written in C#, and will require the .NET Core Runtime to function. [1]

Statement of Goals

- Create a kernel-space component to be added to the Linux kernel that monitors all processes and creates in-memory logs for them with a maximum window size of 10. The logging system should be toggled and controlled via system calls from user space, and also be read via a system call from user space (the userspace program responsible for this must have root user privileges).
- Create two userspace programs:

[1] https://www.microsoft.com/net/download/linux

    - ids_logger: This is a program written in C that executes the custom system calls added to the kernel and log files to disc.
    - ids: This is a program written in C# and runs under the .NET Core runtime. This program analyzes logs, builds databases of valid system calls, and performs intrusion detection
- Provide examples of good and bad logs to show what would pass intrusion detection and what would not.
- Provide the appropriate makefiles for all parts of the system to ensure easy building and execution of the IDS system.

Kernel Space

In order for the intrusion detection system to work, system calls will have to be logged. Since system calls are kernel space components, the code for that will have to be written in kernel space.

In general, the goal for the kernel-space code is to keep the logging as simple as possible and with minimal changes needed to the existing kernel sources, so as to minimize the potential for kernel panics. The approach that is followed for the kernel space component of the IDS will be as follows:

- All system calls must proceed through the 64-bit "slowpath". To achieve this, the following modifications are needed to existing kernel sources:
    - The file **/arch/x86/entry/entry_64.S** must have the following line added to line 251:
        - jmp entry_SYSCALL64_slow_path

      This will cause all system calls to proceed to a function defined in **common.c** called **do_syscall_64**. The following header file must be included (described later):
        - #include "../../../../p2ids/kernel/toggler.h"

      In addition to that, the following must be added on line 277 of the file:
        - do_logging(nr);
    - Secondly, three new system call definitions are required in the syscall table:

      333 common sys_ids_log_on sys_ids_log_on
      334 common sys_ids_log_off sys_ids_log_off
      335 common sys_ids_log_read sys_ids_log_read
    - Similarily, the following lines must be added to **include/linux/syscalls.**h:

      asmlinkage long sys_ids_log_on(unsigned int process_id);
      asmlinkage long sys_ids_log_off(unsigned int process_id);
      asmlinkage long sys_ids_log_read(unsigned int process_id, unsigned char *log_data);

- Now that the existing kernel sources have the required files changed, the following new kernel-space source files are contained under **p2ids/kernel**:
    - **addsyscalls.c**: This contains the implementation of the three new system calls, all of which can only be called by a root user. Specifically,
        - sys_ids_log_off(unsigned int process_id): This turns off system call logging for a given process_id. Reutrns 0 on success, or some negative value on failure.
        - sys_ids_log_on(unsigned int process_id): this turns on the system call logging for a given process_id. Reutrns 0 on success, or some negative value on failure.
        - long sys_ids_log_read(unsigned int process_id, unsigned char *log_data): This is where the userspace system call logger (described later) must read the logs from memory to disk. The process_id is the process ID that is being tracked, and log_data is the userspace pointer that must have already been malloc'd to accept 100 bytes of data. This is an example of "window" data that gets copied to this pointer:

          4706,16,16,16,1,9,10,9,10,9,101,-1

          This indicates that process with ID 4706 has made syscall with ID 16 three times, then syscall 1, then 9, then 10, etc. If the space is occupied with a -1, it means that no system call was made further in that window. -1 is a placeholder since the window size of 10 is statically defined per tracked process.

          Once the memory has been read to user space, this is in-memory log is cleared and all values are set to "-1" again, except for the process ID of course.

    - **toggler.h**: This is simply the header definition file for the logger (which was called "toggler" because it toggles system call tracking on and off). It contains a custom-defined data structure like so:

```
typedef struct tracked_process {
    /* the process ID as an unsigned int */
    unsigned int pid;
    /* flag for turning the logging on the process on or off */
    bool is_on;
    /*number of syscalls in syscalls[]. Highest is 10*/
    int syscall_len;
    /* array of syscalls of max length 10 */
    int syscalls[IDS_MAXLEN];
```

```
    /* pointer to the next process in list */
    struct tracked_process *next;

} tracked_process;
```

This data structure defines a process that is being tracked. It is stored in a linked list (which was also custom implemented because I couldn't figure out how to get the kernel's linked list working correctly).

The most important part of this struct is the array syscalls[]. This is the array of system calls that were called in a given window size (currently defined as 10 in the implementation of this project). If no system calls are made, all values in this array must be initialized to -1. Then, when a system call is made, the first "-1" must be replaced with the actual system call number. If no more spots with "-1" in them are left, the system call is discarded.

