Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities

Jianfeng Pan, Guanglu Yan, and Xiaocao Fan, ***IceSword Lab, 360 Internet Security Center***

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/pan

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities

Jianfeng Pan, Guanglu Yan, Xiaocao Fan
IceSword Lab, 360 Internet Security Center

Abstract

Discovering vulnerabilities in operating system (OS) kernels and patching them is crucial for OS security. However, there is a lack of effective kernel vulnerability detection tools, especially for closed-source OSes such as Microsoft Windows. In this paper, we present Digtool, an effective, binary-code-only, kernel vulnerability detection framework. Built atop a virtualization monitor we designed, Digtool successfully captures various dynamic behaviors of kernel execution, such as kernel object allocation, kernel memory access, thread scheduling, and function invoking. With these behaviors, Digtool has identified 45 zero-day vulnerabilities such as out-of-bounds access, use-after-free, and time-of-check-to-time-of-use among both kernel code and device drivers of recent versions of Microsoft Windows, including Windows 7 and Windows 10.

1 Introduction

Software vulnerabilities have been well studied over the years, but they still remain a significant threat to computer security today. For instance, improper use of parameters or memory data can lead to program bugs, some of which can become vulnerabilities, such as time-of-check-to-time-of-use (TOCTTOU), use-after-free (UAF), and out-of-bounds (OOB) vulnerabilities. These vulnerabilities are often the root cause of successful cyberattacks. However, symptoms resulting from these vulnerabilities tend to be delayed and non-deterministic, which makes them difficult to discover by regular testing. Therefore, dedicated vulnerability identification tools that can systematically find software vulnerabilities are urgently needed.

There are usually two aspects in detecting vulnerabilities: path exploration and vulnerability identification. Combining path exploration with vulnerability identification tools is an effective way to detect vulnerabilities. Most fuzzing tools, such as AFLFast [12] and SYMFUZZ [16], only adopt path exploration to probe code branches. As a typical example of a path explorer, S2E [17], based on virtualization technology, combines virtual machine monitoring with symbolic execution to automatically explore paths. Vulnerability identification tools are used for recording exceptions (e.g., the abuse of parameters or illegal memory access) in the paths that have been probed. While we could have also investigated path exploration, the main focus of Digtool is vulnerability detection.

Depending on the detection targets, vulnerability identification tools can be classified into two categories: (1) tools for checking applications in user mode, and (2) tools for detecting programs in kernel mode. However, most of the current vulnerability identification tools, such as DESERVE [29], Boundless [15], and LBC [21], have been designed for applications in user mode. They cannot be directly used to detect kernel vulnerabilities. However, vulnerabilities in OS kernels or third-party drivers have a far more severe threat than user-level vulnerabilities. Thus, there is still a need for effective detection of kernel vulnerabilities.

Several Linux kernel vulnerability identification tools, such as Kmemcheck [32], Kmemleak [6], and KEDR [35], have been developed. They can effectively capture kernel vulnerabilities. However, since they rely on the implementation details and the source code of the OS, it is difficult to port these tools to other OSes, especially to a closed-source OS such as Windows.

In Windows OS, a notable tool for checking kernel vulnerabilities is Driver Verifier [28], which is used to detect illegal function calls or actions that might corrupt the system. While Driver Verifier is able to detect many potential bugs, it is an integrated system, but not a dedicated tool for detecting kernel vulnerabilities. For instance, it cannot be used to identify certain vulnerabilities, such as TOCTTOU vulnerabilities.

Vulnerability identification tools based on virtualization are much more portable to support different OSes, including closed-source ones. However, the current vulnerability identification tools based on virtualization, such as VirtualVAE [18] and PHUKO [38], are dedicated to detecting a single, specific type of vulnerabilities. Moreover, they have not been evaluated in detecting zero-day kernel vulnerabilities. It is worth noting that the virtualization-based tool Xenpwn [41] makes use of Libvmi [34] to discover vulnerabilities in para-virtualized devices of Xen (not for the Windows OS). It traces guest physical addresses through extended page tables (EPTs). However, it is not appropriate for monitoring guest virtual addresses.

For closed-source OSes such as Windows, it is even more difficult to build a vulnerability identification tool. We are neither able to insert detection code at compile-time to detect program errors like those tools for Linux, nor able to rewrite or modify the OS source code like Driver Verifier. Under these constraints, we adopt virtualization to hide the internal details of the Windows OS, and carry out the detection at a lower level, i.e., at the hypervisor. Therefore, a novel vulnerability identification framework named *Digtool* is proposed, which captures dynamic behavior characteristics to discover kernel vulnerabilities in the Windows OS by using virtualization technology.

Contributions. In short, we make the following contributions in this paper:

- A virtualization-based vulnerability identification framework, *Digtool*, is proposed to detect different types of kernel-level vulnerabilities in the Windows OS. It does not need to crash the OS, and thus it can *capture multiple vulnerabilities and provide the exact context of kernel execution*. It is designed to *be independent of kernel source code*, which enlarges its applicable scope. In addition, it does not depend on any current virtualization platform (e.g., Xen) or emulator (e.g., bochs).
- Based on the framework, virtualization-based detection algorithms are designed to discover four types of vulnerabilities, including *UNPROBE* (no probe, i.e., no checking on the user pointer to the input buffer), *TOCTTOU, UAF*, and *OOB*. These algorithms can effectively detect kernel vulnerabilities by accurately capturing their dynamic characteristics.
- With Digtool, we found *45 zero-day kernel vulnerabilities* from both Windows kernel code and third-party device driver code. These vulnerabilities had never been published before. We have made responsible disclosure and have helped the corresponding vendors fix the vulnerabilities. The root cause of some of the vulnerabilities is also analyzed in this paper.

The rest of this paper is organized as follows. In Section 2, we describe the background. In Section 3, we provide the overall design of the framework. In Section 4, we detail the implementation of Digtool, and, in Section 5, evaluate its effectiveness and efficiency. In Section 6, we discuss its limitations and directions for future research. In Section 7, we review the related work, and in Section 8 we conclude.

2 Background

UNPROBE, TOCTTOU, UAF, and OOB vulnerabilities have widely appeared in various programs including OS kernels. They can lead to denial-of-service attacks, local privilege escalation, and even remote code execution, which directly affect the stability and security of the victim program.

No checking of a user pointer to an input buffer could lead to a vulnerability that is denoted *UNPROBE* in this paper. Many kernel modules omit the checking for user pointers (especially when the user pointers are nested in a complex structure). According to the historical data of common vulnerabilities and exposures (CVEs), there have been many UNPROBE vulnerabilities in the Windows kernels, and there are also many such vulnerabilities in third-party drivers (e.g., the vulnerabilities in the experiment described herein). An UNPROBE vulnerability could result in an invalid memory reference, an arbitrary memory read, or even an arbitrary memory overwrite. Therefore, detection of UNPROBE is necessary. While fuzzing based on path exploration can help solve some problems, it is difficult to test all pointer arguments nested in complicated structures.

A *TOCTTOU* vulnerability stems from fetching a value from user memory more than once. Usually, a brittle system-call handler fetches a parameter for the first time to check it and for the second time to use it. Thus, an attacker has a chance to tamper with the parameter in the user space between the two steps. Consequently, the system-call handler will fetch and use the compromised parameter, which will lead to a TOCTTOU vulnerability. Similar to UNPROBE above, TOCTTOU could also result in an invalid memory reference, an arbitrary memory read, or an arbitrary memory overwrite. It is difficult to detect this type of vulnerability through fuzzing based only on path exploration. Bochspwn [24] was developed to identify many TOCTTOU vulnerabilities in the Windows kernel. However, its application is extremely restricted by the disappointing performance of the bochs emulator [25]. In addition, the bochs emulator

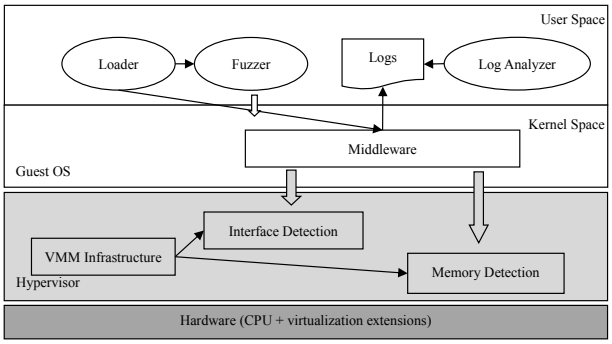


Figure 1: Digtool architecture.

cannot simulate all actual operations and functionalities of a real-world machine (e.g., inability to emulate certain real, hardware-specific kernel modules, such as modern video card drivers). As a result, Bochspwn cannot cover all of the kernel modules.

