Kernel Memory Protection by an Insertable Hypervisor Which Has VM Introspection and Stealth Breakpoints

Kuniyasu Suzaki[1], Toshiki Yagi[1], Kazukuni Kobara[1], and Toshiaki Ishiyama[2]

[1] National Institute of Advanced Industrial Science and Technology, Japan
{k.suzaki,yagi-toshiki,k-kobara}@aist.go.jp
[2] FFRI, Inc., Japan
ishiyama@ffri.jp

Abstract. Recent device drivers are under threat of targeted attack called Advanced Persistent Threat (APT) since some device drivers handle industrial infrastructure systems and/or contain sensitive data e.g., secret keys for disk encryption and passwords for authentication. Even if attacks are found in these systems, it is not easy to update device drivers since these systems are required to be non-stop operation and these attacks are based on zero-day attacks. DriverGuard is developed to mitigate such problems. It is a light weight hypervisor and can be inserted into pre-installed OS (Windows) from USB memory at boot time. The memory regions for sensitive data in a Windows kernel are protected by VM introspection and stealth breakpoints in the hypervisor. The hypervisor recognizes memory structure of guest OS by VM introspection and manipulates a page table entry (PTE) using stealth breakpoints technique. DriverGuard prevents malicious write-access to code region that causes Blue Screen of Death of Windows, and malicious read and write access to data region which causes information leakage. Current implementation is applied on pre-installed Windows7 and increases security of device drivers from outside of OS.

Keywords: Computer Security, Information Leakage, Virtual Machine Introspection, Stealth Breakpoints.

1 Introduction

Device drivers are key components on current operating systems since they bridge between logical space of the operating system and physical space of devices. As current device drivers are flexible and intelligent, most of them are loaded after booting and plugged-in to a kernel. They are stackable to an existing device driver and add intelligent functions. The feature enables to add access control, encryption, and compression on an existing device driver. The intelligent functions include sensitive data in a device driver (e.g., secret keys for disk encryption, passwords for authentication, tables of access control, etc).

Device drivers were thought to be safe since they run in privilege mode. However, device drivers become a target of attacks as the importance is increased, and the vulnerability is revealed. For example, Stuxnet[5] and Duqu[3] are famous attacks for device drivers. These attacks use vulnerabilities of device drivers in a commodity operating system as a steppingstone of attacks on a real device (e.g., nuclear reactor, chemical plant, etc). They are targeted attacks and called Advanced Persistent Threat (APT). Most of them are zero-day attacks and signature based security tools cannot detect these attacks. Furthermore if the target is a critical infrastructure or an industrial control system, the availability is important. The countermeasures must be taken without stopping the operating system.

In order to mitigate the problems, we propose DriverGuard which is a light weight and insertable hypervisor to a pre-installed OS (Windows). DriverGuard has a function of VM introspection[6,7,9], and recognizes the data structure of Windows7. It also has stealth breakpoints technique[14] which manipulates page table entries (PTE) on a shadow page table of hypervisor. The combination of VM introspection and stealth breakpoints in an insertable hypervisor prevents malicious write-access to code region which causes Blue Screen of Death of Windows, and malicious read and write access to data region which causes information leakage.

This paper is organized as follows. Section 2 briefs related works and Section 3 introduces threat model for DriverGuard. Section 4 reviews countermeasures for the threats. Section 5 describes the design of DriverGuard. Section 6 reports the current implementation. Section 7 discusses some issues for DriverGuard, and Section 8 summarizes our conclusions.

2 Relates Work

Device drivers are recognized as a weak point in kernel space, and many protecting methods have been proposed.

Nooks[13] is a famous research for protecting device drivers. Nooks offers reliable subsystem that isolates a kernel from device drivers. It uses special memory management system to limit access to the device driver. The limitation of access resembles DriverGuard, but the aim of Nooks is to enhance OS reliability from failures. The limitation of accesses on Nooks is also used for security, but Nooks does not prevent information leakage from device drivers.

OS2, which is developed by IBM and Microsoft in 90s, uses protection rings architecture of IA-32 that offers one more privilege level for device drivers. It can increase the security level, but it requires operating system to recognize the ring levels and makes difficult to develop a device driver.

Instead of protection ring architecture, virtualization architecture (e.g., Intel VT or AMD SVM) is developed and used widely. It offers a mode for virtualization that is independent of OS and makes easy to make a hypervisor. Some hypervisors have a function to recognize behavior of OS, called to VM introspection[6,7,9]. It also makes possible to manipulate device drivers and prevent attacks on them.

