IJREAT International Journal of Research in Engineering & Advanced Technology, Volume 3, Issue 1, Feb-Mar, 2015
ISSN: 2320 – 8791 (Impact Factor: 1.479)
www.ijreat.org

A Comprehensive Study Of Kernel Attacks And Its Countermeasures In Virtual Machines

Bright Prabahar P[1], Vetrivelan[2]

[1]*Assistant professor, Parisutham Institute of Technology & science, Thanjavur-613007, India*

[2]*Professor, Periyar Maniammai University, Thanjavur-613403, India*

***Abstract*— A kernel is the core component of an operating system. With the use of interprocess communication and system calls, it acts as a bridge between various applications and the data processing performed at the hardware level. During a privilege escalation attack, the attacker to grant himself higher privileges. This is typically achieved by performing kernel-level operations that allow the attacker to run unauthorized/malicious code which results in corrupting the kernel. The main aim of the paper is to present the study of various kernel & kernel attacks present in different operating systems. This paper is also focus on study of various counter-attack methods which are used to protect kernel. Normally kernel can be protected by using three different strategies which includes monitoring the invoked process, snooping the incoming packets at network level and establishing trust of a process by using TCB(Trusted computing Base-A database of malicious process which are periodically updated by the admin) different methods in different layer for example In network layer by snooping incoming packets.**

***Index Terms*— Kernel, OS, Process monitoring, Malware analysis, Virtual Machine Monitor.**

I. INTRODUCTION

Nowadays malware use a variety of techniques to cause divergence in the attacked program's behavior and achieve the attacker's target. Traditional malicious programs such as computer viruses, worms, and exploits have been using code injection attacks which inject malicious code into a program to perform a nefarious function. Some Intrusion detection techniques based on such code properties effectively detect or prevent this class of malware attacks [5], [6].

Attackers are becoming strong they use alternate methods such as return-to-libc attacks [7], return-oriented programming [8], and jump-oriented programming [9] reuse existing code to create malicious logic. Additionally, kernel malware can be launched via vulnerable code in program bugs [10], [11], third-party kernel drivers, and memory interfaces [12] which can allow manipulation of kernel code and data using legitimate code (i.e., kernel or driver code).

In order to detect such attacks, another group of detection techniques which focus on identifying malicious software based on its behavior [13]. These approaches generate malware signatures by using a code sequence pattern of malware (e.g., instruction sequences or system call sequences) to match such behavior. However, some malware employ techniques that triggers a variety of code execution pattern. For example, code obfuscation[14] and code emulation[15] techniques can confuse behavior-based malware detectors and hence avoid detection.

This race between malware and malware detectors centers around the various properties of malicious code: injection/integrity of code or the causal sequences pattern of malicious code. This paper focus on the various malware detection techniques[1][2][3][4] which protects kernel.

II. DATA-CENTRIC OS KERNEL MALWARE DETECTION

In data-centric OS kernel malware detection[1] has a brilliant scheme which detects and characterizes various malware attacks based on the properties of manipulated data objects during the attacks. This framework consists of two system components.

- First, a runtime kernel object mapping system which has an un-tampered view of kernel data objects resistant to manipulation by malware. This view is effective at detecting a class of malware that hides dynamic data objects.
- Second, a new kernel malware detection approach that generates malware signatures based on the data access patterns specific to malware attacks. This approach has an extended coverage that detects not only the malware with the signatures but also the malware variants which share the attack patterns by modeling the low level data access behaviors as signatures.

A.Kernel object mapping scheme:

At runtime, the VMM captures all allocation and deallocation of events by intercepting whenever one of the allocation/deallocation functions is called. There are three things which has to be determined during runtime: (1) the call site, (2) the address of the object allocated or deallocated, and (3) the size of the allocated object. To determine such call site, The return address of the call is used to the allocation function. In the instruction stream, the Kernel level memory allocation functions are similar to user level ones. The function kmalloc, for example, has not take a type but a size to allocate memory. return address is the address of the instruction after the call instruction. The captured call site is stored in the kernel object map hence that type can be determined at the time of offline source code analysis. The address and size of objects has been allocated or deallocated can be derived from the arguments and return value. For the allocation function, the size is given as a function argument and the memory address as the return value. For a deallocation function, the address is given as a function argument. These values can be determined by the VMM by using functions of call conventions. Function arguments are delivered by the use of the stack or the registers, and they are captured by inspecting these locations at the entry of memory allocation/deallocation calls. In order capture the return value, the timestamp & location of the return value is used. Integers up to 32-bits as well as 32-bit pointers are delivered via the EAX register and all values of same types. The return value is available in this register when the allocation function returns to the caller. In order to capture the return values at the correct time the VMM uses a mechanism called virtual stack. When a memory allocation function is called, the return address is extracted from the function and pushed on to this stack. When the address of the code to be executed matches the return address on the stack, the VMM intercepts and captures the return value from the EAX register.

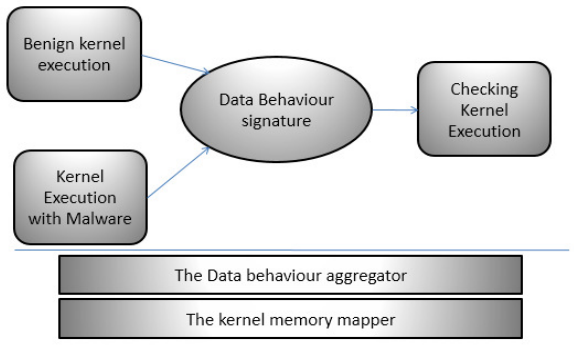

Figure 1. Data behavior based malware characterization

B.Kernel malware detection:

The data behavior of kernel malware is characterized and used to determine the presence of malware. The overview of this component is presented in Figure. 1, and the sub-components are as follows. As a basic unit to represent the kernel's data behavior, DataGene generates a summary of the various access patterns for all kernel objects accessed in a kernel execution instance. For each access on kernel memory in the guest OS, the VMM intercepts and records the relevant information about the memory access of kernel, such as the accessing code, the accessed memory type, and the accessed offset (shown as The Data Behavior Aggregator).

To determine malware behavior, the memory access patterns for two kinds of kernel execution instances are generated: benign kernel which runs and malicious kernel runs where kernel malware is active. By taking the difference between these two sets of memory access patterns, the data behavior specific to the kernel malware was determined and its signature also generated (Data Behavior Signature). Later, in order to detect kernel malware, the generated signatures are compared to the memory access patterns of a running instance of the OS (Checking Kernel Execution).

III. HyperCheck: A Hardware-Assisted Integrity Monitor

HyperCheck[2] is composed of three key components: the physical memory acquiring module, the analysis module and the CPU register checking module. The memory acquiring module reads the contents of the physical memory of the machine and sends them to the analysis module. In analysis module, It checks the memory contents and verifies if anything is altered. The CPU register checking module reads the registers and validates their integrity. The overall architecture of HyperCheck is shown in figure 2.

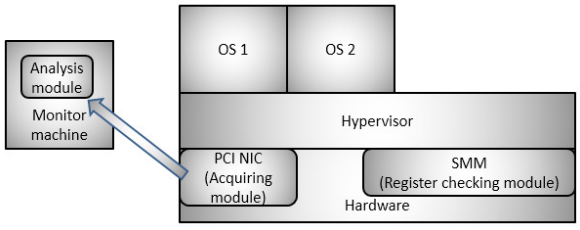

Figure 2. Hyper check

HyperCheck should not rely on any software running on the machine except the boot loader. Since the software may be compromised, one cannot trust even the hypervisor. Therefore, we use hardware – a PCI Ethernet card – as a memory acquiring module and SMM to read the CPU registers. Usually, Ethernet cards are PCI devices with bus master mode enabled and are able to read the physical memory through DMA, which does not need help from CPU. SMM is an independent operating mode and could be made inaccessible from protected mode hence hypervisor and privileged domains cannot run.

Previous researchers only used PCI devices which is used to read the physical memory. However, CPU registers are also important because the location of active memory used by the hypervisor or an OS kernel such as CR3 and IDTR registers are defined by CP registers. Without these registers, the attacker can launch a copy-and-change attack. It means the attacker copies the memory to a new location and modifies it. Then the attacker updates the register to point to the new location. PCI devices cannot read the CPU registers, thereby failing to detect this kind of attacks. By using SMM, HyperCheck can examine the registers and report the suspicious modifications.