A *UAF* vulnerability stems from reuse of freed memory. An *OOB* vulnerability results from accessing memory that is beyond the bounds of allocated heaps or memory objects. In many cases, these vulnerabilities could lead to local privilege escalation. For the Linux OS, tools such as AddressSanitizer [36] have been released to detect these vulnerabilities. For the closed-source Windows OS, it is difficult for a third party to build such detection tools. Driver Verifier [28] proposed by Microsoft can be used to discover these types of vulnerabilities. However, it is much more likely to miss a vulnerability in some scenarios (e.g., the UAF detection scenario described in Section 4.3.1).

Digtool adopts virtualization technology to detect the above four types of vulnerabilities in Windows kernels and device drivers with better detection results. As a framework, it could also be used to detect some other types of vulnerabilities, such as double-free and information leakage, by expanding its detection algorithms.

3 Overview

The overall architecture of Digtool is illustrated in Figure 1. The subsystems and logic modules of the Digtool are distributed across user space, kernel space, and the hypervisor. The thin arrows in the figure indicate that there are direct invoking relationships or direct channels for passing messages between modules. The thick arrows illuminate that two modules act on each other indirectly via some event-triggering mechanisms.

One of the most important tasks for the hypervisor is to *monitor virtual memory access*. This is the basis for interface detection and memory detection. However, the memory monitor methods in current vulnerability identification tools are unsuitable for our scenario. Without source code, we cannot monitor memory access through patching source code like Driver Verifier [28], or through configuring compile-time instrumentation like AddressSanitizer [36]. Patching the system exception handler to intercept memory references by using page access rights is an alternative, but it will introduce significant, internal modifications in the kernel that may impact the stability of the OS and be the least portable. Binary rewriting could help to solve part of the problem. However, tools such as Pin [27] and DynamoRIO [13] work well in user mode, but it is difficult for these tools to work in kernel mode. Drk [5] tried to port the DynamoRIO to the kernel space for Linux, but it has not been updated for years, and there are few special tools for the Windows kernel. As an alternative, QEMU [11] or the recent extension PEMU [42] could be used to implement kernel program instrumentation for the Windows OS, but it is complicated and has a heavier effect on performance even without monitoring memory access.

Therefore, there is a clear need to develop an efficient alternative mechanism for tracing memory access outside a guest OS. As most programs run in virtual address space, we should focus more on the virtual address than on the physical address. Thus, the method of using EPT to trace *physical addresses*, like Xenpwn [41], cannot be directly used in our scenario, especially for the Windows OS, whose memory mapping between virtual and physical addresses is nonlinear. In view of the poor performance of Bochspwn [24], we did not adopt a full-stack emulator. In order to build a practical framework that focuses on the virtual address space, a *shadow page table (SPT)* based on hardware virtualization technology is employed to *monitor virtual memory access*, which is very different from Xenpwn and Bochspwn in both design and implementation.

In kernel space, the major work includes setting the monitored memory area, communicating with the hypervisor, and intercepting specified kernel functions. The monitored memory area depends on the type of vulnerability to be detected. It will be changed along with the occurrence of some kernel events (e.g., allocating or releasing memory). Hence, it is necessary to trace these events in kernel space. For communication, the service interfaces are exported by Digtool. Kernel code invokes these interfaces to request services from the hypervisor. In addition, some kernel functions of the OS should be hooked to trace some particular events. All of these tasks that should be reserved in kernel space make up the middleware.

The loader, fuzzer, and log analyzer are placed in user space to simplify the code and make the entire system more stable. The loader activates the hypervisor and loads the fuzzer that is used to probe program paths.

Thus, the behavior characteristics in the probed paths can be recorded for the log analyzer.

Unlike emulator-based tools (e.g., Bochspwn [24]), Digtool is able to run in a physical machine with this architecture design. *It is widely applicable to almost all main kernels and third-party drivers.*

3.1 Hypervisor Components

Digtool does not rely on any current hypervisor such as Xen or KVM, and we implemented our own hypervisor that contains three important components, including VMM infrastructure (VMM, i.e., virtual machine monitor, which is equivalent to a hypervisor), interface detection, and memory detection.

To begin with, VMM infrastructure checks the hardware environment and the OS version to ensure compatibility. It then initializes the hypervisor and loads the original OS into a VM. The initialization of the hypervisor mainly consists of the following tasks: (1) building SPTs to monitor virtual memory access in the guest OS, (2) initializing modules for tracing thread scheduling, and (3) establishing communication between the OS kernel and the hypervisor. As such, the interface detection and memory detection components can monitor and handle some special events.

Interface detection monitors the parameters passed from user-mode programs during the system-call execution. It traces the *use* and the *check* of these parameters to discover potential vulnerabilities. The SPTs are needed to monitor the *user memory space* during the system-call execution. As system calls are always invoked in kernel mode, we do not need to monitor user memory when the processor runs in user mode. Otherwise, many `VMEXIT` events will be triggered, which will bring a substantial decrease in performance. In order to focus on vulnerabilities in a limited scope of system calls, interface detection is able to configure the detection scope of system calls through correlative service interfaces. Thus, it can obtain the potential vulnerabilities in specified system calls.

Memory detection monitors the use of *kernel memory* in the guest OS to detect illegal memory access. The SPTs are used to monitor the kernel memory. To detect some specified types of vulnerabilities in different detection targets (e.g., the multi-user Win32 driver: `Win32k`), memory detection is able to set monitored memory area and configure detection targets. It also dynamically calibrates the monitored memory area when capturing events of memory allocation or deallocation. All of these are implemented through corresponding service interfaces. Thus, it will obtain the exact characteristics of potential vulnerabilities during the memory access process.

3.2 Kernel-Space Components

The middleware locates in the kernel space of the guest OS. It is used to connect the subsystems in the hypervisor and the programs in the user space. For example, before loading the fuzzer, we can set the detection scope of system calls through the configuration file. Then, the middleware transfers the configuration information and the fuzzer process information from the loader to the hypervisor. Thus, the hypervisor can detect vulnerabilities in the environment of the fuzzer process.

For interface detection, the middleware records all behavior events in log files through a work thread. The recorded data include system call number, event type, event time, instruction address, and accessed memory of the event. Thus, the log analyzer can detect potential UNPROBE and TOCTTOU vulnerabilities from the log files. Note that only the system calls in the detection scope are recorded, which is meaningful when the system calls are invoked frequently. The number of frequent system calls could be limited to reduce the performance cost and alleviate the stress on the log analyzer. We can then obtain more effective data with less performance overhead.

For memory detection, the middleware helps dynamically calibrate the monitored memory by hooking some specified memory functions. In order to obtain more relevant data and reduce performance cost, it also limits the areas of monitored memory and the scope of kernel code (e.g., the code segment of `Win32k`) through invoking the service interfaces. If a potential vulnerability is found, the middleware records it and interrupts the guest OS through single-step mode or a software interruption. Thus, the guest OS is able to be connected with a debug tool such as `Windbg`, and the exact context is obtained to analyze the vulnerability.

3.3 User-Space Components

There are three modules in the user space: loader, fuzzer, and log analyzer. The loader is used for loading the target process, after which Digtool provides a process environment for detecting vulnerabilities. The loader can also limit the detection scope of system calls and set the virtual addresses of the boundary for ProbeAccess events (which will be described in Section 4.2) through the configuration file.

The fuzzer is responsible for discovering code branches. It is loaded by the loader. In Digtool, the fuzzer needs to invoke the system calls in the detection scope, and discovers as many branches as possible in the code of a system call by adjusting the corresponding parameters. A higher path-coverage rate can certainly help achieve a more comprehensive test. However, as this

paper mainly focuses on the *vulnerability identification tool*, not path exploration, we will not go into much detail regarding the fuzzer or the path coverage.

The log analyzer is designed to discover potential vulnerabilities from log files. It extracts valuable information from the large amount of recorded data according to the characteristics of vulnerabilities. The log analyzer's vulnerability detection algorithm needs to be changed depending on the types of vulnerabilities (e.g., UNPROBE or TOCTTOU) to be detected, since we use different policy to detect them.

4 Implementation

In this section, we provide the implementation details of how we implement Digtool, especially its hypervisor components, including VMM infrastructure, interface detection, and memory detection. The implementation of other components, such as the middleware, loader, fuzzer, and log analyzer, is also described in this section.

4.1 VMM Infrastructure

The main task of VMM infrastructure is to initialize the hypervisor and provide some basic facilities. After initializing the hypervisor, it loads the original OS into a VM. Then, the hypervisor is able to monitor the OS through the facilities.