HUKO[15] is a hypervisor-based integrity protection system designed to protect commodity OS kernel from untrusted device drivers. HUKO manipulates CR3 register of IA-32 architecture which manages page table entry (PTE), and separates virtual memory space between the kernel and device drivers. The device drivers use isolated virtual memory space from the kernel. On the other hand, DriverGurad manipulates PTE contents, and the access to the memory for sensitive data is protected by Stealth Breakpoints technique [14]. DriverGurad does not require additional virtual memory space for device drivers.

SecVisor [10] is a hypervisor that ensures code integrity for OS kernels. It protects the kernel against code injection attacks, which works as same to DriverGuard. Furthermore, SecVisor uses the IO Memory Management Unit (IOMMU) to protect kernel code from Direct Memory Access (DMA) access, which is more progressive than DriverGuard. However, SecVisor requires to add 2 hypercalls in a target OS kernel. The feature is not accepted our target because DriverGuard treats Window7 which does not allow to customize the kernel. DriverGuard detects the code and data region using VM Introspection. It requires small customization to allocate sensitive data, but the customization is trivial because it only changes memory allocation method with normal Windows' function. In addition, SecVisor requires customization on bootstrap code in Linux because SecVisor has to be loaded as a part of Linux kernel. On the other hand, DriverGuard is insertable hypervisor which uses chain-loader of GRUB and does not require the change of the existing boot procedure.

Taint tracking technique is useful to prevent information leakage. The technique tracks data flow and finds illegal usage of data. Some hypervisors integrate taint tracking mechanism and are used to find information leakage dynamically. For examples, TTAnalyze [1], TEMU[12], V2E[16] and Ether[4] are developed on open source hypervisor Xen or QEMU. They are used to analyze malware behavior because they can avoid anti-debugger mechanism in a malware. Unfortunately, they take much time to track data flows because they have to monitor data flows aside from the original processing. Heavy overhead is not accepted to prevent sensitive data at normal operation. Fortunately, DriverGuard does not need to track data flow because DriverGuad knows the region of sensitive data and only have to prevent malicious access to there. It does not cause extra overhead to track data flow.

3 Threat Model

We assume two types of threat model for DriverGuard. One of the threats is code injection attack to a device driver's code, and the other is information leakage from the device driver's data. Most of them are zero-day attacks, and security patch and security signature are not available.

The aim of code injection attack is to take control and run malware. The attack re-writes an existing code on memory and passes control to the malware. Even if the attack cannot get the full control, failure is enough for attackers on an infrastructure system because the aim is to stop or runaway the system. Therefore, security systems for infrastructure have to prevent Blue Screen of Death (BSoD) on Windows, even if the system is shrunk.

The other threat is stealing or re-writing sensitive data of device driver's data region memory. Current device drivers are intelligent and have some sensitive data.

Attackers try to read or write the sensitive data with some techniques (e.g., buffer overflow). The access to the sensitive data should be limited to the legitimate device driver's code only.

4 Requirement for Countermeasures

DriverGuard is used for protecting device drivers in industrial infrastructure systems. These systems have already established and security features must be added on the systems. Furthermore, some attacks to device drivers exploit a previously unknown vulnerability in the operating system and cannot be protected by the operating system itself. As a countermeasure of zero-day attack, anomaly behavior detection is one approach, but it cannot avoid false-negative.

DriverGuard offers a white-list approach. The user must notify the hypervisor identifications of device drivers. The identifications are used to find the region of legitimate code when the drivers are loaded. The device drivers also must notify the hypervisor the memory region for sensitive data. After the setup of DriverGuard, the code region of device driver is not re-written, and sensitive data region is accessed by legitimate code of device driver only. The accesses to protected regions are monitored by the hypervisor of DriverGuard, which works as a small Trusted Computing Base (TCB).

In order to satisfy the requests, DriverGuard uses insertable hypervisor which has VM introspection and stealth breakpoints.

4.1 Hypervisor for an Existing OS

The hypervisor has to offer full virtualization in order to boot pre-installed OS as a guest OS. The hypervisor should be as light as possible to make a small impact on pre-installed OS. Current popular hypervisors (e.g., KVM, Xen, VMware) require a control OS (host OS), even if the hypervisor is type I (Bare-metal hypervisor. The example is Xen.) or type II (Hypervisor hosted by an OS. The example is KVM). A control OS requires much memory and storage. It is not suitable for our purpose. In order to solve the problem, we build DriverGuard on the hypervisor called BitVisor[11] which does not require a control OS.

Furthermore, most hypervisor has a fixed device model (QEMU-Device model on KVM and Xen). The device model requires remapping from pseudo devices on a VM to real devices. Even if these hypervisors allows to boot a pre-installed OS, they require to install device drivers for pseudo devices on a pre-installed OS, which is not acceptable for our purpose. BitVisor has a para-passthrough mechanism which offers bare-metal devices to a guest OS and does not require any change of pre-installed OS.