When the userspace program calls the system call defined earlier to read this log, this log must be cleared to -1 again.

Other than that, this header file contains some function definitions:

- toggle_ids_logger(bool value, unsigned int process_id): toggles the IDS logger on or off, given the value "true" for on and "false" for off. If false is passed, the process_id is searched for in the linked list and remove.d If "true", the process_id is added to the linked list if it was not already there. If it was already added, nothing happens.

- is_ids_logger_on(void): simply returns true if the IDS has been toggled at all for any process, or if any process is being tracked.

- tracked_process * get_node(unsigned int process_id): gets the tracked_process struct by process_id if it exists in the linked list of tracked processes. Returns NULL if nothing is found.

- do_logging(unsigned long syscall_id): This is an important function that will attempt to log the system called to the "syscalls" part of the tracked_process struct if the CURRENT process that called that system call is in the list of tracked processes. Will just return if the CURRENT process is not being tracked.

- **toggler.c**: This file contains the implementation details for toggler.h. Nothing terribly interesting here, it is basically a fancy (or simple) linked list of tracked processes that manages itself. As long as the rules of toggler.h are followed (the comments should be obvious too), the IDS logging should work fine.

User Space

The userspace program consists of two parts: ids_logger and ids. The former is a C program that must be run as root user that makes calls to the new system calls that were added described in the previous section. Its primary function is telling the kernel to track/untrack process IDs, and to read the in-memory logs provided by the kernel and write them to a log file on sic. The latter is a program written in C# that runs in the .NET Core runtime. This program contains the logic for creating databases of valid system calls based on logs provided by ids_logger, and also analyzing logs and comparing them against the database of valid system calls for a given program.

- ids_logger: This is a simple C program that, as described, interfaces with the kernel to create logs for system calls. It must run as root user. It needs to accept three basic commands:
    - ./ids_logger track <pid>
        - Example: ./ids_logger 123
        - <pid> -- this is the process ID. This makes a call to system call sys_ids_log_on and passes the process ID. If 0 is returned, we are successful. Otherwise, the demsg log must be consulted to figure out what the errors are if any.
    - ./ids_logger untrack <pid>
        - Example: ./ids_logger 123
        - <pid> -- this is the process ID. This makes a call to system call sys_ids_log_off and passes the process ID. If 0 is returned, we are successful. Otherwise, the demsg log must be consulted to figure out what the errors are if any.
    - ./ids_logger log <pid> <msec>
        - Example: ./ids_logger 123 150
        - Makes a call to sys_ids_log_read and passes the process ID. The msec parameter is just the time in milliseconds between each successive querying of the kernel. 150msec is the default, and if not provided, will be assumed.
        - Some notes about this function: When it reads the in-memory logs from the kernel, it will attempt to resolve the actual executable binary from /proc/<pid>/exe, which is a symblink, and write that to the actual log. If it cannot resolve this (likely because the process has closed too quickly), the “ids” described below will ask you to specify what program was running. A good example of this would be “ls”. It is possible to get the process ID of ls and track it but the logger cannot easily catch ls before it loses its symblink in /proc. When training the IDS or analyzing logs, the user should specify to the IDS what program was running.

        - If the kernel returns a string containing -1,-1,-1,-1,-1,-1,-1,-1,-1,-1, no log file is created since this means no additional system calls were made at the time the kernel was queried.
        - The log file is simply named as the UNIX timestamp in milliseconds with a .log appended to it.
    - Although not terribly sophisticated, if the user forgets what process IDs they added or removed for tracking, they can consult the dmesg log. The system calls in the kernel designed for this IDS does log its actions there.
- **ids:** This is the Intrusion Detection System itself, which is located under userland/ids. In this directory, there is a Visual Studio solution file (.sln) and in the subdirectory "IntrusionDetectionSystem", there is a C# project file (.csproj). This file has one dependency: Newtonsoft.JSON. This is a popular JSON library for C# that lets one encode and decode JSON files to and from C# class objects. When the IDS is loaded for the first time, .NET Core will resolve this dependency from a network of packages called Nuget.
- This project is the brain of the intrusion detection system. Attempts were made to follow the paper "Intrusion Detection using Sequences of System Calls" by Hofmeyr et al. (1998). In order for the intrusion detection system to work, it must first build its own database using logs that are known to come from programs that are safe.
    - Upon first running of the IDS, a database of known system calls is loaded into memory. This database is defined as a JSON file called syscalls.json (it was converted directly from the syscall table in the Linux kernel). If syscalls.JSON does not exist, this is an error and will be treated as such. The required directories are also created if they don't exist, as well as the config.txt file, which stores the path log directory.
    - The IDS program has 4 basic functions: training, analyzing, changing log directory, and exiting gracefully.
    - The general algorithm for "training" the IDS is as follows:
        - All files present in the log directory that follow the *.log wildcard are loaded into an array, and each one is iterated over.
        - The log is read to a string and parsed. Commas are exploded to an array of size 12 (if the array is not size 12, the program will throw an exception since this is an assertion failure).
        - The first entry in the log denotes the location of the executable binary of the program. If this follows the /proc/<pid>/exe pattern, this means that the ids_logger was not able to resolve it. The user is promoted to state which program this was (it is very important that the user is consistent with this, see next bullet point for why).
        - The string denoting the location or name of the executable binary is hashed to an MD5 string. This will be the "key" in the IDS database.