The initialization process runs as follows. In the beginning, Digtool is loaded into the OS kernel space as a driver that checks whether processors support hardware virtualization through `CPUID` instruction. If they support it, VMM infrastructure builds some facilities for the hypervisor. Then, it starts the hypervisor for every processor by initializing some data structures (e.g., `VMCS`) and registers (e.g., `CR4`). Finally, it sets the state of guest CPUs according to the state of the original OS. Thus, the original OS becomes a guest OS running in a VM.

The Intel developer's manual [23] can be referenced to obtain the implementation details of hardware virtualization. This paper mainly focuses on the modules that help to identify vulnerabilities. These modules include the virtual page monitor, thread scheduling monitor, CPU emulator, communication between kernel and hypervisor, and the events monitor. Among these, the CPU emulator and events monitor are associated with particular types of vulnerabilities, so these two parts will be described in corresponding subsections.

4.1.1 Virtual Page Monitor

Digtool adopts SPTs to monitor virtual memory access. To reduce performance cost, SPTs are only employed for the monitored threads (i.e., the fuzzer threads). For non-monitored threads, the original page tables in the guest OS are used. When thread scheduling occurs, the virtual page monitor needs to judge whether the new thread that will get control is a monitored thread. Only when it is a monitored thread, will a SPT be built for it. Thus, performance is optimized.

Figure 2 shows the workflow of the virtual page monitor for a monitored thread. Digtool adopts a sparse BitMap that traces virtual pages in a process space. Each bit in the BitMap represents a virtual page. If a bit is set to 1, the corresponding page needs to be monitored, and the P flag in its page table entry (PTE) of the SPT should be clear [note that the SPT is constructed according to the guest page table (GPT)]. Thus, access to the monitored virtual page will trigger a `#PF` (i.e., page fault) exception that will be captured by the hypervisor.

When the `#PF` exception is captured, the page-fault handler in the hypervisor will search for the BitMap. If the bit for the page that causes the `#PF` exception is 0, the page is not monitored. The SPT will be updated through GPT. Then, the instruction that causes the exception will re-execute successfully. If the bit is 1, it is a monitored page. Then, the Handle module will be used to handle this exception. It will (1) record the exception, or (2) inject a private interrupt (0x1c interrupt, which has not been used) into the guest OS. The recording process for the exception is described in the following (i.e., the part of shared memory described in Section 4.1.3). The private interrupt handler stores some information (e.g., the memory address that is accessed, and the instruction that causes the `#PF`) about the `#PF` exception, and then it connects to a debug tool by triggering another exception, such as software interruption, in the guest OS. After that, Digtool *"single steps"* the instructions in the guest OS by setting a `MTF` (monitor trap flag, which can be used in new version of processors) or `TF` (trap flag, which is used in old versions of processors) in the hypervisor. Meanwhile, the SPT is updated through GPT to make the instruction that causes the exception re-execute successfully. Because of `MTF` or `TF`, a `VMEXIT` will be triggered after executing one instruction in the guest OS, and then the hypervisor will get control again. Thus, the handler of `MTF` or `TF` in the hypervisor has a chance to clear the P flag, and the page will be monitored once again. Finally, it disables the `MTF` or `TF` to cancel the single-stepping operation.

We have noticed that, in most cases, we need to monitor a *memory region* rather than an entire memory page. A memory region covers only one part of a memory page or contains several pages. All of the memory pages owned by a monitored memory region should be traced. When a `#PF` exception is triggered, its handler needs to

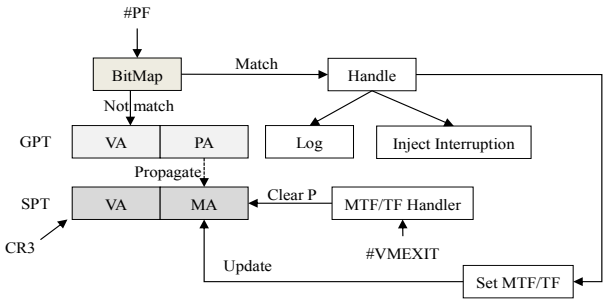


Figure 2: Workflow of virtual page monitor.

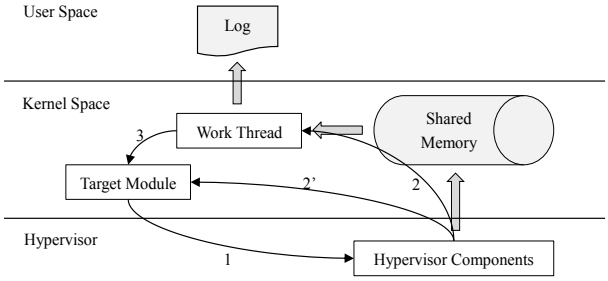


Figure 3: Communication between kernel and hypervisor via shared memory.

further recognize whether the address causing the #PF exception is in the monitored memory region.

4.1.2 Thread Scheduling Monitor

As discussed above, Digtool only focuses on the monitored threads. It needs to trace thread scheduling to enable detection for monitored threads and disable detection for non-monitored threads. Thus, it achieves better performance with more effective data. The method of the thread scheduling monitor is shown below.

In the Windows OS, the _KPRCB structure contains the running thread information for its corresponding processor. The _KPRCB is referenced by the _KPCR structure whose address can be obtained through the FS register (for x64 architecture, the GS register). The running thread of the current processor can be obtained through the following relationship:

FS->_KPCR->_KPRCB->CurrentThread.

With respect to how to acquire _KPRCB, the methods described in ARGOS [43] could be leveraged to uncover this data structure, though currently we use manual reverse engineering and internal Windows kernel knowledge to get it. Note that there are also other data structure agnostics approaches to detect kernel threads, such as using kernel stack pointer (e.g., [20]). After obtaining the _KPRCB structure, the CurrentThread member in the _KPRCB is monitored. Any write operation to the CurrentThread means a new thread will become the running state, and this will be captured by the hypervisor. If the new thread is a monitored thread, the virtual page monitor will be activated to detect vulnerabilities.

4.1.3 Communication Between Kernel and Hypervisor

The communication between kernel and hypervisor includes two main aspects. One is that the kernel component makes a request to the hypervisor, and the hypervisor provides service. The other is that the hypervisor sends messages to the kernel component, and the kernel component handles the messages. The former is mainly implemented by the service interfaces, and the latter is carried out through the shared memory.

Digtool exports some service interfaces for the kernel-space components. They can be directly invoked by kernel code. The service interfaces are implemented through a VMCALL instruction, which will trigger a VMEXIT to trap into the hypervisor. Thus, the service routines in the hypervisor can handle the requests.

The shared memory is applied to exchange data between the hypervisor and kernel code. The hypervisor writes the captured behavior information to the shared memory and notifies the kernel space components. Then, the kernel space components read and deal with the data in the shared memory. The workflow of the shared memory is shown in Figure 3.

The main data flow is represented by the thick arrows in the figure. When the hypervisor captures some behavior characteristics, it records them into shared memory. The middleware in the kernel space uses a work thread to read the data in the shared memory. It also records characteristic information into log files.

The following stream of instructions is shown by the thin arrows in Figure 3: (1) When the target module (which is being detected) triggers an event monitored by the hypervisor, a VMEXIT will be captured by the hypervisor. (2) The hypervisor records the event information into shared memory. If the shared memory is full, it will inject a piece of code into the guest OS. The code will notify the work thread to handle the data in shared memory (i.e., read them from shared memory and write them into log files). If the shared memory is not full, it will jump back to the target module (the arrow represented by 2'). (3) After notifying the work thread, the injected code will return to the target module and re-execute the instruction that causes the VMEXIT.

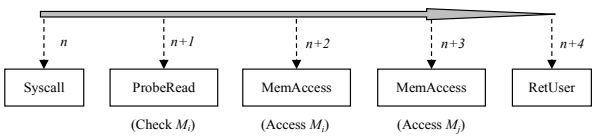


Figure 4: Example of recorded events during a system call.

4.2 Detecting Vulnerabilities at System Call Interface

Interface detection traces the execution process of system calls and monitors their parameters passed from user-mode programs. It then decides whether the *check* or the *use* of these parameters will create potential hazards.

Interface detection monitors the entire execution process of system calls from the point of *entering into kernel mode* to the point of *returning to user mode*. During this process, it monitors how the kernel code handles the user memory. Then, it records the behavior characteristics to analyze potential vulnerabilities. Interface detection is implemented by defining and intercepting different *behavior events* during the execution of system calls. These behavior events and their interception methods make up the events monitor.