A pre-installed OS is stored on a real hard disk and users do not want to change the contents. It means hypervisor is requested to be inserted from other devices at boot time.

4.2 VM Introspection

The hypervisor for DriverGuard has to recognize the memory map and behavior of the guest OS since it needs to know the memory region for code and sensitive data. The function is called VM introspection [6,7,9]. Unfortunately, most hypervisors do not have the function because they have to solve semantic gaps between guest OS and hypervisor.

BitVisor also has no function for VM introspection. Fortunately we can use GreenKiller [8] which offers VM introspection on top of BitVisor. GreenKiller recognizes the memory map of Windows and hooks some system calls. We build DriverGuard on GreenKiller.

4.3 Stealth Breakpoints

Debugger is a fundamental tool to analyze a malware. It places a breakpoint on an instruction, where the control goes to a debugger from the targeted code. The targeted instruction is replaced with an instruction of software interrupt. On IA-32 architecture, INT 3H (0xCC) instruction is used.

Breakpoints are useful for debugging, but they are detected by some type of malware. If the malware finds break points (INT 3H instructions) on its code region, it recognizes that it is analyzed. The function is called Anti-Debugger. The malware with Anti-Debugger changes its behavior in order to prevent the analysis.

Stealth breakpoints technique[14] is used to solve this drawback. The technique manipulates page table entry (PTE) which indicates the address of the page. The PTE content is changed in order to cause a page fault, when an access is issued to the page. The page fault is carried to stealth breakpoints as a break point. Stealth breakpoints changes the status of PTE and allows the access to the page. After that, stealth breakpoints sets single step mode and returns control to the original code. The original code causes an exception of the single step, which is carried to stealth breakpoints again. At the exception handler of the single step, stealth breakpoints disables the page in order to work as breakpoint again, and releases single step mode. Then, the control is returned the original code.

Stealth breakpoints works as normal break point and countermeasure for Anti-Debugger of malware. However, the cost is heavy because page fault is slower than software interrupt. Furthermore, stealth breakpoints hooks accesses to the region which is outside of the region for sensitive data in the protected page. It will make performance degradation when it applied on code region, which accessed frequently. DriverGuard avoids this problem by applying stealth breakpoints on heap region which includes sensitive data only.

5 Driverguard

DriverGuard is build on top of GreenKiller[8] which has VM introspection. GreenKiller is based on BitVisor[11] which is a thin hypervisor with para-passthrough. BitVisor offers bare-metal devices to guest OS and does not require any changes on a pre-install OS. Current target OS is pre-installed Windows7. This section describes key features of DriverGuard.

5.1 Inserting DriverGuard in an Pre-installed Windows

In order to load DriverGuard before booting Windows, we used USB boot and chain-loader. It does not require any change on the hard disk. Figure 1 show the steps.

Most current BIOS can select USB storage as a boot device. DriverGuard is loaded from the GRUB bootloader on the MBR of USB storage. DriverGuard occupies the VMX root mode of Intel VT and remains on the memory. After that, DriverGuard returns the control to the MBR of the booted device (USB memory). The GRUB has a function called chain-loader which sends the control to another bootloader. The control goes to the MBR of hard disk which includes bootloader of pre-installed Windows. The bootloader boots Windows besides DriverGuard hypervisor.

The kernel of Windows7 has a security function of ASLR (Address Space Layout Randomization) It allocates the starting address of the kernel at random and prevents buffer overflow attacks. DriverGuard must find the starting address for VM introspection. Current implementation detects the starting address by linear search technique. A MD5 hash of beginning contents of the kernel is used as an identifier, which is passed to the DriverGuard as a parameter of bootloader GRUB. The DriverGuard searches the starting address with the MD5 when the kernel is loaded. MD5 hash is used instead of SHA-1 because GRUB has a size limitation of arguments and must pass some other identifiers mentioned in the next section.

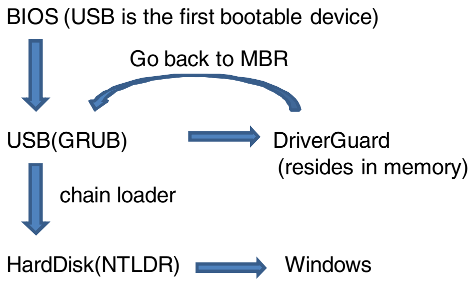

Fig. 1. Method to insert DriverGuard before booting Windows

5.2 Set Up DriverGuard

DriverGuard has to recognize which device drivers are protected and where are the protected memory regions. The setting up of DriverGuard has three steps, which are illustrated in Figure 2.

