Malware Detection by Invoked Systemcall Sequence

Ali Ghaffarian

Table of Content

Abstract

In order to affect the system in any way, a program needs to invoke the appropriate system call, requesting a certain operation to be done. We will trace these invocations to capture the behavior of the program, and to determine if a program is trying to harm the system.

Introduction

To satisfy the need for gathering of data for this project, we first introduce some OS and Linux concepts.

System Calls

Most operating systems, including Windows and Linux, take advantage of the two execution modes offered by most CPUs, privileged mode and non-privileged mode, also known as kernel mode and user mode. Programs we use directly as end users are user mode programs (like web browsers and text editors). These programs have no way to use the hardware except invoking system calls that allow them to request an operation to be done by the kernel, which runs in kernel mode.

Tracing

Linux allows us to gather information about what it's doing by providing various facilities and APIs, such as tracepoints, kprobes and ftrace. We will use these facilities to collect all invoked system calls to sort and format them according to our needs.

eBPF

eBPF is a technology that allows us to attach programs to various events defined in the Linux kernel and have them triggered and ran when those events happen [1]. This allows us to have a custom tracer that have only one job, submit the currently invoked system call along its arguments to the userspace program that's responsible for formatting and storing them.

Tracepoints

One of the events that our eBPF program can attach to are predefined trace points, called tracepoints [2]. We are interested in one particular tracepoint called sys_enter.

sys_enter event

The tracepoint we will be using is called sys_enter. It is triggered when a system call is invoked, with the system call number as it's first argument.

libbpf

libbpf is a c library that highly simplifies the process of writing, attaching, loading and manipulating ebpf programs[3].

Model type

No two programs behave the same, some get their work done in a second, some are expected to run until stopped. As we are interested in a full life cycle of a program, our captured sycall traces will be variable in length. For this reason we chose to make a RNN. For the sake of simplicity and delivering the deadline, we accepted the vanishing/exploding gradient problem.

Used Dataset

To train our RNN model, We used a labelled version of the ADFA LD dataset, which still to this day i can't fully trust because i the following is unknown to me:

- List of Programs which are traced
- Source code of traced malwares (except for hydra)
- Specification of the host system
- How much of program's life cycle is traced
  Unfortunately, due to my tight deadline it was the best i could find, and it seems rather credible, according to [4], it is used by three articles, but from the same researcher.

Included Malware Types

The dataset included the following malware traces:

- **Adduser:** A persistence technique, involving adding a user (often privilegded) to hide malicious activity.
- **Hydra_FTP and Hydra_SSH:** Brute force attempts against FTP and SSH servers using hydra tool. We assumed the trace is server side.
- **Meterpreter:** By using in-memory DLL injection, a meterpreter aims to provide an interactive shell to the attacker [5].
- **Java_Meterpreter:** A java version of meterpreter.
- **Web_Shell:** A type of backdoor that uses a webserver to enable persistence on the victim machine.

Complications

The mentioned dataset was not very easy to work with, below we describe the problems we faced and the approach to solution

Improper Formatting

The directory schema of the original dataset repository is the following:

```
├── Attack_Data_Master
│   ├── Adduser
│   ├── Hydra_FTP
│   ├── Hydra_SSH
│   ├── Java_Meterpreter
│   ├── Meterpreter
│   └── Web_Shell
├── Training_Data_Master
└── Validation_Data_Master
```

Considering that the Validation data was not labeled, we had to split and rearrange the dataset ourselves.

Lack of Documentation

We couldn't find any hint of how the dataset is organized, so we assumed the `Training_Data_Master` directory contains traces of normal programs, where as `Attack_Data_Master` directory contains traces of malwares of different kinds.

Mapping of Syscalls

system call id's differ between CPU architectures. Inside `ADFA-LD+Syscall+List.txt`, there's a mapping of syscall names to values. This mapping is in linux arm64 architecture, which doesn't match our experiment environment (amd64). We remap our amd64 traces to arm64 architecture when using the model.

Duplication of Data

Surprisingly, some of the data is duplicated in the dataset, a list of duplicated can be generated by the `scripts/duplicates.sh`.

Developed Toolkit

Some tools are made to enable a more simple user experience, including a tracer, log formatter and a simple wrapper for the project.

A Wrapper for Everything

The project is filled with scripts, each doing something small and contributing to the end result. Instead of coming up with a tutorial of where everything is and how use them, a makefile is written that acts as a wrapper for everything. The provided functionalities include:

- **Build the dataset:** Reformat and rearrange the dataset that has the same format of our used dataset, to the format that our dataset python class can work with.
- **Trace:** User can provide the desired program name and after the tracer is built and deployed, it will trace any program that has the matching program name that the tracer has observed it's startup. After the interruption by the user, a trace file per traced process will be written in the current working directory.
- **Check log:** Will use the provided trace file to ask the model to classify it.

Training the Model

The trained model that is readily accessible inside the repository is the result of 10 iterations of training, with 30% of the data being used for testing. At the last iteration of training, The train accuracy of 92% and test accuracy of 89% is observed.

Using the Model

Modified tracer to trace everything

Tested normal programs

Traced normal programs include

Core System Utilities

These programs are short lived, and serve a very specific purpose, for instance `ls` list the files inside the given directory.
Examples: ls, grep, uname, basename, date.

Complex applications

These programs live longer and are more likely to be used directly by the end user.
Examples: firefox, obsidian, xfce4-terminal.

Administrative Tools

Carrying out critical tasks, like installing packages, these tools are expected to have a higher false positive rate than other categories.
Examples: sudo, curl, ping, dpkg.

Deployed Malwares

Due to the tight deadline, no malwares are deployed and tested.

Confusion Metrics

False Positive: False Positive rate of 0.9% is experienced with the tested normal programs.
Flaged programs: xprop, sudo, crashhelper, zsh and python3 (can't interpret much from python3 since it will execute a given program).

Conclusion

In the course of doing this project, we made a RNN model thats purpose is to classify malicious activity from normal. Given an outcome of malicious activity is to expensive to repair, detection is not enough. A useful model should be able to interrupt the attacker and prevent damages. Nevertheless, Detecting such activity helps to develop preventive solutions in the first place. We also learned that some programs are better to be excluded from the process of tracing, such as sudo, since malwares behave very similar to them.

References

[1] https://ebpf.io/what-is-ebpf/
[2] https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_TRACEPOINT/

[3] https://libbpf.readthedocs.io/en/latest/libbpf_overview.html

[4] https://github.com/verazuo/a-labelled-version-of-the-ADFA-LD-dataset

[5] https://doubleoctopus.com/security-wiki/threats-and-tools/meterpreter/

[6] https://www.imperva.com/learn/application-security/web-shell/