Ten types of behavior events are defined in the event monitor: Syscall, Trap2b, Trap2e, RetUser, MemAccess, ProbeAccess, ProbeRead, ProbeWrite, GetPebTeb, and AllocVirtualMemory events. Particular combinations of these events can help locate potential vulnerabilities in the large amount of log data (e.g., two continuous MemAccess events suggest a potential TOCTTOU vulnerability). The behavior events recorded in the execution of a system call are shown in Figure 4. The boxes denote recorded events. The values (e.g., n and $n+1$) above the boxes are the event time (which only records order but not the actual intervals). The M_i and M_j under the boxes represent the user memory addresses accessed by the event.

In the Windows OS, fast system call, interruption of `0x2b`, and interruption of `0x2e` are the three entry points that allow user-mode code to invoke kernel functions. The fast system call adopts the `sysenter/syscall` instruction to go into kernel mode. The interruption of `0x2b` is used to return from a user-mode callout to the kernel-mode caller of a callback function. The interruption of `0x2e` is responsible for entering into kernel mode in older Windows OSes. In Digtool, the three entry points are traced by intercepting corresponding entries in the interrupt descriptor table (IDT) or MSR register. They are defined as three types of behavior events, which are marked as Syscall event, Trap2b event, and Trap2e event, respectively.

The return point is obtained by another way. When the control flow returns to the user mode, the processor will prefetch the user-mode instructions. Thus, Digtool obtains the point of returning to user mode by monitoring the user-mode pages access. This behavior event is marked as RetUser event.

After obtaining the two key points (i.e., the Syscall/-Trap2b/Trap2e event and RetUser event), interface detection will record the instructions that manipulate user memory between the two points. To achieve this, one important function is to monitor access to the user memory through SPTs. This behavior event is marked as a MemAccess event. It is noticed that, the user-mode pages are monitored only if the processor runs in kernel mode, and this will significantly reduce the performance cost.

To improve the efficiency of discovering and analyzing vulnerabilities, interface detection also defines and intercepts some other behavior events, including ProbeAccess, ProbeRead, ProbeWrite, GetPebTeb, and AllocVirtualMemory. Among the five events, the first three are used to record whether the user memory address has been checked by the kernel code, while the last two events suggest that the user memory address is legal; that is, there is no need to check it again and thus false positives can be reduced. These events are intercepted by hooking corresponding kernel functions, except for the ProbeAccess event.

GetPebTeb and AllocVirtualMemory events are used to reduce false positives. In order to improve the detection accuracy, we should focus on the user memory that is passed as parameters from the user-mode program, rather than on the memory that has been checked or that will be deliberately accessed by kernel code. For example, kernel code sometimes accesses a user memory region returned by a `PsGetProcessPeb` function or allocated by a `NtAllocateVirtualMemory` function during a system call. In these cases, the user memory is not a parameter passed from a user-mode program, and it has less of a chance of causing a vulnerability. Digtool defines GetPebTeb and AllocVirtualMemory events, respectively, to handle these cases. These events inform the log analyzer that the access to user memory is legal and that no bug exists.

In addition to invoking the `ProbeForRead` (i.e., ProbeRead event) or `ProbeForWrite` (i.e., ProbeWrite event) function, kernel code can also adopt direct comparison to check the legitimacy of the user memory address; for example, "`cmp esi, dword ptr [nt!MmUserProbeAddress (83fa271c)]`" where the `esi` register stores the user memory address to be checked, and the exported variable `nt!MmUserProbeAddress` stores the boundary of the user memory space. This kind of behavior event is

marked as a ProbeAccess event. We cannot intercept it by hooking a kernel function as this event is not handled by any kernel function. Moreover, there is no access to user memory space. Hence, we cannot intercept it through monitoring a MemAccess event either. For this particular type of event, the CPU emulator is proposed.

The CPU emulator is placed in the hypervisor to help obtain behavior characteristics that are difficult to obtain through regular methods. The CPU emulator is implemented by interpreting and executing a piece of code of the guest OS. Its workflow is shown in Figure 5. The DR registers are used to monitor the target memory. For the ProbeAccess event, the target memory stores the boundary that is used for checking the user-mode address. Usually, the exported variable, `nt!MmUserProbeAddress`, is one of the target memory. Kernel code can reference this variable directly or restore its value into another variable, such as `win32k!W32UserProbeAddress`. All of these variables are target memory. The address of target memory can be set by the configuration file of the loader, and then the hypervisor obtains the target memory through the middleware and monitor the memory access through DR registers. When the guest OS accesses target memory, the debug exception handler (DR handler) in the hypervisor will capture it. The handler updates the processor state of the CPU emulator (i.e., *Virtual CPU*) through that of the VM (i.e., *Guest CPU*). Thus, the CPU emulator is activated to interpret and execute the code of the guest OS around the instruction that causes the debug exception. Since the debug exception is a trap event, the start address for the CPU emulator is the instruction directly before the guest EIP register.

As the ProbeAccess event adopts direct comparison to check pointer parameters for a system call, the CPU emulator should focus on `cmp` instructions when it interprets and executes the code of the guest OS. The user-mode virtual address (UVA) for a pointer passed from a user-mode program is obtained by analyzing `cmp` instructions. Then, the ProbeAccess event is recorded in log files via shared memory.

There may be more than one UVA to be checked in a system call. The device driver may restore the value from target memory to a register and then check the UVAs by comparing them to the register separately. The maximum number of UVAs (represented by the letter `N` in Figure 5) could be set through the configuration file. After finishing `N cmp` instructions or a fixed number of instructions, the hypervisor will stop interpreting and executing, and return to the guest OS to continue executing the following instructions.

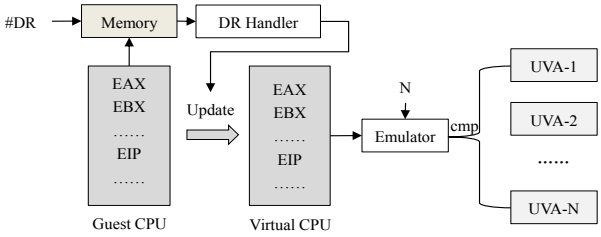


Figure 5: Workflow of CPU emulator.

4.2.1 Detecting UNPROBE Vulnerabilities

For the Windows kernel and device drivers, user memory (pointed by a user pointer) can be accessed under the protection of structured exception handling (SEH) at any time. It is safe to de-reference a user pointer if it points into the user space. Otherwise, it will bring on a serious vulnerability that is called UNPROBE in this paper. Theoretically, before using a pointer passed from a user-mode program, a system-call handler should check it to ensure that it points into the user-mode space. As a consequence, it will cause a ProbeAccess, ProbeRead, or ProbeWrite event before a MemAccess event under normal circumstances. If there is no such type of checking event before a MemAccess event, there may be an UNPROBE vulnerability in the kernel code.

To detect an UNPROBE vulnerability, we focus on whether there is a checking event before a MemAccess event, and whether the virtual addresses in the two events are the same. As discussed above, the ProbeRead and ProbeWrite events are directly obtained by hooking the checking functions in the kernel. The difficulty lies in the ProbeAccess event. In the Windows kernel, there is much code that checks parameters via direct comparison. Only intercepting ProbeRead and ProbeWrite events will result in a large number of false positives. A significant number of false positives will create more workload and make it more complicated to perform reverse analysis. Hence, monitoring a ProbeAccess event through the CPU emulator is of significant importance. We therefore propose the use of CPU emulator to detect UNPROBE vulnerabilities.

Take Figure 4 as an example, at the event time of "$n+3$", the kernel code triggers a MemAccess event by accessing user memory. If there is no ProbeAccess/ProbeRead/ProbeWrite event to check the user address beforehand, or no AllocVirtualMemory/GetPebTeb event to imply the legitimacy of the address, an UNPROBE vulnerability may exist in the kernel code. In contrast, if there is a ProbeAccess/ProbeRead/ProbeWrite event or GetPebTeb/AllocVirtualMemory event to suggest that the user address is legal, and the event is trig-

gered in the same system call as the MemAccess event, the code is safe.

To detect an UNPROBE vulnerability, the fuzzer invokes the test system calls and tries to discover as many branches as possible by adjusting their parameters. Furthermore, the log analyzer looks for MemAccess events in which the user addresses have not been verified by a ProbeAccess/ProbeRead/ProbeWrite or GetPebTeb/AllocVirtualMemory event during a system-call execution.

4.2.2 Detecting TOCTTOU Vulnerabilities

There are two key factors in a TOCTTOU vulnerability. One is that the parameter passed from a user-mode program should be a pointer. The other is that the system-call handler fetches the parameter from user memory more than once. Thus, the user-mode code has a chance to change the parameter referenced by the pointer.