The first step is identification of device driver protected by DriverGuard. Identification is based on MD5 hash value of a binary of device driver. The identification is passed as parameters of bootloader GRUB.

The second step is to recognize the code region of the protected device drivers. DriverGuard still knows the identifications, but does not know when and where the codes of device drivers are loaded. DriverGuard uses a mechanism of VM introspection which comes from GreenKiller.

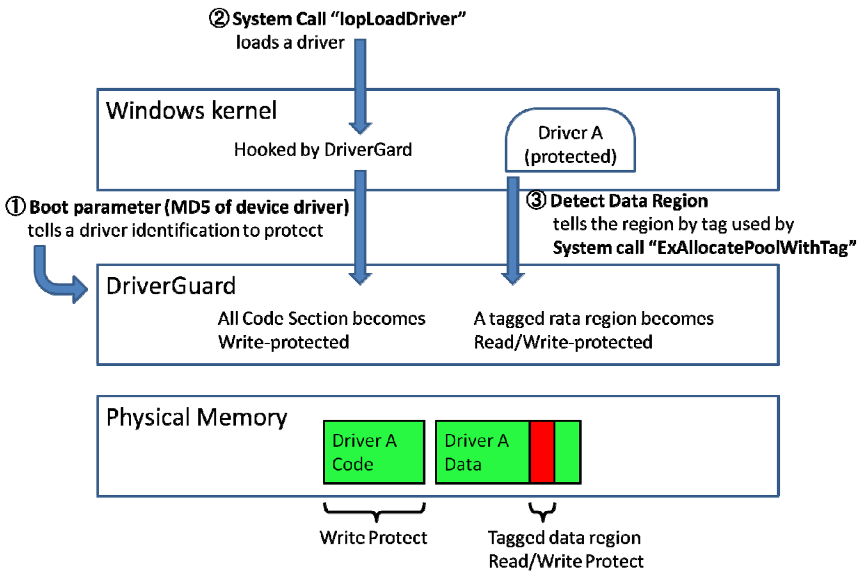
② System Call "IopLoadDriver"
loads a driver
Windows kernel
Hooked by DriverGard
Driver A (protected)
① Boot parameter (MD5 of device driver)
tells a driver identification to protect
③ Detect Data Region
tells the region by tag used by
System call "ExAllocatePoolWithTag"
DriverGuard
All Code Section becomes Write-protected
A tagged rata region becomes Read/Write-protected
Physical Memory
Driver A Code
Driver A Data
Write Protect
Tagged data region Read/Write Protect

Fig. 2. Setup procedure of DriverGuard

DriverGuard hooks "IopLoadDriver" system call to recognize the protected device drivers. IopLoadDriver is an internal function of ntkrnlpa.exe, which inserts a device driver to the kernel space. DriverGuard recognizes the memory map of Windows7 and replaces an instruction of IopLoadDriver with INT 3H (0xCC) instruction as a break point of debugger. When IopLoadDriver is called, the break point causes an exception and switches to DriverGuard. The DriverGuard analyzes the data structure of the created process using identification (MD5 hash value), which allows to detect a protected device driver and know the code region. After the analysis, DriverGuard returns the control to the break point with the replaced original instruction.

The third step is to recognize protected region for sensitive data. Current implementation requires to customize the device driver to tell the region of sensitive data. The region must be allocated dynamically with a "tag" caused by "ExAllocatePoolWithTag" function. The VM introspection of DriverGuard detects memory region using the tag caused by ExAllocatePoolWithTag. The pages which used by tagged memory are protected by DriverGuard.

DriverGuard recognizes that the request comes from the code of legitimate device driver and registers the memory region to be prohibited from read and write accesses of other code. The code region is detected by VM Introspection when the driver is installed by IopLoadDriver. After the setting up DriverGuard, the code and sensitive data region are protected from malicious accesses.

5.3 Protecting Code Region

DriverGuard protects the code of device drivers from write-access, but the code region is mapped as read-only by Windows7 already. Therefore, DriverGuard does not need to change the permission in general. However, when a write-access is issued to the read-only memory, the exception handler is called as a Bug Check Code (0xBE: ATTEMPTED_WRITE_TO_READONLY_MEMORY), which causes Blue Screen of Death (BSoD) of Windows. If an attacker wants to stop the Windows, the attack means a success.

In order to prevent an attack, DriverGuard hooks the exception handler and causes an infinite loop. The infinite loop runs as low Interrupt ReQuest Level (IRQL) and causes high CPU load on Windows7. However, it is interrupted by other higher IRQL, the user can cope with the situation.