IV. Process Authentication for High System Assurance

Our Authenticated Application (A2) design[3] which enables the authentication of applications. It consists of three components: Credential Registrar, Authenticator and Service Access Monitor (SAM). There are three components in the architecture which are explained below.

- Credential Registrar is for generating a credential for the application and registering the application with the kernel.
- Authenticator is for authenticating a process when it first starts.
- Service Access Monitor (SAM) is for verifying the authentication status of a process at runtime, i.e., whether the process has been successfully authenticated by the Authenticator

A.Compatibility Mode for Legacy Applications:

Process authentication protocol requires the modification of legacy applications to support the interaction with the Authenticator, raising a compatibility issue. A middleware to perform the authentication on behalf of the application was designed. The credential generation operation is unchanged. System authenticate legacy applications using a helper program referred to as the Verifier. The Verifier has the read access to the credential list L maintained by the registrar, but not the write access. The steps are explained below.

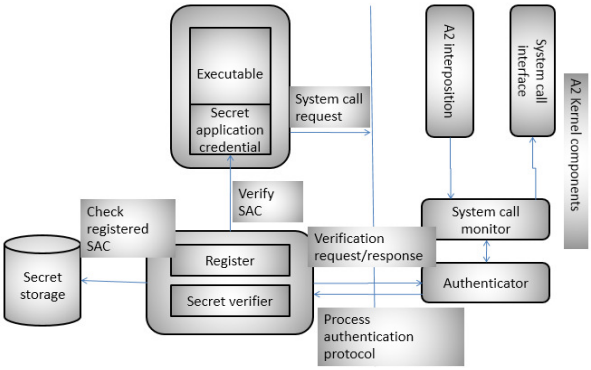

Figure 3. Workflow of A2 in compatibility mode with the verifier

1. To authenticate a newly started process p with process name p.name, process ID p.pid, and the path to the code capsule p.path (all obtained from the kernel), the Authenticator checks if the process has already been verified by looking its p.pid up in the status list T. If p.pid /2 ϵ T, the Authenticator sends to the Verifier (p.path, p.name).
2. The Verifier reads p.path to retrieve the application's copy of the credential at the end of its code capsule. This credential is denoted by p.cred. It throws an error if the credential cannot be found.
3. The Verifier looks up the credential list T by p.name to retrieve the corresponding credential, which is denoted by p.cred'. It throws an error if p.cred' is null.
4. The Verifier checks if p.cred' == p.cred 2. If yes, then the authentication succeeds. Otherwise, fails. The Verifier notifies the Authenticator with the authentication result.
5. The Authenticator updates the status list with p.pid.

In A2 prototype, the Verifier is implemented as a userspace application. It has a shared memory region with the Authenticator to exchange verification messages. The Verifier is equipped with a manually installed credential, so that itself can be authenticated as a bootstrapping procedure. When the Verifier's process starts, the Authenticator authenticates its identity to prevent identity spoofing.

V. Scalable Distributed Service Integrity Attestation for Software-as-a-Service Clouds

IntTest[4], a scalable and effective service integrity attestation framework for SaaS clouds. IntTest provides a novel integrated attestation graph analysis scheme which is shown below that can provide stronger attacker pinpointing power than previous schemes. Moreover, IntTest can automatically enhance result quality by replacing bad results produced by malicious attackers with good results produced by benign service providers.

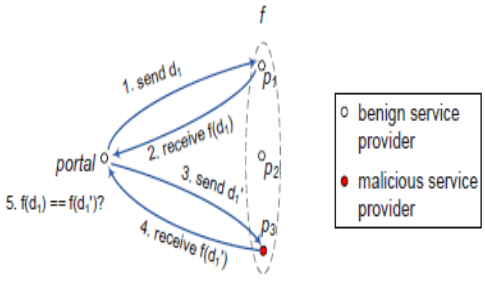

Figure 4. Replay-based consistency check

In order to detect service integrity attack and pinpoint malicious service providers, Intest algorithm relies on

replay-based consistency check to derive the consistency/inconsistency relationships between service providers. For example, Figure 4 shows the consistency check scheme for attesting three service providers p1, p2, and p3 that offer the same service function f. The portal sends the original input data d1 to p1 and gets back the result f(d1). Next, the portal sends d′ 1, a duplicate of d1 to p3 and gets back the result f(d′1). The portal then compares f(d1) and f(d′ 1) to see whether p1 and p3 are consistent.