Take Figure 4 again for instance, if a piece of kernel code accesses the same user memory at the time of "$n+2$" and "$n+3$," there may be a TOCTTOU vulnerability in the kernel code. To discover this type of vulnerability, the key point is to look for the user memory that has been accessed more than once in the log files. The event time could help to improve the accuracy. If there are two MemAccess events that fetch from the same user memory, we can judge whether they are triggered in the same system call by comparing the two events' times with the Syscall/Trap2b/Trap2e event time and the RetUser event time. Only when they are in the same system-call execution, may a TOCTTOU vulnerability exist.

The fuzzer needs to invoke the test system calls, and it should discover as many branches as possible by adjusting parameters. At the same time, interface detection records the dynamic characteristics via the middleware. Then, the log analyzer is used to look for the user memory addresses that have been accessed more than once during a system-call execution.

4.3 Detecting Vulnerabilities via Memory Footprints

Memory-footprint-based detection is used to detect illegal use of kernel memory by tracing the behavior of memory allocation, release, and access. In this paper, we will focus on two aspects of illegal memory use: accessing beyond the bounds of allocated heaps and referencing to freed memory. These can lead to *OOB* and *UAF* vulnerabilities.

To capture the dynamic characteristics of vulnerabilities, we need to monitor the allocated, unallocated, and freed memory. Accessing allocated memory is allowed, but using unallocated or freed memory is illegal. Digtool monitors the kernel memory through the virtual page monitor. Illegal memory access will be captured by its page-fault handler in the hypervisor. Then, it records the memory access error or submits it to a kernel debug tool like `Windbg` [8]. Thus, the exact context of kernel execution can be provided for the vulnerability detection.

In order to obtain more relevant data and reduce performance overhead, the monitored memory pages can be restricted. The middleware helps to limit the scope of monitored pages, and passes the scope to the memory detection by invoking our exported service interfaces of Digtool. For instance, when detecting UAF vulnerabilities, we are only concerned with freed memory, so we need to limit the scope to freed pages. Furthermore, to put more emphasis on the kernel code under test, Digtool can also specify target modules to define a scope of kernel code. Only the instructions in the target modules that cause illegal memory access are recorded. Thus, we can concentrate on the target code tested by the fuzzer.

For tracing the allocated and freed memory, Digtool hooks memory functions such as `ExAllocatePoolWithTag` and `ExFreePoolWithTag`. These functions are used to allocate or free kernel memory in the guest OS. Thus, we can determine which memory region is allocated and which is freed. As the size of freed memory cannot be directly obtained through the arguments of the free functions, Digtool records the memory address and the memory size via the parameters of allocation functions. Thus, when a free function is called, the memory size can be obtained by searching for the record.

Memory allocations before Digtool is loaded cannot be captured. Therefore, Digtool should be loaded as early as possible to achieve more precise detection. It is feasible to load Digtool during boot time by setting the registry. Thus, there are only a few modules loaded before Digtool and the unmonitored memory allocations are few, which largely limits the attack surfaces. To summarize, the memory allocations before loading Digtool have a negligible impact on precision. Built atop virtualization technology, our memory-footprint-based approach can be applied to various kernels and device drivers without any compile-time requirements.

4.3.1 Detecting UAF Vulnerabilities

UAF results from reusing the freed memory. To detect it, memory detection needs to trace the freed memory pages until they are allocated again. Any access to the freed memory will be marked as a UAF vulnerability.

In order to trace freed memory, memory functions such as `ExAllocatePoolWithTag`, `ExFreePoolWithTag`, `RtlAllocateHeap`, and `RtlFreeHeap` (as discussed above, hooking mem-

ory allocation functions is done to record the size of freed memory) need to be hooked. Note that the Windows OS implements some wrapper functions for these. For instance, both `ExAllocatePool` and `ExAllocatePoolEx` are the wrapper functions for `ExAllocatePoolWithTag`. To avoid multiple monitoring and repetitive records, Digtool only hooks underlying memory functions such as `ExAllocatePoolWithTag` rather than wrapper functions. Inappropriate use of lookaside lists will also cause UAF vulnerabilities. Digtool hooks the corresponding functions, including `InterlockedPushEntrySList` and `InterlockedPopEntrySList`, to monitor the freed memory blocks in the lookaside lists.

Any instruction operating on the freed memory (or blocks) is regarded as the "*use*" instruction of a UAF vulnerability. It is obtained through the virtual page monitor. The "*free*" instruction of a UAF vulnerability is obtained by recording the free function when it is invoked, and its call-stack information is recorded through a *backtrace* of the stack to facilitate analysis.

A UAF vulnerability may be missed in some scenarios. Considering such situations, there is a memory block `A` referenced by pointer `P`. After freeing block `A`, another program allocates a memory block `B` that covers the entire memory of block `A`. Then, the first program tries to manipulate block `A` through the pointer `P`. Obviously, there is a UAF vulnerability in the first program. However, as the memory region of block `A` is allocated again, it is difficult to detect the vulnerability. This is the reason that Driver Verifier may miss a UAF vulnerability. In order to solve this problem, Digtool delays the release of the freed memory to extend the detection time window. The freed memory will be released until it reaches a certain size.

4.3.2 Detecting OOB Vulnerabilities

An OOB vulnerability can be caused by accessing memory that is beyond the bounds of allocated heaps. To detect it, the monitored memory space should be limited to the unallocated memory areas. Any access to the unallocated memory areas will prompt an OOB vulnerability.

Digtool calibrates the unallocated memory areas through the help of the middleware. In general, except for the memory areas occupied by kernel modules and stacks, the rest of the memory pools are defined as initial unallocated memory areas. As the kernel memory state keeps changing, memory functions that allocate or free memory need to be hooked. Thus, it can adjust the unallocated memory areas dynamically. During the detection process, Digtool needs to search the records of allocated or unallocated memory areas. An AVL tree (i.e., a self-balancing binary search tree) is employed to improve the performance of the memory search. It adds a node when a memory area is allocated, and deletes the node if the memory is freed. Thus, when a monitored page (not a memory area) is accessed (note that the monitoring granularity of memory virtualization is a page, but the size of a memory area may be less than a page; the monitored pages are recorded via the BitMap, while the monitored memory areas are stored in the AVL tree.), Digtool searches the AVL tree for the accessed memory area. If no related node is found, an OOB vulnerability may exist.

Note that, as unallocated memory contains freed memory in the detection, an "OOB" may be caused by accessing a freed memory area. Some reverse-engineering effort is needed to further distinguish between OOB and UAF vulnerabilities.

An OOB vulnerability may be missed in some scenarios. Considering such situations, two memory blocks `A` and `B` are allocated and they are adjacent. A brittle program tries to access block `A` with a pointer and an offset, but the offset is so large that the accessed address locates in block `B`. This is an obvious OOB vulnerability. However, block `B` is also in the AVL tree, so it is difficult to detect this error. To solve this problem, Digtool will allocate an extra memory area with `M` bytes when a hooked memory allocation function is invoked. As a result, the total size of block `A` is `sizeof(A)+M`, and the start address of block `B` will be backward for `M` bytes. However, the size of block `A` recorded in the AVL tree is still defined as `sizeof(A)` bytes. As a consequence, the extra memory area with `M` bytes is not in the AVL tree. Thus, instead of block `B`, the brittle program will access the extra memory area, and an OOB vulnerability will be then captured by Digtool.

5 Evaluation

5.1 Effectiveness

We checked the detection capability of Digtool by testing the programs of different products, including the Windows OS and some anti-virus software (all of the products were the latest version at the time of the experiments). The experimental environments included Windows 7 and Windows 10. (Digtool can support Windows XP/Vista/7/8/10, etc.) We chose some *zero-day* vulnerabilities that had been responded to and fixed by the responsible vendors as examples to illustrate the experimental results. *All of the vulnerabilities discussed below were first discovered by Digtool* (all have been reported to the corresponding vendors, among which Microsoft, Avast, and Dr. Web have confirmed and fixed their vulnerabilities).

Table 1: List of UNPROBE vulnerabilities.