5.4 Protecting Data Region

DriverGuard allows memory accesses on the protected region from processes that officially use the registered device drivers. The processes loads registered .sys files only. The other processes which loads registered .sys files with others are recognized as malicious processes by DriverGuard.

The region of sensitive data is informed by a protected device driver as mentioned in Section 5.2. DriverGuard protects the memory region using stealth breakpoints technique on shadow page table. Shadow page table is pseudo page table that offers a virtual memory on a virtual machine. The management unit is 4KB page, and the protected region is rounded to the 4KB unit.

Figure 3 shows the data protection that uses stealth breakpoints. Each process has its own page directory and a set of page table entries (PTE), which are virtualized as shadow page table by DriverGuard. A PTE has two addresses for virtual memory and physical memory in order to map them. The address of page directory is in CR3 (page directory register) when the process is running. The PTEs are set by the operating system.

DriverGuard manages page table entries and changes the P-bit (persistent bit). P-bit is used for swapping and indicates that the page exists in the memory or swaps out. The P-bit for a page table entry for a protected page is set to 0 by DriverGuard. It means all access to the page causes a page fault. The page fault is hooked by DriverGuard and analyze whether the access comes from legitimate code or not. If an access comes from non-legitimate code (process B in Figure3), the access is failed. The DriverGuard decides it as malicious access and brings to an infinite loop with low IRQL. Even if an access comes from legitimate code and is allowed, a page fault occurs.

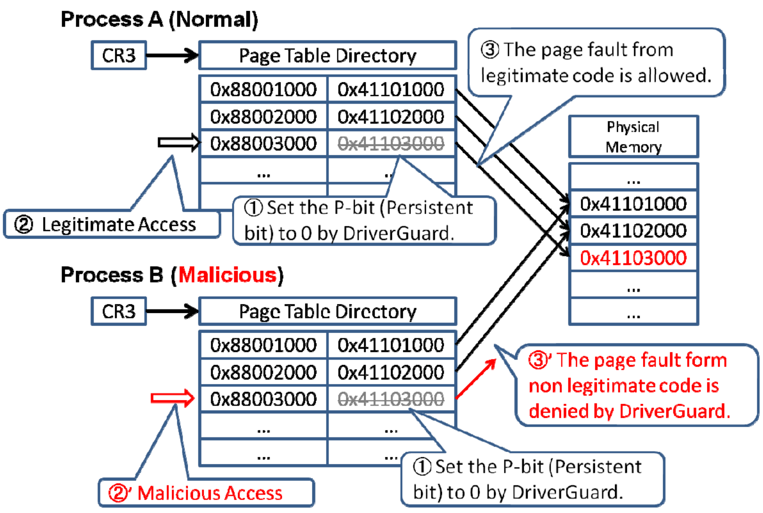
Process A (Normal)
CR3
Page Table Directory
0x88001000 0x41101000
0x88002000 0x41102000
0x88003000 ~~0x41103000~~
② Legitimate Access
① Set the P-bit (Persistent bit) to 0 by DriverGuard.
③ The page fault from legitimate code is allowed.
Physical Memory
0x41101000
0x41102000
0x41103000
Process B (Malicious)
CR3
Page Table Directory
0x88001000 0x41101000
0x88002000 0x41102000
0x88003000 ~~0x41103000~~
②' Malicious Access
① Set the P-bit (Persistent bit) to 0 by DriverGuard.
③' The page fault form non legitimate code is denied by DriverGuard.

Fig. 3. Data protection mechanism of DriverGuard. P-bit (persistent bit) of page table entry on shadow page table is set to 0 to cause page fault for any accesses. DriverGuard investigates all page faults at the page, and malicious access to the sensitive data is denied.

Figure 4 shows the procedure of stealth breakpoints in DriverGuard. A protected driver runs on VMX non-root mode and DriverGuard runs on VMX root mode. When an access is issued to the sensitive data, the access causes a page fault because the P-bit for the page table entry is set to 0. It causes VMEnter to change the VMX root mode and invokes DriverGuard. DriverGuard investigates the address of instruction that cased the page fault. If the address does not come from the legitimate code, DriverGuard decides it as malicious access and bring to an infinite loop with low IRQL.

When the access comes from the legitimate code, DriverGuard goes to the procedure of stealth breakpoints. It sets a hardware break point to the next instruction. DriverGuard sets P-bit 1 and allows the access. After that DriverGuard cause VMExit to bring the control back to the driver. The driver can access to the sensitive data. After the access, the next instruction is trapped immediately by hardware breakpoint and causes VMEnter to invoke DriverGuard. DriverGuard clear the hardware breakpoint and sets the P-bit set 0 again. After that DriverGuard causes VMExit and returns the control to the driver.