The intuition behind this approach is that if two service providers disagree with each other on the processing result of the same input, at least one of them should be malicious. Note that we do not send an input data item and its duplicates (i.e., attestation data) concurrently. Instead, we replay the attestation data on different service providers after receiving the processing result of the original data. Thus, the malicious attackers cannot avoid the risk of being detected when they produce false results on the original data. Although the replay scheme may cause delay in a single tuple processing, we can overlap the attestation and normal processing of consecutive tuples in the data stream to hide the attestation delay from the user.

If two service providers always give consistent output results on all input data, there exists consistency relationship between them. Otherwise, if they give different outputs on at least one input data, there is inconsistency relationship between them. We do not limit the consistency relationship to equality function since two benign service providers may produce similar but not exactly the same results. For example, the credit scores for the same person may vary by a small difference when obtained from different credit bureaus. We allow the user to define a distance function to quantify the biggest tolerable result difference. Using the graph scheme the malicious behavior of software is detected in cloud based services.

VI. CONCLUSION

In modern computing era, virtual machines takes major role. Virtual machine security is very important in cloud computing, we have discussed various virtual machine architectures & its techniques to prevent malicious attacks in kernel. In order to withstand such kind of attacks a lightweight and scalable security architecture has to be designed from the disadvantages of methods which explained in the above section.

REFERENCES

[1] Junghwan Rhee; Riley, R.; Zhiqiang Lin; Xuxian Jiang; Dongyan Xu, "Data-Centric OS Kernel Malware Characterization," Information Forensics and Security, IEEE Transactions on , vol.9, no.1, pp.72,87, Jan. 2014

[2] Fengwei Zhang; Jiang Wang; Kun Sun; Stavrou, A., "HyperCheck: A Hardware-AssistedIntegrity Monitor," Dependable and Secure Computing, IEEE Transactions on , vol.11, no.4, pp.332,344, July-Aug. 2014.

[3] Almohri, H.M.J.; Danfeng Yao; Kafura, D., "Process Authentication for High System Assurance," Dependable and Secure Computing, IEEE Transactions on , vol.11, no.2, pp.168,180, March-April 2014.

[4] Juan Du; Dean, D.J.; Yongmin Tan; Xiaohui Gu; Ting Yu, "Scalable Distributed Service Integrity Attestation for Software-as-a-Service Clouds," Parallel and Distributed Systems, IEEE Transactions on , vol.25, no.3, pp.730,739, March 2014.

[5] H. Etoh. GCC Extension for Protecting Applications From Stacksmashing Attacks. http://www.trl.ibm.com/projects/ security/ ssp/ . Accessed May 2011.

[6] Vendicator. Stack Shield: A “Stack Smashing” Technique Protection Tool for Linux. http://www.angelfire.com/ sk/ stackshield/ info.html. Accessed May 2011.

[7] Bypassing Non-executable-stack during Exploitation using Return-to-libc. Phrack Magazine.

[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS’08), pages 27–38. ACM Press, Oct. 2008.

[9] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating Return-Oriented Rootkits with ”Return-Less” Kernels. In Proceedings of the 5th European conference on Computer systems (EUROSYS’10), 2010.

[10] The Month of Kernel Bugs (MoKB) archive. http://projects.info-pull.com/mokb/.

[11] US-CERT. US-CERT Vulnerability Notes Database. http://www.kb.cert.org/vuls/.

[12] Devik and sd. Linux On-the-fly Kernel Patching without LKM. http: //www.phrack.com/issues.html?issue=58&id=7.

[13] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS’10), 2010.

[14] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS’08), 2008.

[15] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, 2009.