A Survey of Malware Detection Using System Call Tracing Techniques
Kyle Denney*, Cengiz Kaygusuz*, Julian Zuluaga*
*Cyber-Physical Systems Security Lab (CSL), College of Electrical and Computer Engineering
Florida International University, Miami, FL
{kdenn016, ckayg001, jzulu013}@fiu.edu

Abstract—In modern computing, a program must utilize the operating system in order to run. To do so, the program must use system calls. These system calls provide the means for the program to access resources on the system (e.g., input/output, memory, etc.). Malicious programs, or malware, must also use these system calls in order to function. Due to this, we can develop techniques to analyze the system calls used by a program in order to determine if it is malicious or benign.

In this paper, we survey the methodologies used to analyze system calls for malware detection. We first examine the methods to collect system calls in varying operating systems. Then, we survey the techniques used to analyze system calls. After discussing the techniques, we briefly discuss methods where malware can thwart system call tracing techniques and how one can counter these thwarting attempts. We conclude this paper by discussing where future work is needed in these three topics.

I. INTRODUCTION

System calls are used to allow user-level programs access to system-level resources. These system calls effectively act as a bridge between user space and kernel space on a computer. With the proper tools, these system calls can be traced, or monitored, to infer what is happening during runtime.

Unless a malicious program is embedded as a rootkit in the system, malicious programs must also use system calls in order to perform their malicious functions. With that in mind, it is possible to monitor system calls to discover, analyze, and prevent malicious patterns.

In this paper, we will discuss techniques used to monitor system calls and analyze the behavior of system call patterns in order to discover malicious programs. We also discuss methods used by malware to subvert system call tracing to remain undetected.

Organization: The rest of this paper is organized as follows:

II. BACKGROUND

A. System Call Overview

System calls are the way for user-level programs to request services from the operating system (e.g., writing to a file, accessing the hard drive, etc.)

Typically, system calls are not directly accessible. Instead, the operating system provides function libraries for a programmer to access. The program calls functions from these libraries and the functions then make calls to appropriate system calls.

Once a system call is made, the running code transfers from user space to kernel space. The user program who called the system call is interrupted, transferring control to the kernel. The kernel then performs the system-level function and returns the result to the user program, allowing the program to continue operation.

B. Malware Basics

Malicious software, commonly referred to as malware, attempt to harm a victim’s computer. Unless the malware itself is installed in the kernel of the operating system, the malware will have to use system calls to function.

With this in mind, it is possible to trace system calls and analyze them for any discernible patterns. We can take common tasks accomplished in malware and apply system call trace patterns to them – thereby allowing us to determine if unknown programs are malicious should the unknown program use malicious system call traces.

III. SYSTEM CALL TRACING TECHNIQUES

In order to analyze system calls for processing, we must first collect them from the computer. In this section, we discuss methods for collecting system calls in various operating systems.

There are numerous operating systems (OS) that are frequently in use. In context of system calls collection, these OS can be categorized into following:

- UNIX-like: Most of the operating systems in mainstream use today are descendants from UNIX. As such, system call structure and tracing techniques are somewhat similar.
- Windows: Arguably the most popular operating system before the advent of smartphones, it is still widely used as an operating system in personal computers (PC).
- Other: Any other operating system that does not fall into the previous two categories drastically differ, and each one of them should be considered individually.

We shall now particularize the tracing techniques with respect to each identified category.

A. UNIX-Like

As mentioned, the OS that fall into this category share a common ancestor; as a result they have somewhat similar API structure, and the tracing techniques are alike. There is a family of standards dubbed POSIX that provides a set of API
definitions with the aim of finding a common ground between all the UNIX variants. While not all of them are completely POSIX-compliant, most of them satisfy to a certain degree that is is unusual for programs to malfunction because of the divergence from the POSIX specification.

Collection of traces in UNIX-like systems can be analyzed in two modes:

- **Userspace**: Userspace indicates code that runs in an unprivileged context. In such mode, machine code of a target program is modified so that before every system call, a piece of code that records information such as the name of the system call and its arguments, and then yields to the code that executes the system call.