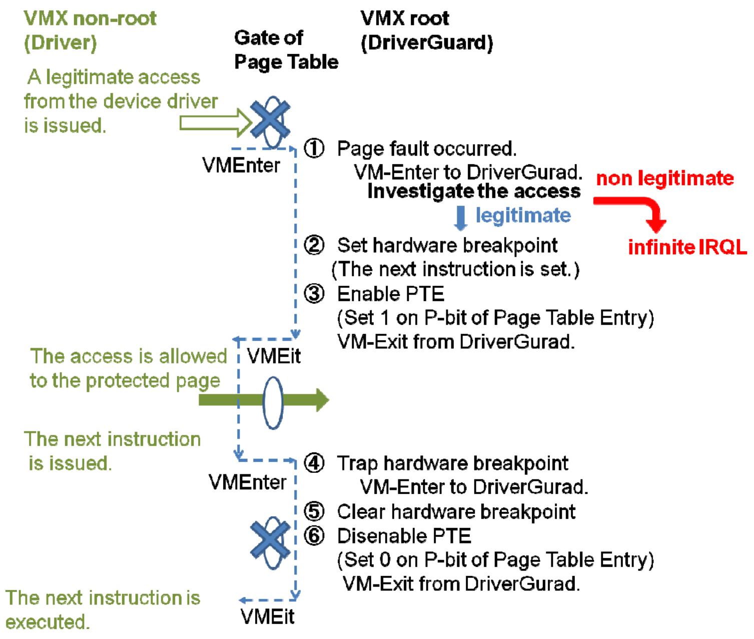
VMX non-root (Driver)
Gate of Page Table
VMX root (DriverGuard)
A legitimate access from the device driver is issued.
VMEnter
① Page fault occurred.
VM-Enter to DriverGurad.
Investigate the access
non legitimate
legitimate
infinite IRQL
② Set hardware breakpoint
(The next instruction is set.)
③ Enable PTE
(Set 1 on P-bit of Page Table Entry)
VM-Exit from DriverGurad.
The access is allowed to the protected page
VMEit
The next instruction is issued.
VMEnter
④ Trap hardware breakpoint
VM-Enter to DriverGurad.
⑤ Clear hardware breakpoint
⑥ Disenable PTE
(Set 0 on P-bit of Page Table Entry)
VM-Exit from DriverGurad.
The next instruction is executed.
VMEit

Fig. 4. Procedure of stealth breakpoints in DrvierGuard

The implantation of stealth breakpoints in DriverGuard is different from the Original. Original stealth breakpoints uses single step mode to hook the next instruction, but stealth breakpoints in DriverGuard uses hardware break point. Original stealth breakpoints is designed to hide hardware break point from a malware, but DriverGuard does not need to care about detection from a malware. Furthermore, the implementation is easy, because DriverGuard can use the information which is recorded at VMEner. VMEnter uses VMCS (Virtual Machine Control Structure) to record the status information of VMX mode. The information includes the instruction which causes VMEnter, and DriverGurad know the address of the next instruction. DriverGuard easily sets a hardware break point to the next instruction. It makes easy to implement.

Page fault occurred by stealth breakpoints does not cause access to a disk, which is quicker than normal page fault. Normal page fault takes milli-second to get data from the disk, but page fault of stealth breakpoints takes micro-second order. The real performance is showed in the next section.

6 Implementation

This section reports current implementation issues on hardware, guest OS, and the performance.

6.1 Limitation

DriverGuard requires a CPU which has Intel VT since DriverGuard depends on hardware virtualization assists. The VMX root mode on the CPU is occupied by DriverGuard and another hypervisor cannot use the mode.

The guest OS cannot use page size extension (PSE) since a unit of page size becomes 4MB. It is too large to protect sensitive data. 4KB page is common page size for many operating systems and good for DriverGuard.

The guest OS is limited to Windows7 ServicePack1 since the VM introspection of DriverGuard is design for the OS. Other Windows may be applied by DriverGuard, but we have not tested yet.

Swapping page mechanism must be disabled since current DriverGuard cannot follow the swapped-out pages. Hibernation is also unsupported.

Some device drivers are loaded at the boot time, but DriverGuard cannot recognize them since they do not use IopLoadDriver system call. Some of device drivers are necessary to boot Windows (e.g., storage driver for root file system and video card driver), and they are recognized as parts of Windows kernel. Therefore, current DriverGuard does not care them.

6.2 Performance

We measured the performance of DriverGuard on Lenovo ThinkCentre (Intel Core2 Duo E6850 3.0GHz, 2 GB memory). The size of current DriverGuard is 16.2 MB. It is not significant because it includes VM introspection for Windows7.

