POSTER: KXRay: Introspecting the Kernel for Rootkit Timing Footprints

Chen Chen Darius Suciu Radu Sion

Stony Brook University
New York, USA
{chen18, dsuciu, sion}@cs.stonybrook.edu

ABSTRACT

Kernel rootkits often hide associated malicious processes by altering reported task struct information to upper layers and applications such as ps and top. Virtualized settings offer a unique opportunity to mitigate this behavior using dynamic virtual machine introspection (VMI). For known kernels, VMI can be deployed to search for kernel objects and identify them by using unique data structure "signatures".

In existing work, VMI-detected data structure signatures are based on values and structural features which must be (often exactly) present in memory snapshots taken, for accurate detection. This features a certain brittleness and rootkits can escape detection by simply temporarily "un-tangling" the corresponding structures when not running.

Here we introduce a new paradigm, that defeats such behavior by training for and observing signatures of *timing access patterns* to any and all kernel-mapped data regions, including objects that are not directly linked in the "official" list of tasks. The use of timing information in training detection signatures renders the defenses resistant to attacks that try to evade detection by removing their corresponding malicious processes before scans. KXRay successfully detected processes hidden by four traditional rootkits.

Keywords

Kernel introspection; malware detection; rootkits

1. INTRODUCTION

The arms race between malicious attacks and the defense countermeasures leads to the development of ever more sophisticated tools for both the attacking and defending party. Some of these tools have the purpose of protecting the operating system against malicious processes, that aim to install and hide themselves along their payload in the system's memory. This has lead VM introspection (VMI) [4] to become a well-developed research topic in recent years, many VMI systems being designed to identify malicious applications [8] or ensure the integrity of sensitive files [5].

CCS'16 October 24-28, 2016, Vienna, Austria

ACM ISBN 978-1-4503-4139-4/16/10.

DOI: http://dx.doi.org/10.1145/2976749.2989053

A system can use VMI to analyze running softwares present in a virtual machine (VM) [4], but inspecting the VM from an external layer introduces a semantic gap, whose removal represents another challenge. Kernel analysis tools, such as the Linux crash utility [4], could be used to overcome this challenge, but require recompiling the kernel with debugging symbols enabled, imposing restrictions on the applications that reduces their applicability in the real world. Alternative approaches reconstruct kernel objects by determining their location in the system's memory. Such approaches either have to scan a reconstructed memory object graph, rooted at global variables, as the KOP[1] and MAS[2] systems do, or they use a brute force approach to scan the entire memory for potential matches with invariant signatures of data structures[3, 6, 7]. There is a limitation common in all these solutions, rooted in their foundation, as they can only analyze a snapshot of the memory.

Actions made by benign and malicious processes are reflected as changes in the system's memory, leading to time periods (memory snapshots) when the memory contents could contain leads useful in identifying attacks. These leads however are temporary, because any change made to memory contents can be reverted and appear to never have occurred. Rootkits could avoid detection by shutting down their related processes when they anticipate imminent memory snapshot scans. In contrast, performing a memory operation is a irrevocable action, which can not be undone by the rootkit. Thus, in this poster we present a detection tool that can analyze in real-time the system's memory behavior and identify the objects present. This tool can be used to detect hidden malicious processes the moment they are scheduled to run, offering the system an opportunity to stop them in their tracks.

Among the hundreds object types, one of the most interesting objects for VM introspection are process descriptors, as they contain almost all information related to runnning processes, being usefull in the identification of hidden processes. Thus, the kernel object detection tool proposed in this poster focuses on the detection of the process descriptors present in the VM. For brevity, we use the process descriptor in Linux (task_struct) as a representative.

To describe the proposed technique briefly, we aim to detect all active task_struct instances of a system in real-time, by analyzing the memory access behavior. We analyze memory reads and writes events, rather than memory values, making our design naturally resistant to attacks that can only manipulate the memory value.

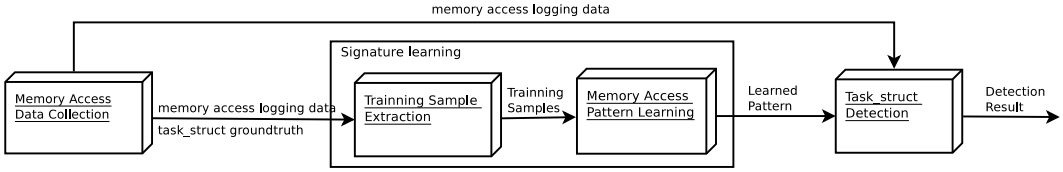

Figure 1: The framework of the system.