  The program `strace` is the most widely used one for system call tracing. In most of the UNIX-like systems, this program or a variant of it can be found. `strace` itself uses the system call named `ptrace`. It works by: in the event of a system call, the kernel halts the monitored program and notifies `strace`, which then inspect the monitored program’s registers and stack before letting it continue. `ptrace`-based methods also make up the underlying mechanism for debuggers.

- **Kernelspace**: In this mode, code runs in a privileged context, which implies it has unlimited access to the system resources. Techniques employed at this level work by either modifying the system call table or the code which dispatches system calls.

After the general methods, we shall now describe how it could be achieved for popular UNIX-like operating systems.

1) **Linux**: For userspace tracing, `strace` is directly available for Linux systems, and included in most distributions. A more featureful program, `ltrace`, is available, which also conducts shared library call tracing.

A caveat of `strace` is that, both the caught system calls and the `stderr` of the monitored programs are printed into the `stderr` of `strace`, essentially mixing them together. This problem may be worked around by redirecting the system call traces to a file by passing a command-line argument. If this file is a named pipe, another program may use `strace` output as it is produced. The same problem and solution is also applicable to `ltrace`.

Generic methods described for UNIX-like systems are applicable to Linux.

Hooxing system calls by changing system call table could be accomplished through loadable modules. Since loadable modules run in privileged mode, it could modify any point in the memory. This approach requires the base address of the system call table, which is obtainable from `/proc/kallsyms`. Offsets for individual system calls may be obtained from `<asm/unistd.h>`, and are the form of `__NR_<call_name>`, e.g. to get the offset for the system call `open`, `__NR_open` would be used. The system call table is an array of pointers which point into the beginning address of the system call in question. A system call may be hooked by simply changing the respective system call’s pointer to the custom system call.

2) **FreeBSD**: In userspace, the program `strace` is available with the same functionality, caveats and underlying mechanism with its Linux counterpart.

In the middle ground between userspace and kernelspace, the program `ktrace` may be utilized. This program enables one to activate built-in tracing mechanism already present in the kernel that combines the conveniency of userspace tracing tools and the robustness of kernelspace-based methods.

Direct modification of system call table is still possible, if one wishes for it. Unlike Linux, the system call table is an array of a struct called `sysent` which contain necessary information about a system call such as number of arguments, the function pointer, and fields dedicated for tracing.

Hooking into the system call dispatcher is also possible, but it provides no advantage over previously discussed methods.

3) **Darwin (OS X)**: There does not exist a tool strictly operates on userspace. In the middle ground between userspace and kernelspace, the program `dtruss` may be used for system call tracing. It is based on dynamic tracing technology `DTrace` which uses a domain-specific language `D` allowing users to write custom scripts for the goal of tracing processes, also capable of tracing system call information.

With the Darwin kernel being an extension of the BSD kernel, it is possible to follow similar kernel-level techniques to trace system calls in Darwin (i.e., modifying the system call table directly).

**B. Windows**

A number of alternatives exist in the userspace. The foremost is `NiTrace`, which could be effectively considered as `strace` equivalent in Windows. The program `Process Monitor` may also be used for monitoring system calls, but it only offers this functionality through a GUI and cannot be interfaced with a program.

In recent versions of Windows, hooking into system calls through the kernel is not possible due to `Kernel Patch Protection`. This technology effectively prevents modification of critical kernel data structures, with system call tables being one of them.

**C. Other**

Most operating systems utilize a form of system call to interface the privileged logic with unprivileged logic. However, exact methods for tracing these system calls are specific to each single one of them.

The generic method of tracing system calls is through `code instrumentation` on machine code level, where the system call invocation code prepended by logic which records relevant information, and then lets the normal flow resume. This procedure is easy to accomplish, as system calls are implemented by either having their own instruction, or executing an interrupt of a specific number.