The DriverGuard is inserted at boot time. The insertion of DriverGuard took about 6 seconds, which excludes the loading time of the GRUB. The boot time of Windows7 on DriverGuard took 40 seconds while the boot time of original Windows7 took 17 seconds. The overhead was caused by setup OS, which was pressure on a hypervisor. However, we did not feel any stress to use the Windows 7 on DriverGuard after the booting.

The Windows7 on DriverGuard recognized 1.83GB memory while the normal Windows7 recognized 2.00GB memory. The difference size of memory (about 170MB) was used by DriverGuard. The size is not big as a hypervisor which has VM introspection. When another hypervisor requires host OS for VM introspection, much more memory will be used.

We made a pseudo malware to attack a device driver and confirmed that the malicious write-access to sensitive data was detected and went to an infinite loop at low IRQL. The CPU load on Windows 7 went to 100%, but the keyboard and mouse were active and we could control the windows7.

Figure 5 shows the procedure for stealth breakpoints in DriverGuard, and the elapsed time which took on Core2 Duo E6850. The time is measured by the function equipped in BitVisor and the resolution is 1 micro-second. There were no time difference for read and write accesses.

The elapsed time between page fault and enabling PTE took 5 micro-seconds. The elapsed time by trapping hardware breakpoint took 18 micro-seconds. It took 22 micro-seconds by the end of stealth breakpoints in DriverGuard. The measurement was achieved in DrvierGurad and did not include the time for the first VMEnter and the last VMExit.

The time is heavy for one access to normal memory, but the region includes sensitive data. The overhead is acceptable for sensitive data which are used a few times, for example, sensitive data for authentication.

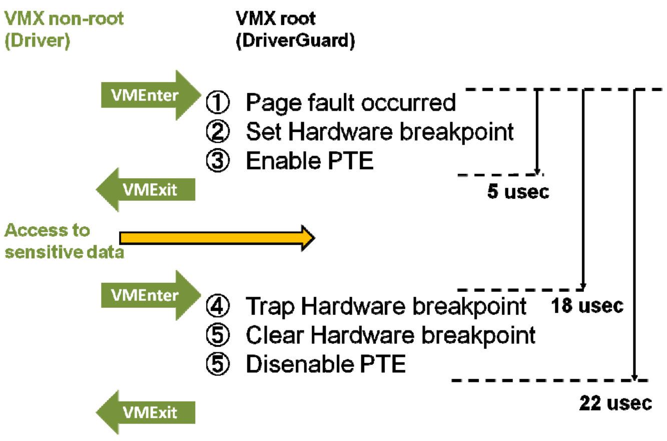

Fig. 5. Performance in DriverGuard

The most part of the elapsed time was spent between enabling PTE and trapping hardware breakpoints, which includes 2 switches between VMX root mode and VMX non-root mode. It causes VMEnter and VMExit which spend thousands of CPU cycles on Core2 Duo. Furthermore, DriverGuard uses shadow page table and takes time to manage it. If the time for the switches becomes short, DriverGurad can improve the performance. This issue is discussed in Section 7.

7 Discussions

Current implementation assumes that DriverGuard is inserted at boot time securely. There is, however, no method to verify the procedure. We have a plan to include trusted boot which records the procedure in a secure chip; Trusted Platform Module (TPM). The recorded data in the TPM is tamper-proof, and can be sent to a trusted third party so that the receiver can verify the boot procedure. When the data is exported from a TPM, the data is digitally signed with a secret key in the TPM and is verified with the public key of the TPM. If DriverGuard utilizes this mechanism, the integrity is confirmed.

Return oriented programming (ROP) is not prevented by DriverGuard since it reads loaded code only and does not require write-access to the code. In order to prevent such attacks, we have a plan to improve DriverGuard to protect read access from malicious processes.

DriverGuard employs white-list approach. The granularity of it can be categorized as middle since the unit of white-list is application software. The smallest granularity is a system call offered by Korset[2]. Korset analyzes the source code of an application and makes control flow graphs (CGF) of system calls in order to trace the behavior. If the behavior of the process does not follow the CFG, Korset alerts that the process is intruded by malware. It offers very strict white-list and can detect any intrusions. We are studying how to introduce the same idea to DriverGuard.