2. PROBLEM DESCRIPTION

Stealthy rootkits or other malicious processes can be identified by detecting their associated task_structs, which they try to hide from the system. Our objective is detecting all task_structs inside a system in real-time by building a tool that can analyze the system's memory access behavior. First we require a task_struct specific pattern (a signature) to search for hidden processes. Existing memory snapshot based signatures would only allow us to detect rootkits after they wreak havoc, so we plan to find a new signature in the memory access behavior information instead. The relationship between multiple accesses and their order can form unique sequences, which once identified can be used to build the signature.

We formalize the construction of the memory access based signature as a common sequence problem: Given a training set where each sample Sq_i is a memory access sequence to a task_struct during its lifetime, find a sequence U that is not only the "subsequence" of every training sample but also occurs frequently in each sample. If the majority of a signature sequence U is a subsequence of Sq_i, we assume that U is the signature.

Once we obtain the signature, we need a new scanning scheme which can track all events that occur in the running system and use it to identify malicious processes.

3. PROPOSED SYSTEM

A system overview diagram is depicted in Figure 1. Our detection tool is composed of three main modules: (1) The data collection module. (2)The signature learning module (3) The detection module. The data collection module is an essential component that profiles all useful memory accesses in a fixed format. This module provides the data source for both the signature learning and detection modules. In our work, we use the PADNA system to record the memory accesses inside a VM. With the retrieved data, we use the signature learning module to generate the task_struct's access signature.

The signature needs to capture a unique pattern shared by all task_structs memory access sequences. Manually finding it in the enormous memory access timeline is an impossible task, but we can use a set of training samples that describe the memory access to single task_struct to automatically find it using machine learning. We decompose this process into 2 phases: a training sample extraction phase and a access pattern learning phase. We require training samples to represent the ground truth regarding the actual location of a task_struct in memory. We construct them by extracting subsequences containing only memory accesses to one tack_struct from a running system's memory access timeline.

Before we begin the signature learning process, we have to transform the input sequences into segments based on their memory access density in a unit of time, because the accesses to a task_struct are not distributed uniformly over time (repeated sequences always occur in a short time period and some of them are followed by a few irregular scattered accesses).

In order to obtain a good access pattern, we make use the Longest Common Subsequence (LCS) and the Shortest Common Supersequence (SCS) extraction algorithms to get rid of the individual differences between subsequences, while keeping the common accesses part of the signature sequence. For each task_struct, a feature sequence is learned, that describes the repetitive sequences present in a single task_struct. To retrieve these feature sequences, a agglomerative hierarchical clustering algorithm is used to group the segments which are similar, using the edit distance to measure the dissimilarity. Next, we use agglomerative hierarchical clustering to group the similar feature sequences and then apply the longest common subsequence algorithm. The longest common subsequence obtained represents a subsequence present in all task_struct access timelines, which we refer to as the signature of task_struct objects. An illustration of this learning process is presented in Figure 2.

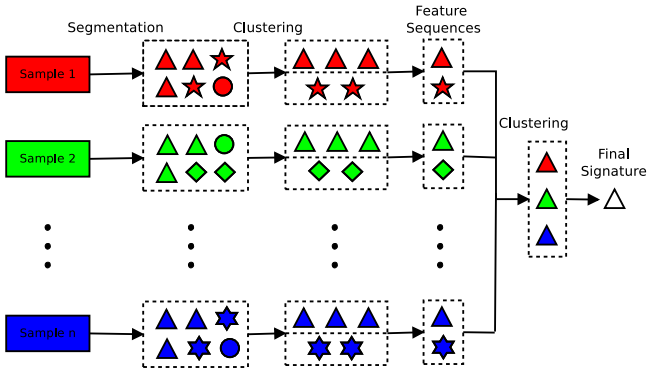

Figure 2: An illustration of the signature learning process

In the monitoring phase we just slide a time window and compare the data inside with our obtained signature,relying on the edit distance[9] as the similarity metric for determining task_struct signature matches. In each window, the scanner first collects all possible candidate substrings that could match a predefined task_struct signature and their corresponding base addresses. This operation is only concerned about the memory addresses being accessed, regardless the relationship between multiple accesses. Next, we filter our all false positives from the candidates, retaining only candidate substrings whose edit distance to the signature is small enough. This process is depicted in Figure 3. In the end, ev-

ery remaining sequence represents an access to a task_struct. If the process this task_struct is associated with is not included in the results presented by a system utility (e.g. ps), then we have identified a hidden process, indicating a possible rootkit infiltration.