Techniques used for system calls could still be used even if the target device does not use an operating system. One method is to record subroutine calls; these subroutines may be instrumented by the compiler, or by another program working on the output of the compiler. A “higher resolution”
data may be obtained by instrumenting branch instructions, though this method may have non-negligible effect on the performance.

IV. SYSTEM CALL ANALYSIS TECHNIQUES

Once we have a method to collect system calls, we need a way to analyze them. In this section, we discuss methods used to analyze system calls to determine whether a program is malicious or benign.

A. Malicious Classification Approaches

The first approach we look at are models used to determine and classify if a program is malicious in nature. To build these models, a set of malicious system calls are gathered and machine-learning techniques are used to further determine if an unknown set of system calls are malicious. Below we describe 4 popular models built for malicious classification [12].

1) Signature-based Detection: Signature-based detection is among the most popular options for any malware detection method; including system call traces. In a general sense, known malicious programs are analyzed and signatures are made of the program. The detection model then cross-references unknown programs with the signatures and determines if the unknown program matches with any known malicious program.

With system call traces, a signature-based approach collects the system calls, or a set of system calls, from a malicious program to build into signatures. From there, an unknown program is referenced against the collection of signatures and any matches mean the unknown program is malicious.

While simple and effective, the obvious downside of signature-based detection is the ability to fool the model. For instance, a program may simply change the order of its routine and therefore change the system call traces, leaving a new signature of the program. While a model can fine-tune itself to build signatures only on fine-grained malicious routines, this takes time and effort to tune the model.

2) Probabilistic Approaches: This detection scheme relies on the assumption that given a system call \( a \), system call \( b \) is expected. Once the model has a set of malicious system calls, it builds a probability of concurrent system calls. When an unknown program runs against the model, the model takes a set of system calls and calculates the likelihood that the specific sequence is malicious. The program is classified as malware when this likelihood overcomes a predetermined percentage.

Common approaches using probabilistic data include Log-likelihood Ratio Test, Support Vector Machines, and Logistic Regression [12]. While the math model varies for each of these methods changes slightly, the core idea remains the same. The model takes a set size of

The advantage of this method is its simplicity to construct and ability to reproduce. This model works particularly well when system calls are input in sequence.

However, this model is prone to overfitting of data – leaving to possibility of models to be unreliable. Additionally, the model is built off the assumption that system calls are independent – causing the model to be unreliable when the calls are not independent.

3) Sequential Detection: The previous methods involved detecting malware with a set size of system call sequences. This method instead assumes an unknown amount of system calls and determines if a running program is malicious or benign.

The method itself is similar to probabilistic approaches as it determines the likelihood of a sequence of system calls being malicious. However, this method is particularly useful for detecting latent malware (i.e., malware that only becomes malicious after certain criteria are met). Once the program becomes malicious, the sequential detection will be able to observe this malicious behavior and act accordingly.

However, sequential detection methods are not as robust as previous methods for malware detection [12]. With that in mind, sequential detection and another detection method are often used in tandem to work more efficiently – other methods are used to detect general malware and sequential detection for real-time and latent malware detection.

B. Anomaly Detection Approaches

Benign-based models, commonly referred to as anomaly detection, is similar to malware classification in that it still uses system call traces to build a model. However, only benign data is collected to construct the model. From there, the model looks for anomalies (i.e., unexpected behavior) in the system. Anomalous detection systems can further be broken down into specification-based or learning-based [11], which will be discussed below.

1) Specification-based Detection: Specification-based detection involves building models on specific applications and programs. Here, one can collect the system call traces from trusted programs on the system and establish a baseline for system performance. From there, the model can detect when the specific programs diverge on expected behavior and classify the action as malicious when it does so.

The advantage of this specification-based detection is that it is lightweight by design and useful when a system has little variance in its performance. For instance, embedded systems would have an unchanging set of routines – specification-based anomaly detection would be able to detect when these embedded systems vary from their routine in a simple and cost-effective manner.