This ensures we are always treating the exact program in the exact same way.

- If the database does not exist, it is created, and it won't have any sequence entries. A database entry is simply the ID of the program (the MD5 hash) and a dictionary containing valid sequences of system calls for a window size K of size 5. Hofmeyer et al. chose 3, but the project says we should choose 5.
- An example of an entry in this dictionary is as follows:

```
[1] => {[
    1,
    5,
    2,
    null
], [
    5,
    2,
    null,
    null
]}
```

That is, for system call with ID "1", a valid sequence after "1" are three (and only three) system calls, namely 1, 5, 2. Because 5 and 2 can also proceed from 1, there is a second entry with just 5 and 2 in it. Similarly, there will also be a dictionary entry for key "5" containing 2. Hopefully, the pattern is clear: We are storing all valid sequences of K.

- Only one database entry is stored per file. The database files are present under userland/ids/IntrusionDetectionSystem/database
- Log files aren't deleted when processed.

- When analyzing logs for intrusion detection, this program will consult its database. Like with the training algorithm on steps 1 and 2, each log file is iterated over and asserted to have 12 parts, then the first part is hashed to an MD5 string denoting the entry ID. However, there are some differences from that point onward:
  - If no database exists for the given log entry, it is skipped and a warning is displayed to the user. Otherwise, that database entry is loaded into memory from the JSON file.
  - Each system call is processed on a block of 5. For example, if we have a sequence of system calls like 1, 2, 3, 4, 5, 6, 7, -1, -1,-1 then the window will proceed over the calls as follows:

    1, 2, 3, 4, 5
    2, 3, 4, 5, 6

3, 4, 5, 6, 7,
4, 5, 6, 7, -1
5, 6, 7, -1, -1
6, 7, 8, -1, -1, -1

Each time, the first entry of the window is queried in the dictionary of known sequences.

- If the first entry is known, then the rest of the K – 1 values are searched for in the list of sequences under that first system call.
- If the first entry is not known, then this would be defined as "unknown behavior", meaning that it might not necessary be an intrusion, but it could mean that the program was forced to do an unexpected thing. Such as running out of disk space and making strange system calls as a result of that. The user is warned about this as such.
- If the first entry is known, but the sequence of system calls is not in the list of K – 1 known sequences, then this would be classified as a potential intrusion. If this occurs, then as per Hofmeyer et al, the minimum hamming distance of all known sequences for the beginning system call is selected from the list of known sequences.
- If any intrusions or undefined behavior occurs, the messages are saved to a log as a timestamp value under userland/ids/IntrusionDetectionSystem/intrusion_logs
- Example of a log that could be generated by the IDS:

```
POTENTIAL INTRUSION (/usr/bin/gnome-shell -- 939):
 ---> Potential breach from syscall 'futex'.
 ---> The window values are: 'futex futex writev poll writev '
 ---> The syscall sequence it most closely relates to is: 'futex futex writev poll poll '
 ---> The hamming distance: 1

UNKNOWN BEHAVIOR (/usr/bin/gnome-shell -- 939):
 ---> There are no known sequence combinations for system call 'sys_ids_log_off'.
 ---> The window values are: 'sys_ids_log_off sys_ids_log_off recvmsg write recvmsg '
```

- The program can change the location of the log directory if desired. This is a trivial user prompt problem. The user is prompted until they select a valid directory.
- The user has the option to exit the program gracefully.

Additional References

- System call dispatcher information: https://linux-kernel-labs.github.io/master/lectures/syscalls.html
- Hofmeyer et al.'s paper: https://bluegrit.cs.umbc.edu/~lsebald1/cmsc421-sp2018/hofmeyr98intrusion.pdf