Software products	Unsafe system calls
Avast Free Antivirus 11.2.2262	NtAllocateVirtualMemory NtCreateSection
Dr. Web 11.0	NONE
AhnLab 8.0	NtQueryValueKey NtCreateKey NtDeleteValueKey NtLoadKey NtOpenKey NtSetValueKey NtUnloadKey
Norman Security Suite 11.0.0	NtCreateMutant NtCreateEvent NtCreateFile NtCreateSemaphore
Spyware Detector 2.0.0.3	NtCreateFile NtCreateKey NtDeleteFile NtDeleteValueKey NtOpenFile NtOpenKey NtOpenSection NtSetInformationFile NtSetValueKey NtWriteVirtualMemory

5.1.1 Detecting Vulnerabilities via Interface

We chose five anti-virus software products as test targets since they intercept many system calls that could be invoked by user-mode applications. The test was mainly carried out on Avast for its strength of complexity. The other four anti-virus software products included Dr. Web, Ahnlab, Norman, and Spyware Detector. We used some zero-day vulnerabilities discovered through Digtool to verify its ability to detect UNPROBE and TOCTTOU vulnerabilities. The middleware recorded the behavior characteristics into log files to help locate vulnerabilities.

Detecting UNPROBE. Taking a vulnerability in Avast 11.2.2262 as an example, through the log analyzer, the following data were obtained from the Digtool's log file for Avast 11.2.2262:

```
NtAllocateVirtualMemory:
Eip: 89993f3d, Address: 0023f304, rw: R
Eip: 84082ed9, Address: 0023f304, PROBE!
KiFastSystemCallRet
```

`aswSP.sys`, the Avast driver program, used the instruction at the address `0x89993f3d` to fetch the value from the user address (i.e., `0x23f304`) without checking. The subsequent checking instruction at the address `0x84082ed9` belonged to the `NtAllocateVirtualMemory` function. Therefore, there was a typical UNPROBE vulnerability in `aswSP.sys`.

Using Digtool, 23 similar vulnerabilities were found in the five anti-virus software programs tested. The results are shown in Table 1. For security reasons, we only give the system calls for which vulnerabilities exist.

When the log analyzer points out a potential UNPROBE vulnerability, and the tested driver only uses the `ProbeForRead` and `ProbeForWrite` functions to check a user pointer (this is a common scenario in third-party drivers), no human effort is needed for further confirmation as the detection is precise due to the facts that the start address and length information of the input buffer can be obtained through the corresponding kernel function. If the driver uses direct comparison to check a user pointer, Digtool may produce false positives or false negatives. This results from a lack of accurate address ranges in the ProbeAccess event as we cannot obtain the "size" of the input buffer. We must assume the length for the input user-mode buffer. If the assumed length is larger then the real one, false negatives may be produced. Otherwise, false positives may be generated.

In the case of ProbeAccess, Digtool only helps point out a potential vulnerability. Human effort is still needed to obtain the exact length of the input user-mode buffer through reverse analysis so that we can determine whether the instruction (given by the log analyzer) could really cause an UNPROBE vulnerability.

Detecting TOCTTOU. Taking a vulnerability in Dr. Web 11.0 as an example, through the `log analyzer` the following dynamic characteristics were distilled from Digtool's log file for Dr. Web 11.0:

```
NtCreateSection:
Count:3 ==============
Eip: 83f0907f Address:3b963c Sequence:398 rw: R
Eip: 89370d54 Address:3b963c Sequence:399 rw: R
Eip: 89370d7b Address:3b963c Sequence:401 rw: R
KiFastSystemCallRet
```

The user address `0x3b963c` was accessed by the kernel instructions more than once, so there may be a TOCTTOU vulnerability. `dwprot.sys`, the Dr. Web driver program, used the instruction at the address `0x89370d54` to fetch the value from the user address (i.e., `0x3b963c`), and then it invoked the `ProbeForRead` function to check it. At the address `0x89370d7b`, the `dwprot.sys` fetched the value again to use it. Therefore, there was a typical TOCTTOU vulnerability in `dwprot.sys`.

With the help of Digtool, 18 kernel-level TOCTTOU vulnerabilities were found in the five anti-virus software programs tested. The results are shown in Table 2. For security reasons, we only give the system calls for which vulnerabilities exist.

Digtool may produce false positives that originate from the fact that it detects TOCTTOU vulnerabilities through double-fetch. Further manual analysis is needed to confirm that double-fetch is a TOCTTOU vulnerability.

5.1.2 Detecting Vulnerabilities via Memory Footprints

We chose 32-bit Windows 10 as a test target. Some zero-day vulnerabilities discovered by Digtool were selected

Table 2: List of TOCTTOU vulnerabilities.

Software products	Unsafe system calls
Avast Free Antivirus 11.2.2262	NtUserOpenDesktop NtQueryObject NtUserBuildNameList NtOpenSection NtCreateEvent NtCreateEventPair NtCreateIoCompletion NtCreateMutant NtCreateSection NtCreateSemaphore NtCreateTimer NtOpenEvent NtOpenEventPair NtOpenIoCompletion NtOpenMutant NtOpenSemaphore NtOpenTimer
Dr. Web 11.0	NtCreateSection
AhnLab 8.0	NONE
Norman Security Suite 11.0.0	NONE
Spyware Detector 2.0.0.3	NONE

to verify its effectiveness in detecting UAF and OOB vulnerabilities. Instead of logging, the middleware was set to interrupt the guest OS and connect to `Windbg` when a program error was captured. Thus, an exact context can be provided for analysis.

Detecting UAF. The following content is shown by `Windbg` when the UAF vulnerability (`MS16-123/CVE-2016-7211` [3]) is captured in `win32kfull.sys`; this vulnerability was first discovered through Digtool:

```
Single step exception - code 80000004
win32k!_ScrollDC+0x21:
96b50f3e  83ff01 cmp edi,1
```

The "Single-step exception" is triggered by Digtool. As it is a trap event, the instruction that triggers the exception has already been finished, and the guest OS is interrupted at the address of the next instruction to be executed. The instruction just before `0x96b50f3e` is the exact instruction that tries to access a freed memory area and causes the UAF vulnerability. We can obtain it by `Windbg` as follows and its address is `0x96b50f3b`. The `esi` register (at the address of `0x96b50f3b`) stores the address of the freed heap:

```
96b50f3b 8b7e68 mov edi,dword ptr [esi+68h]
96b50f3e 83ff01 cmp edi,1//win32k!_ScrollDC+0x21
```

Detecting OOB. Vulnerabilities including `MS16-090/CVE-2016-3252` [2], `MS16-034/CVE-2016-0096` [1], and `MS16-151/CVE-2016-7260` [4] were first discovered by Digtool. Taking `MS16-090/CVE-2016-3252` as an example to illustrate the detection result, the following content was shown when the vulnerability was captured in `win32kbase`:

```
Single step exception - code 80000004
win32kbase!RGNMEMOBJ::bFastFill+0x385:
93e34bf9 895304      mov dword ptr [ebx+4],edx
```

This is similar to the content of the above UAF, and `0x93e34bf9` is the address of the next instruction to be executed. The instruction just before `0x93e34bf9` is the exact instruction that tries to access an unallocated memory area and causes the OOB vulnerability.

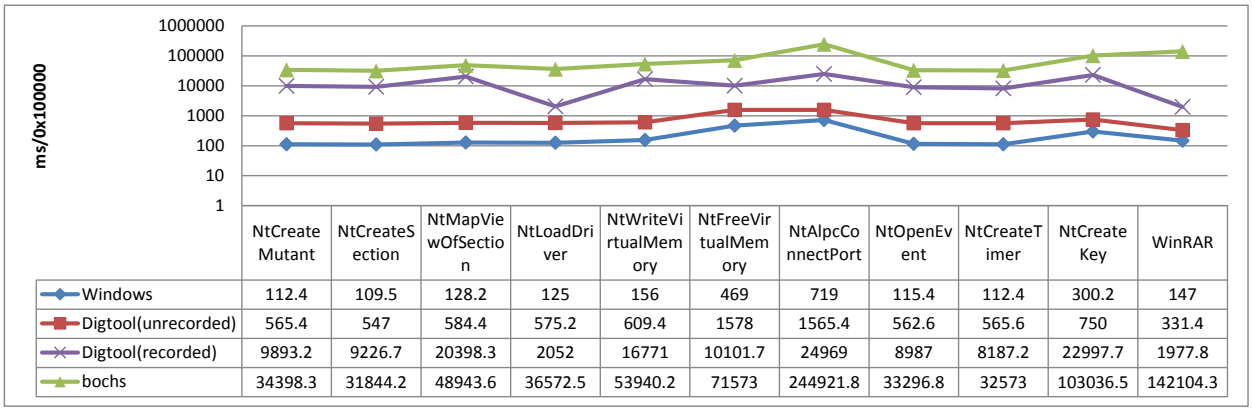
Note that there is no false positive in the UAF/OOB detection, and no human effort is needed for locating or confirming the vulnerability. Whenever an exception is captured, it is always a vulnerability.