However, specification-based detection is not as effective when variance is introduced into a system. A specification-based model would have to account for all variance in the programs and applications, which may be unrealistic for larger systems.

2) Learning-based Detection: To account for larger systems, learning-based anomaly detection was built. This method does not rely on specific programs to build its model. Instead, system calls are collected during normal run-time. From there, the model collects a set of “distinct system call sequences” of a predetermined length [11]. Once the model is built, new system call sequences are cross-referenced to the model’s set of sequences and any anomalies trigger alerts. Benign
sequences are retroactively added to further the learning of the model.

While possibly more costly than a specification-based system, the core advantage of a learning-based model is its flexibility on a system. The model will be able to learn the difference between general benign and malicious system calls—allowing for a broader use on a system.

V. COUNTERMEASURES TO SYSTEM CALL TRACING

Malware developers get smarter every day. With each new method to detect malware, there is a malware that can circumvent its detection. The idea behind using system call traces to detect malware lies in the particular behavior of the malware when executed dynamically and the nature of this API. If we can describe malware through this behavior and use it to differentiate it from benign software, we can detect it and maybe even achieve “zero-day” detection of new malware. In this section, we discuss known methods that malware can use to evade system call tracing from User and Kernel level tools. Additionally, we will mention techniques used to evade certain detection mechanisms that depend on system call sequence behavior once a legitimate trace is established.

A. User Level

In order to bypass user level programs like Process Monitor a malicious process can remove itself from the process linked list. A process can hide itself by removing the link to its previous and next nodes to remain hidden in memory. A similar approach is when a process creates a thread and then kills it’s parent process, leaving an orphaned thread. With strace in Linux, the attempt to trace a given process’ set of system calls by executing the malware within strace environment. Malware can replace ptrace, the kernel level program that interfaces with strace, and provide the output of another process. Malware can also check if it’s being run within strace and then quit unexpectedly. Additionally, given the right escalation of privileges, malware may implement its own version of process enumeration system call by hooking, which ultimately relies on the original function but filters out its own processes.

Alternatives like dtrace are more sophisticated allowing the injection of bytecode to specific places in the kernel. This code may be triggered when system calls are invoked. Syadig utilizes a driver that captures system call events leveraging tracepoints within the kernel. Tracepoints helps install a handler that can be called from specific functions in the kernel. These are a bit more involved for malware to bypass.

B. Kernel Level

Deceiving tracing techniques at the kernel level is much more difficult, but has been demonstrated with the use of kernel driver installation or manipulation. Researchers have shown how a technique called “shadow attacks” help create shadow process communication (SPC) channels between malware and its shadow processes to achieve tracing evasion [13]. This technique involves splitting critical system call sequences of malware and exporting them to separate processes.

C. ADDITIONAL COUNTERMEASURES

Malware may evade mechanisms of detection even if its system call traces are obtained. Malware can perform a “mimicry attack”, where the sequence of system calls appears normal but can still perform the malicious behavior. This involves “nullified” system calls, where the apparently normal system calls are nulled and thus obfuscate the main objective of the malware. Other ways of avoiding detection include malware that attempts short sequences of system calls enough to avoid threshold triggers within dynamic detection, or manipulating a benign sequence of system call’s parameters to escalate privileges by which they become the attack itself and so on. [15]

We can observe that there are a lot of variables involved when attempting to design a detection mechanism and attackers always find ways around them.

VI. DISCUSSION AND CONCLUSION

System call tracing is an effective method in detecting malicious programs in a system. By collecting system calls used on a machine, we can then analyze them to distinguish common patterns. With machine learning techniques, we can then determine behavior among malicious/benign programs to allow us to uncover the intention of an unknown program’s system call behavior.

However, system call tracing is not an end-all solution to malware detection. It is possible for malware to disguise their behavior by subverting system call tracing techniques and hiding themselves in the operating system. Additionally, rootkits who do not utilize system calls in the first place are completely impervious to this detection schema. System call tracing is simply another tool of many that can be utilized for detecting malicious activity in a system.

REFERENCES