Current implementation uses shadow page table that is virtual memory emulated by software, and does not utilize hardware assist for memory virtualization called nested page table. Current X86 architecture CPUs have the function, for examples, Intel's EPT (Extended Page Table) or AMD's NPT (Nested Page Table). Some reports show the performance improvement caused by Intel EPT and AMD NPT. In order to receive the benefit, DriverGuared have to change the code. Fortunately, SecVisor[10], which modifies page table entry as DriverGuared does, shows two implementations on shadow page table and AMD's NPT. We will refer the implementation of SecVisor and revise DriverGuard. It will improve the performance of DriverGuard and make more useful for yours.

8 Conclusions

Current device drivers are under threat of targeted attack since they include sensitive data and control industrial control systems. The operations must be non-stop and protected from attacks without change after booting. We proposed DriverGuard to prevent malicious access to the device driver's memory. It is a light weight hypervisor based on GreenKiller and BitVisor. DriverGuard inherits para-passthrough and can be inserted from USB to a pre-installed OS at boot time. DriverGuard recognizes the memory structure of Guest OS (Windows7) and hooks some system calls by VM introspection of GreenKiller.

DriverGuard prevents malicious write-access to code region, and read and write-access to data region of protected device drivers. The code region is set as read-only by Windows, and it causes Blue Screen of Death when a write access is issued on the region, which is a kind of death attack. DriverGuard hooks the exception handler and prevents to go to Blue Screen of Death. The control hooked by DriverGuard is brought to an infinite loop of Low Interrupt ReQuest Level (IRQL). It caused an overhead on the guest OS, but the user still control the OS. This feature is useful for infrastructure systems and industrial control systems.

The protection on data region is based on stealth breakpoints technique and manipulates page table entries of shadow page table. Read and write accesses to the protected data region is also hooked and investigated the address of the instruction which cause the page fault. If the instruction is not a part of the legitimate device driver's code, the access is brought to an infinite loop of IRQL. When the instruction is in the legitimate code, the access is allowed by the treatment of stealth breakpoints. The implementation uses hardware breaking point.

The implementation was applied on pre-installed Windows7 and the performance was measured. Most part of the overhead of stealth breakpoints is estimated to the operation of VMExit and VMEnter. The result shows that DriverGuard is acceptable for operation which accesses sensitive data a few times, such as authentication. It increases security of device drivers from outside of OS.

Acknowledgement. This work was in part supported by the Strategic Information and Communications R&D Promotion Programme (SCOPE) of the Ministry of Internal Affairs and Communications, Japan.

References

[1] Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A Tool for Analyzing Malware. In: 15th European Institute for Computer Antivirus Research, EICAR (2006)

[2] Ben-Cohen, O., Wool, A.: Korset: Automated, Zero False-Alarm Intrusion Detection for Linux. In: Linux Symposium (2008)

[3] Bencsáth, B., Pék, G., Buttyán, L., Félegyházi, M.: Duqu: Analysis, Detection, and Lessons Learned. In: European Workshop on System Security, EuroSec (2012)

[4] Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions. In: ACM Conference on Computer and Communications Security, CCS (2008)

[5] Falliere, N., Murchu, L.O., Chien, E.: W32.Stuxnet Dossier, Symantec Security Response (2011)

[6] Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: 10th Annual Network & Distributed System Security Symposium, NDSS (2003)

[7] King, S.T., Dunlap, G.W., Chen, P.M.: Operating System Support for Virtual Machines. USENIX Annual Tech. (2003)

[8] Murakami, J.: FFR GreenKiller - Automatic kernel-mode malware analysis system. In: 12th Associates of Anti-Virus Asia Reserachers International Conference (2009) `http://www.fourteenforty.jp/research/research_papers/avar-2009-murakami.pdf`

[9] Nance, K., Bishop, M., Hay, B.: Virtual Machine Introspection: Observation or Interference? IEEE Security and Privacy 6(5) (2008)

[10] Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In: The 21st ACM Symposium on Operating Systems Principles, SOSP (2007)

[11] Shinagawa, T., et al.: BitVisor: A Thin Hypervisor for Enforcing I/O Device Security, Virtual Execution Environments, VEE (2009)

[12] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A New Approach to Computer Security via Binary Analysis. In: International Conference on Information Systems Security, ICISS (2008)

[13] Swift, M.M., Bershad, B.N., Levy, H.M.: Improving the Reliability of Commodity Operating Systems. In: 19th ACM Symposium on Operating Systems Principles, SOSP (2003)

[14] Vasudevan, A., Yerraballi, R.: Stealth Breakpoints. In: 21st Annual Computer Security Applications Conference, ACSAC (2005)

[15] Xiong, X., Tian, D., Liu, P.: Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extension. In: 18th Annual Network & Distributed System Security Symposium, NDSS (2011)

[16] Yan, L., Jayachandra, M., Zhang, M., Yin, H.: V2E: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis, Virtual Execution Environments, VEE (2012)