5.2 Efficiency

Owing to the fact that Bochspwn [24], which is based on the bochs emulator [25], only detects TOCTTOU vulnerabilities among the four types of vulnerabilities by now, we tested Digtool's performance cost in detecting TOCTTOU vulnerabilities, and compared its performance with that of the bochs emulator in the same environment (i.e., the same hardware platform, OS version, parameters of system calls, and arguments of the test program). We chose ten common system calls that are the most widely used and hooked by anti-virus software to test the efficiency. In order to obtain a more comprehensive result, we also chose a frequently used program, WinRAR 5.40 [7], for an efficiency test. The performance cost is shown in Figure 6 (the result may be affected by some factors, such as the parameters of system calls and the WinRAR input file).

The performance cost of Digtool is divided into two categories: "unrecorded" and "recorded." "Unrecorded" means that the system calls are not included in the configuration file, and thus no page is monitored and no log is recorded for them. However, the other modules in interface detection are activated. This class of performance cost can provide a comprehensive comparison with the bochs emulator since the bochs emulator records nothing. In addition, it also reflects the state of the entire system since most of system calls and threads are unmonitored when detecting TOCTTOU. "Recorded" indicates that the system calls are put into the configuration file and their behaviors are recorded. It describes the performance cost of the related system calls in the specified monitored thread, but has nothing to do with the performance of the other system calls and threads. "Windows" denotes the performance of a clean OS without any tools, and "bochs" represents the performance cost of the OS running into bochs emulator.

In the case of "unrecorded," the result of system calls showed that Digtool is from 2.18 to 5.03 times slower than "Windows," but 45.4 to 156.5 times faster than "bochs." From the WinRAR result, Digtool is 2.25



	NtCreateMutant	NtCreateSection	NtMapViewOfSection	NtLoadDriver	NtWriteVirtualMemory	NtFreeVirtualMemory	NtAlpcConnectPort	NtOpenEvent	NtCreateTimer	NtCreateKey	WinRAR
Windows	112.4	109.5	128.2	125	156	469	719	115.4	112.4	300.2	147
Digtool(unrecorded)	565.4	547	584.4	575.2	609.4	1578	1565.4	562.6	565.6	750	331.4
Digtool(recorded)	9893.2	9226.7	20398.3	2052	16771	10101.7	24969	8987	8187.2	22997.7	1977.8
bochs	34398.3	31844.2	48943.6	36572.5	53940.2	71573	244921.8	33296.8	32573	103036.5	142104.3

Figure 6: Performance overhead.

times slower than "Windows," but 428.8 times faster than "bochs." In the case of "recorded," most of the monitored system calls are from 70 to 90 times (which depends on the arguments and system calls) slower than "Windows," but still much faster than "bochs." From the WinRAR result (all of the system calls in the `NT` kernel are recorded), the "recorded" case is 13.45 times slower than "Windows." This finding offers another perspective on the average performance cost of an application under the situation of monitoring all system calls. In this extreme case, Digtool is still 71.8 times faster than the bochs emulator. Thus, Digtool achieves an acceptable level of performance.

5.3 Comparison and Analysis

Next, we illustrate Digtool's advantages by comparison with Driver Verifier [28], which is a notable tool for checking Windows kernels.

Crash resilient. Digtool is able to capture dynamic characteristics of potential vulnerabilities without needing a "Blue Screen of Death" (BSOD). As the analysis process only requires the recorded data containing accessed memory address, event type, and event time, there is no need for triggering a BSOD to locate a program error. The fuzzer only needs to discover as many code branches as possible, and it does not have to crash the OS. During this process, Digtool will record all dynamic characteristics. Without a BSOD, it keeps recording, which will help find more vulnerabilities.

However, it is inevitable that Driver Verifier will cause a BSOD to locate and analyze a vulnerability. It does not stop crashing the OS at the address of the same program error until the error is fixed. This will make it difficult to test other vulnerabilities. For example, when we test Avast with Driver Verifier, the cause of a BSOD is always the same:

```
Arg1:f6, Referencing user handle as KernelMode.
Arg2:0c, Handle value being referenced.
```

The BSOD results from using a user-mode handle under the `KernelMode` flag. If the problem is not solved, Driver Verifier cannot further test Avast.

Interrupting the OS with an exact context. Through the middleware, Digtool can be set to interrupt the guest OS when a program error happens. Thus, it can provide an exact context for the vulnerability by connecting to a debug tool.

Driver Verifier has to crash the OS to locate and analyze a program error. However, the context has been changed since the OS is not stopped at the moment the program error occurs (usually, the OS will keep running for a moment to trigger the program error). Much more human effort is needed to locate the error.

Taking `MS16-090/CVE-2016-3252` [2] as an example, Digtool exactly locates the instruction (just before `0x93e34bf9`) that causes the vulnerability:

```
win32kbase!RGNMEMOBJ::bFastFill+0x385:
93e34bf9 895304      mov dword ptr[ebx+4],edx
```

However, from Driver Verifier, the captured context is as follows:

```
BAD_POOL_HEADER (19)
FOLLOWUP_IP:
win32kfull!NSInstrumentation::PlatformFree+10
a0efaade 5d      pop ebp
```

Driver Verifier only points out a "bad pool" (OOB) error, but does not provide an exact context for the vulnerability. Much more reverse-engineering effort is required to locate the vulnerability from the above information.

Capturing more vulnerabilities. Digtool can effectively detect UNPROBE and TOCTTOU vulnerabilities. However, as no similar detection rule is designed, Driver Verifier cannot be used to detect them. Moreover, Driver

Verifier may sometimes miss a UAF or OOB vulnerability because the vulnerability may happen to access a valid memory page, and does not cause a BSOD. Thus, Driver Verifier cannot find it.

The above UAF vulnerability (`MS16-123/CVE-2016-7211`) discovered by Digtool is an example. It accesses a freed memory block that is almost immediately reallocated again under normal circumstances. As a consequence, the physical page of the freed memory block is valid, and it does not violate the rule of Driver Verifier, no BSOD is caused, and no bug is found. However, the vulnerability can be captured by Digtool due to the fact that it delays the release of freed memory. Thus, Digtool is more powerful in this regard.

To summarize, Digtool discovers *45 zero-day* kernel vulnerabilities, and effectively detects the four types of program errors: *UNPROBE, TOCTTOU, UAF*, and *OOB*. In terms of efficiency, it achieves significantly better performance than Bochspwn. Compared to Driver Verifier, it can *capture multiple vulnerabilitie with an exact execution context*. As such, Digtool can be considered a complement to Driver Verifier.

6 Discussion

Digtool has a number of limitations. First, the performance cost could be optimized. Although it is much faster than an emulator, the performance overhead is still costly in the monitored threads. The performance cost mainly comes from the frequent switches between the hypervisor and guest OS. How to reduce the switches and the performance cost could be a research topic.

Second, the supported platforms could be extended. Digtool currently only supports the Windows OS. Via virtualization technology, the hypervisor runs outside of the guest OS, which tends to be more portable and has the potential of supporting other OSes. However, the middleware in the kernel space is platform-specific. The main work of supporting various platforms (e.g., MacOS) is adapting the middleware.

Third, there is still room for extension in the detection algorithms. Currently, Digtool is able to detect UNPROBE, TOCTTOU, UAF, and OOB vulnerabilities. As it can almost monitor any memory page, it could be used to detect some other types of vulnerabilities, such as race conditions, by extending the detection algorithms.

7 Related Work

7.1 Static Analysis

Static analysis is to detect potential vulnerabilities from programming language literature. Unlike other detection methods, it does not depend on executable binary files. Wagner *et al*. [39] proposed an automated detection method of finding program bugs in C code that can discover potential buffer overrun vulnerabilities by analyzing source code. Grosso *et al*. [19] also presented a method of detecting buffer overflows for C code that does not need human intervention to define and tune genetic algorithm weights, and therefore it becomes completely automated.

Static analysis achieves a high rate of code coverage, but its precision may be insufficient when dealing with difficult language constructs and concepts. In addition, it cannot detect program bugs without source code.

7.2 Source Instrumentation

Source instrumentation is also called compile-time instrumentation; it inserts detection code at compile-time to detect program bugs. CCured [30] is used to detect unsafe pointers for C programs. It combines instrumentation with static analysis to eliminate redundant checks. AddressSanitizer [36] creates poisoned redzones around heaps, stacks, and global objects to detect overflows and underflows. Compared to other methods, it can detect errors not only in heaps, but also in stacks and global variables.

Source instrumentation has higher precision, but its code coverage may be less comprehensive than static analysis. In addition, it has the same limitation as static analysis; that is, it cannot detect program bugs without source code.

