A Survey Paper on Novel Malware Clustering system based on Kernel Data Structure

Trupti Vasantrao Bhandare1, Pramod B. Mali2

1Dept. of Computer Engineering, Smt. Kashibai Navale College of Engineering, Vadgaon Bk, Pune, India
2Assistant Professor, Dept. of Computer Engineering, Smt. Kashibai Navale College of Engineering, Vadgaon Bk, Pune, India

Abstract— Analysis of malware detection goes through various problems, such as obfuscations and variation behavior structure in static and dynamic analysis. Malware detection and approaches have been traditionally following on code-centric behavior of malicious programs, such as detection of the injection of malicious programs, code sequences. This paper introduces a new approach to novel clustering of kernel behavior by using kernel data access patterns. However, modern malware has been employing advanced strategies, such as reusing legitimate code or obfuscating malware code to circumvent the detection. To determine the effectiveness, we generate the signatures of the classic rootkits using their data access patterns, and matched them with the kernel execution instances. The core idea is that when kernel malware tampers with core kernel data, there exist unique kernel data access patterns. As such, we could take a subset of data access patterns that consistently appears in multiple kernel execution instances only when the malware is active and generate the malware signatures using the subset.

Keywords— Data access patterns, Clustering of kernel, Malware detection, code-centric behavior of malicious programs

I. INTRODUCTION

An operating system (OS) kernel is the core of system software that is responsible for the integrity and operations of a conventional computer system. It has been targeted by malicious software (malware) that operates in kernel mode to implement advanced stealthy features, such as backdoors or hidden services that can elude user-level anti-malware programs.

Malware tampers with program execution and achieves the attacker’s malicious goals with a variety of techniques. Many traditional malicious programs use code injection attacks (e.g., buffer overflows and format string bugs), which inject unauthorized code into the memory and executes malware functions. Various kinds of malware, such as computer worms, viruses, exploits, and rootkits, have been using this technique to execute malicious logic. Many intrusion detection approaches have been proposed to detect or prevent this type of malware attack in previous research works.

In response to these malware defense approaches, malware writers have crafted advanced attack vectors that avoid explicit injection of malicious code to elude such detection approaches. Return-to-libc attacks\[1, 2\], return-oriented programming \[3, 4, 5\] and jump-oriented programming \[6, 7, 8, 9, 10\] use a combination of existing code pieces to compose malicious logic. Also, raw memory devices, third-party kernel driver code, and program bugs provide other vectors to reuse legitimate or vulnerable code which are a part of programs for malware attacks.

Another group of defense approaches has been using the sequence of malware code to detect malware. These approaches use malware signatures composed of malware code sequences, such as instruction sequences or system call patterns, to match malware behavior. However, in response to them, malware began to employ techniques to vary malware code execution patterns. Several technical publications have presented code obfuscation and code emulation techniques, which can confuse malware detectors and avoid detection. These arms-races observed between malware and malware detectors center around the properties of malicious code: injection of code and the causal sequences of malicious code patterns. Both techniques use primarily code information, ignoring the identification and properties of the accessed data objects.

In general, computer programs are structured as code and data. Therefore, malicious attacks are seen as the manipulation of the code and/or data objects of the program under attack. Code has been a popular target of attacks, and thus it has been intensively studied by existing malware detection approaches. In contrast, there has been little focus to date on the data in malware defense research works.

To address the challenges of relying on only code in malware defense, we propose new approaches based on the properties of data objects that are targeted in malware attacks. These approaches do not require the detection of the injected code or the specific sequence of malicious code. Therefore, they are not directly subject to attacks targeting the approaches based on code properties.
These approaches, however, have unique challenges in monitoring data objects: the dynamic status of data objects and the difficulty of determining their integrity. For instance, many data objects have readable-and-writable content and the locations of dynamic objects are assigned at runtime. A monitor observing data objects should have a higher level privilege than the monitored program to reliably obtain its data memory status. Monitoring kernel data objects is challenging because, in a conventional computing environment, an OS kernel directly interacts with the hardware, thereby lacking a layer below it on which to build a monitor.

II. LITERATURE SURVEY

An operating system kernel is the core of system software which is responsible for the integrity and operations of a conventional computer system. Authors of malicious software (malware) have been continuously exploring various attack vectors to tamper with the kernel. Traditional malware detection approaches have focused on the code-centric aspects of malicious programs, such as the injection of unauthorized code or the control flow patterns of malware programs. However, in response to these malware detection strategies, modern malware is employing advanced techniques such as reusing existing code or obfuscating malware code to circumvent detection.

A. Code Injection Attacks and Code Integrity-based Approaches [16][17]

Code injection attacks insert unauthorized code into a program’s memory space and transfer the control to the injected code. Various kinds of malware, such as computer worms, viruses, shell code, and rootkits use this technique to change program behavior with malicious purposes. There are various attack vectors to inject code. For instance, kernel rootkits load rootkit code into kernel memory space by using kernel drivers or raw memory devices. Then, they move the kernel control to the injected code by patching the system call table or function pointers. This category of malware can be defeated by enforcing the integrity of the program’s code and only allowing the execution of authorized and un-tampered code. There are various mechanisms to achieve this in the user space, in the kernel space, and also in the hardware level.

B. Non-Code Injection Attacks and Defense Approaches [3], [4], [5], [6], [7], [8], [9], [10]

While many malware programs rely on code injection, there is another group of malware that does not require the insertion of malicious code for attacks. The malware of this class reuses an existing program’s code to elude intrusion detection approaches based on code integrity. Following are several attack vectors of this group of malware.