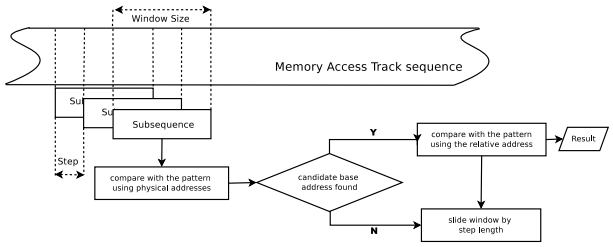

Figure 3: The process flow of the scanner.

4. EVALUATION

By taking advantage of a PANDA platform plug-in, we instrument QEMU for monitoring and record all the memory accesses made on an emulated system with a 32 bit Linux kernel, version 2.6.32.27 and memory size of 1024MB. Using only the tracking data provided by the data recording module and the signature learned using the pattern learning module, the scanner successfully identifies almost all active task_structs, the exceptions including only one task_struct that has very few accesses and one that behaves in a unique manner, corresponding to the idle process.

On a system running a Linux kernel 2.6.32.27, we retrieve two sequences shared by most of the training samples, one related to a process's wake up event, while the other corresponds to a time interrupt event and occurs more frequently during the lifetime of a process. When a process is woken up or an interrupt event is sent, its corresponding task_struct is accessed in a pattern similar to our retrieved sequences, allowing us to identify the task_struct location when each such event occurs. By using them together as a single signature, we can detect both long lived processes, that encounter a lot of time interrupts, but are scheduled to run only under a certain condition, and the frequently scheduled short lived programs, that do not encounter any time interrupts.

After conducting experiments with 4 real-world kernel rootkits, representative to the Kernel Object Hooking (KOH) and Direct kernel object manipulation (DKOM) categories, we present the obtained results in Table 1. The kernel rootkits we use are adore-ng, enyelkm 1.0, hp and linuxfu for Linux 2.6.32, each being used to hide some process in the system. The first two belong to the KOH category, while the last two belong to DKOM. The results indicate that all processes hidden by these rootkits are successfully detected by our prototype.

Kernel rootkit	Hidden Task_struct #	Detected # (wake-up sequence)	Detected # (time-interrupt sequence)
adore-ng	3	3	3
enyelkm	2	2	2
hp	4	4	4
linuxfu	2	2	2

Table 1: Kernel rootkit detection results.

5. CONCLUSIONS

The task_struct object represents a crucial data structure present in the Linux kernel, making it a target of attacks, including kernel rootkits that focus on hiding malicious process's task_structs. In this poster, we propose a framework that automatically learns a memory signature and then uses it for real-time active task_struct detection. The detection process involves checking the similarity of monitored memory operation sequences with a new signature learned during a training phase. This signature captures invariants present in the systems's dynamic memory access behavior, focusing on the order of a group of memory accesses and the relationship that exists between the operations contained within.

In contrast to value invariant-based and structure invariant-based signatures, our proposed solution does not rely on memory values present in a memory snapshot or require a detailed definition of the target data structure or knowledge regarding the kernel's source code. The experiments conducted indicate this detection tool can effectively find hidden processes and rootkits, while reporting no false positives.

Acknowledgments

This work was supported by the National Science Foundation through awards 1161541, 1318572, 1526102, and 1526707.

6. REFERENCES

[1] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 555–565, New York, NY, USA, 2009. ACM.

[2] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *USENIX Security Symposium*, pages 601–615, 2012.

[3] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 566–577, New York, NY, USA, 2009. ACM.

[4] T. Garfinkel, M. Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, volume 3, pages 191–206, 2003.

[5] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with osck. *ACM SIGPLAN Notices*, 46(3):279–290, 2011.

[6] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu. Dimsum: Discovering semantic data of interest from un-mappable with confidence. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, San Diego, CA, February 2012.

[7] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS*, 2011.

[8] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, pages 243–258, 2008.

[9] Wikipedia. Edit distance. Online at https://en.wikipedia.org/wiki/Edit_distance.