7.3 Binary Instrumentation

Binary instrumentation inserts detection code into executable binary files and detects program bugs at runtime. Purify [22] is an older tool for checking program bugs based on binary instrumentation that can detect memory leaks and memory access errors. Valgrind [31] is a dynamic binary instrumentation framework designed to build heavyweight binary analysis tools like Memcheck [37]. Dr. Memory [14] is a memory-checking tool that operates on applications under both Windows and Linux environments.

These tools do not rely on source code, and exhibit an ability to effectively detect program errors. However, many of them only detect bugs for applications in user mode and cannot operate on programs in kernel mode, especially on the Windows kernel. Some Qemu-based tools support the instrumentation of Windows OS kernel, but these tools cannot be used to detect vulnerabilities in a physical machine and their average performance overhead is quite high.

7.4 Specialized Memory Allocator

Another class of vulnerability identification tool uses a specialized memory allocator and does not change the rest of the executable binary files. It analyzes the legality of memory access by replacing or patching memory functions.

Some tools make use of the page-protection mechanism of processors. Each allocated region is placed into a dedicated page (or a set of pages). One extra page at the right (or/and the left) is allocated and marked as inaccessible. A page fault will be reported as an OOB error when instructions access the inaccessible page. Duma [9] and GuardMalloc [26] are in this category.

Some other tools add redzones around the allocated memory. In addition to the redzones, they also populate the newly allocated memory or freed memory with special "magic" values. If a magic value is read, the program may have accessed an out-of-bounds or uninitialized memory. If a magic value in a redzone is overwritten, it will be detected later, when the redzone is examined for freed memory. Therefore, there is no immediate detection of the memory access error. Tools in this category include DieHarder [33] and Dmalloc [40].

These tools do not depend on source code either and are well suited for discovering memory errors, but they share the limitation encountered in other tools, namely that many of them cannot operate on the Windows kernel. Moreover, it is difficult for them to check for UNPROBE or TOCTTOU vulnerabilities.

7.5 Kernel-Level Analysis Tools

There are only a few vulnerability identification tools for programs in kernel mode, and most of them are aimed at Linux. Kmemcheck [32] and Kmemleak [6] are memory-checking tools for the Linux kernel. Kmemcheck monitors the legality of memory access by tracing read and write operations. Kmemleak is used to detect memory leaks by checking allocated memory blocks and their pointers. Both tools can help discover memory errors in the Linux kernel. However, all of the similar tools need to expand the source code of Linux or insert detection code at compile-time, and thus it is difficult to port them to a closed-source OS like Windows.

Driver Verifier [28] is the major tool for detecting bugs in the Windows kernel. It can find program bugs that are difficult to discover during regular testing. These bugs include illegal function calls, memory corruption, bad I/O packets, deadlocks, and so on. Driver Verifier is an integrated system for detecting illegal actions that might corrupt the OS, but not a dedicated tool for detecting vulnerabilities (see Section 5.3 for a discussion of Driver Verifier's ability to detect vulnerabilities). As part of the kernel, in fact, Driver Verifier also relies on the source code of the OS.

Although the above tools can be applied to detect kernel vulnerabilities, they are too tightly coupled with implementation details and the source code of OSes, so they cannot work when no source code is available. Moreover, it is difficult to port them to another type of OS.

7.6 Virtualization/Emulator-Based Methods

Virtualization/emulator-based vulnerability identification tools detect potential vulnerabilities by tracing function calls and monitoring memory access. Through virtualization or emulator technology, they can overcome most OS differences and easily support various OSes.

Among the more common virtualization-based tools and methods are the following. VirtualVAE [18] is a vulnerability analysis environment that is based on QEMU [11]. In [18], it is claimed that it can detect bugs for programs in both kernel mode and user mode. PHUKO [38], based on Xen [10], detects buffer overflow attack, and it checks return addresses for dangerous functions to determine vulnerabilities. These virtualization-based methods only focus on a single type of program error. They are not built as a framework for detecting various vulnerabilities. Moreover, the implementation details for some of them are not exhaustive, and the detection effects have not been illustrated through detection of vulnerabilities in the real world. Their performance may be influenced by a full-stack virtualization framework.

Bochspwn [24] is a notable emulator-based vulnerability identification tool. Dozens of TOCTTOU vulnerabilities have been found in the Windows kernel using Bochspwn. However, its scope of application is limited by the bochs emulator.

8 Conclusions

In this paper, a virtualization-based vulnerability identification framework called Digtool is proposed. It can detect different types of kernel vulnerabilities including *UNPROBE, TOCTTOU, UAF*, and *OOB* in the Windows OS. It successfully captures various dynamic behaviors of kernel execution such as kernel object allocation, kernel memory access, thread scheduling, and function invoking. With these behaviors, Digtool has identified 45 *zero-day* vulnerabilities among both kernel code and device drivers. It can help effectively improve the security of kernel code in the Windows OS.

Acknowledgement

We are grateful to the anonymous reviewers for their insightful comments, which have significantly improved our paper. We also would like to thank Ella Yu and Yao Wang for their invaluable feedback on earlier drafts of this paper.

References

[1] Cve-2016-0096. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0096.

[2] Cve-2016-3252. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3252.

[3] Cve-2016-7211. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7211.

[4] Cve-2016-7260. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7260.

[5] drk. https://github.com/DynamoRIO/drk.

[6] Kernel memory leak detector. http://www.mjmwired.net/kernel/Documentation/kmemleak.txt.

[7] Rarlab. http://www.rarlab.com/.

[8] Windbg. http://www.windbg.org/.

[9] Hayati Aygün and M Eddington. Duma-detect unintended memory access, 2013.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[11] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[12] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

[13] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE, 2003.

[14] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011.

[15] Marc Brünink, Martin Süßkraut, and Christof Fetzer. Boundless memory allocations for memory safety and high availability. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 13–24. IEEE, 2011.

[16] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 725–741. IEEE, 2015.

[17] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[18] Wang Chunlei, Wen Yan, and Dai Yiqi. A software vulnerability analysis environment based on virtualization technology. In *Wireless Communications, Networking and Information Security (WCNIS), 2010 IEEE International Conference on*, pages 620–624. IEEE, 2010.

[19] Concettina Del Grosso, Giuliano Antoniol, Ettore Merlo, and Philippe Galinier. Detecting buffer overflow via automatic test input data generation. *Computers & Operations Research*, 35(10):3125–3143, 2008.

[20] Yangchun Fu and Zhiqiang Lin. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *Proceedings of the Ninth Annual International Conference on Virtual Execution Environments*, Houston, TX, March 2013.

[21] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144. ACM, 2012.

[22] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference*. Citeseer, 1991.

[23] Intel. Intel 64 and ia-32 architectures software developer's manuals. 2016.

[24] Mateusz Jurczyk, Gynvael Coldwind, et al. Identifying and exploiting windows kernel race conditions via memory access patterns. 2013.

[25] Kevin P Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996(29es):7, 1996.

[26] Mac OS X Developer Library. Memory usage performance guidelines: Enabling the malloc debugging features. `http://developer.apple.com/library/mac/#documentation/darwin/reference/manpages/man3/libgmalloc.3.html`.

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[28] Daniel Mihai, Gerald Maffeo, and Silviu Calinoiu. Driver verifier, February 23 2006. US Patent App. 11/360,153.

[29] Amatul Mohosina and Mohammad Zulkernine. Deserve: a framework for detecting program security vulnerability exploitations. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 98–107. IEEE, 2012.

[30] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128–139. ACM, 2002.

[31] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[32] Vegard Nossum. Getting started with kmemcheck, 2012. `http://www.mjmwired.net/kernel/Documentation/kmemcheck.txt`.

[33] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[34] Bryan D Payne. Libvmi. Technical report, Sandia National Laboratories, 2011.

[35] Vladimir V Rubanov and Eugene A Shatokhin. Runtime verification of linux kernel modules based on call interception. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 180–189. IEEE, 2011.

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[37] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[38] Donghai Tian, Xi Xiong, Changzhen Hu, and Peng Liu. Defeating buffer overflow attacks via virtualization. *Computers & Electrical Engineering*, 40(6):1940–1950, 2014.

[39] David Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.

[40] Gray Watson. Dmalloc–debug malloc library, 2004.

[41] Felix Wilhelm. Tracing privileged memory accesses to discover software vulnerabilities. 2015.

[42] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework. In *Proceedings of the 11th Annual International Conference on Virtual Execution Environments*, Istanbul, Turkey, March 2015.

[43] Junyuan Zeng and Zhiqiang Lin. Towards automatic inference of kernel object semantics from binary code. In *International Symposium on Recent Advances in Intrusion Detection*, pages 538–561. Springer, 2015.