- Kernel Memory Devices: Operating systems have kernel memory devices that allow the read and write capability of raw kernel memory. For example, Linux has several devices, such as /dev/kmem, /dev/mem, and /dev/kcore; and Microsoft Windows has similar devices called /Device/PhysicalMemory and /Device/DebugMemory. These devices are intended for kernel debugging, efficient access to video memory, and memory forensic analysis; but if they are misused for malicious purposes, they can be a serious threat to the kernel’s integrity. Some kernel rootkits use these devices to manipulate kernel memory without using kernel drivers. In the Windows platform, several worms (e.g., W32/Myfip.h and W32/Fanbot.A) use raw memory device /Device/PhysicalMemory to tamper with kernel memory.

- Return-oriented Programming: Return-oriented programming generates an attack by combining a large number of short instruction sequences (called gadgets) that allow arbitrary computation. This technique is also used to implement kernel level malware (e.g., return-oriented rootkits). This type of malware is code integrity-based approaches cannot detect its attacks. Several approaches have been proposed to use runtime characteristics during attacks to detect this malware. Other approaches attempt to remove potential gadgets by removing return instructions or potential instruction sequences which can be used as gadgets from the program.

- Jump-oriented Programming: As detection approaches for return-oriented programming appear, other instruction sequences, similar to the return gadgets, are used for constructing attacks. These approaches use instruction sequences that end with jump instructions to connect multiple code pieces and to create malicious logic. These approaches show that essentially any instruction sequence whose control flow can be manipulated by attackers can be used for attacks. This idea was conceptualized as “free-branches”.

C. Malware Defense Based on Code Behavior Signatures [13], [14], [15]

There has been a variety of approaches which characterize malware’s behavior by using its control flow. Several approaches are building control flow graphs using system call events, and another approach uses CPU instructions to represent malware behavior.
While these malware patterns are derived from the events of different system layers, they commonly represent the control flow of malware, which is a sequence of code with causal dependence. There are two kinds of challenges for these approaches.

First, advanced malware can generate variations in the control flow to avoid detection by these approaches. Several papers describe obfuscation techniques such as dead code insertion, code transformation, and instruction substitution. Malware can obfuscate its code execution while retaining the same algorithm. In addition, researchers introduced a new obfuscation technique that hides specific trigger-based behavior by encrypting the code dependent on an input.

Second, malware’s control flow can dynamically vary at runtime and the detection mechanism using malware’s code behavior should be able to handle such variations. Balzarotti et al. presented a system that uses system-call trace to determine analysis-aware malware. In this paper, the authors described several cases where the system-call trace can be inconsistent, such as the expiration of timeout and the delivery of signals. Their system handles this problem by using a flexible matching algorithm.

D. Malware Defense Based on Data Signatures [18]

Like any other program, malware uses data structures. Some malware has its own data structures. Other malware tampers with the data structures that they target to make changes in the program’s behavior. There are several approaches that detect malware based on the signatures of data structures. Researcher Laika determined data structures from a program’s memory. As one application, the authors presented the detection of a botnet program by classifying data structures specific to malware. This approach is effective for user space malware because each user program has its private memory space. However, kernel memory is shared by the kernel text and many kernel drivers. Malware’s code and data are part of a huge number of legitimate kernel code and data and therein lays the challenge to applying this technique to kernel malware detection.

These approaches can discover data structures from a memory image in a benign scenario. However, if those approaches are targeted for detecting malware, there could be the following challenges. First, malware can manipulate the data objects so that the data scanners fail to detect them while such objects are being properly used by malware code. For example, malicious code can set invalid values or pointers in the data structure while the injected malware code properly uses such objects. Second, it is possible that some data structures may not have enough constraints to be matched by these approaches.

For instance, if a kernel data structure is simple, such as a string buffer, these approaches do not have specific constraints to match them, leading to many false positive cases.

E. Kernel Integrity Checking based on Kernel Memory Mapping [11], [12], [13]

There have been several approaches that leverage kernel memory mapping to test the integrity of OS kernels and detect kernel malware. These approaches identify kernel memory objects by recursively traversing pointers in the kernel memory starting from static objects in a similar way to garbage collection mechanisms. A kernel object is identified by projecting the address and the type of the traversed pointer onto memory; thus, we call this mechanism type-projection mapping. When type-projection mapping is used against kernel malware, these problems may pose concerns as such inaccuracy can be deliberately introduced by kernel malware. In type-projection mapping, the kernel memory map is based on the content of the kernel memory, which may have been manipulated by kernel malware. This property may affect the detection of kernel rootkits that hide kernel objects by directly manipulating pointers. The type-projection mapping does not have information to determine this attack because it constructs a map of data instances based on memory values.

To detect such attacks, a detector needs to rely on not only a kernel memory map but also additional knowledge that reveals the anomalous status of the hidden objects. For this purpose, several approaches use data structure invariants. For example, KOP detects a process hidden by the FU Rootkit by using the invariant that there are two linked lists regarding process information that are supposed to match, and one of them is not manipulated by the attack. However, a data invariant is specific to semantic usage of a data structure and may not be applicable to other data structures. For type-projection mapping, it is challenging to detect data hiding attacks that manipulate a simple list structure (such as the kernel module list in Linux) without an accompanying invariant.

III. Conclusion

This paper gives survey about different malware detection approaches. Also, we have surveyed the rootkit detection techniques that modify the controlled and non-controlled data. The technique effectively and efficiently detects binaries with split personalities, while it can successfully replay programs that do not contain any checks for the emulator (Qemu). We have discussed many existing systems and also the development which can be done in this topic.
REFERENCES


