Nioh-PT: Virtual I/O Filtering for Agile Protection against Vulnerability Windows

Mana Senuki
Keio University
senuki@sslab.ics.keio.ac.jp

Kenta Ishiguro
Hosei University
kenta.ishiguro.66@hosei.ac.jp

Kenji Kono
Keio University
kono@sslab.ics.keio.ac.jp

ABSTRACT

Hypervisor vulnerabilities cause severe security issues in multi-tenant cloud environments because hypervisors guarantee isolation among virtual machines (VMs). Unfortunately, hypervisor vulnerabilities are continuously reported, and device emulation in hypervisors is one of the hotbeds because of its complexity. Although applying patches to fix the vulnerabilities is a common way to protect hypervisors, it takes time to develop the patches because the internal knowledge on hypervisors is mandatory. The hypervisors are exposed to the threat of the vulnerabilities exploitation until the patches are released.

This paper proposes *Nioh-PT*, a framework for filtering illegal I/O requests, which reduces the vulnerability windows of the device emulation. The key insight of Nioh-PT is that malicious I/O requests contain illegal I/O sequences, a series of I/O requests that are not issued during normal I/O operations. Nioh-PT filters out those illegal I/O sequences and protects device emulators against the exploitation. The filtering rules, which define illegal I/O sequences for virtual device exploits, can be specified without the knowledge on the internal implementation of hypervisors and virtual devices, because Nioh-PT is decoupled from hypervisors and the device emulators. We develop 11 filtering rules against four real-world vulnerabilities in device emulation, including CVE-2015-3456 (VENOM) and CVE-2016-7909. We demonstrate that Nioh-PT with these filtering rules protects against the virtual device exploits and introduces negligible overhead by up to 8% for filesystem and storage benchmarks.

CCS CONCEPTS

• **Security and privacy** → **Virtualization and security**;

KEYWORDS

security, virtualization, virtual device, cloud computing

ACM Reference Format:
Mana Senuki, Kenta Ishiguro, and Kenji Kono. 2023. Nioh-PT: Virtual I/O Filtering for Agile Protection against Vulnerability Windows. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-March 31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3555776.3577687

SAC '23, March 27- March 31, 2023, Tallinn, Estonia

ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3577687

1 INTRODUCTION

In multi-tenant cloud environments, hypervisors play a crucial role. Hypervisors virtualize hardware resources such as CPU, memory, and peripheral devices for high utilization and flexibility. Consequently, hypervisor vulnerabilities cause severe security issues because hypervisors are in charge of isolating guest virtual machines. However, hypervisor vulnerabilities are continuously reported, and device emulation to multiplex peripheral devices is one of the hotbeds because of its complexity [14, 30, 33, 34].

Although vulnerabilities in device emulation vary in devices and risk, attacking through illegal I/O requests is one of the typical methods to exploit. For example, VENOM [21] and CVE-2016-7909 [19] are vulnerabilities in the floppy disk controller (FDC) emulator and the AMD PC-Net PCI II Ethernet Controller emulator. VENOM allows guest users to cause a denial of service (DoS) or possibly execute arbitrary code in the host hypervisor. CVE-2016-7909 allows guest operating system (OS) administrators to cause a DoS. The attack vector to exploit these vulnerabilities is common. The attackers who control VMs perform illegal I/O requests that are not issued for devices during normal operations.

Fixing vulnerabilities in hypervisors takes time because detailed analysis and careful regression testing are necessary. A *vulnerability window* of a hypervisor is defined as the period from its identification to the installation of a correction or patch to the hypervisor. Long vulnerability windows of hypervisors leave much time for attackers to exploit the vulnerabilities. For example, according to Ngoc et al. [25], 60% of the 24 KVM vulnerabilities took more than 60 days to fix. Alternative protection rather than patching is necessary to reduce the time of exposing hypervisors to vulnerabilities.

This paper proposes a virtual I/O filtering framework, named *Nioh-PT*, that defends against exploitation of vulnerabilities in device emulators during vulnerability windows. Nioh-PT consists of a *slim monitoring layer* and an *I/O request filtering engine*. First, Nioh-PT introduces the slim monitoring layer between the hypervisor and guest VMs. This layer monitors I/O requests from guest VMs and passes their information to the I/O filtering engine. Next, the I/O filtering engine filters out illegal I/O requests according to the filtering rules specified by device emulator developers. To protect against a given vulnerability, system administrators introduce the filtering rule developed with APIs provided by the I/O filtering engine, by specifying the illegal I/O sequence which is a series of I/O requests to exploit the vulnerability. Since the illegal I/O sequences are not usually issued during normal operations, the I/O filtering engine can reject such I/O requests without interfering with normal operations. This protection model allows the system administrators to deal with the vulnerabilities without detailed internal knowledge and modification of device emulators if they know the illegal I/O

sequences. Therefore, Nioh-PT works as a temporary countermeasure against vulnerabilities in device emulation during vulnerability windows.

We implement a prototype of Nioh-PT for KVM [16] version 4.15.0-153-generic and QEMU [8] version 2.9.50. We add 60 lines of code (LoC) for the slim monitoring layer to QEMU and build the I/O filtering engine with 370 LoC in Rust from scratch. By using Rust, a type- and memory-safe language [31], Nioh-PT reduces risks of memory leak and buffer overflow caused by adding the I/O filtering engine. We also implement filtering rules against four real-world vulnerabilities: VENOM [21], CVE-2015-5279 [22], CVE-2016-7909 [19], and CVE-2020-13361 [20]. Our performance experiment with Filebench [1], the filesystem and storage benchmark, shows that Nioh-PT with the 11 filtering rules introduces negligible overhead by up to 8%.

The rest of this paper is organized as follows. Section 2 describes the device emulation in hypervisors and its vulnerability. Section 3 describes our threat model. Section 4 shows the design and implementation of our framework. Section 5 shows the experimental results. Section 6 describes related works. Section 7 concludes the paper.

2 DEVICE VIRTUALIZATION

Guest VMs in multi-tenant cloud environments share a single peripheral device on the same physical server for high utilization and flexibility. To share a single *physical* device among VMs, device emulation in hypervisors provides a *virtual* device for each VM. This section describes the mechanism and security problems of device emulation.

2.1 Device Emulation in Hypervisors

I/O requests to virtual devices are performed by accessing I/O ports or memory-mapped I/O (MMIO) regions in the same way as the requests to physical devices. However, hypervisors must trap and emulate such operations to prevent direct access to physical devices from the VMs because the VMs do not monopolize them.

Suppose that a VM uses IDE as an interface for a virtual disk. I/O requests to the virtual disk from the VM are processed by the hypervisor as follows. 1) The processor raises an exception when a process in the VM accesses its IDE region. 2) The processor transfers the control from the VM to the hypervisor. 3) The device emulator in the hypervisor emulates the I/O requests by changing the virtual disk state. 4) The hypervisor returns the emulated result to the VM and resumes the VM execution.

For example, in KVM [16]+QEMU [8], the device virtualization is performed with the combined effort of KVM and QEMU. Fig. 1 illustrates an architectural overview of KVM+QEMU. KVM is a Linux kernel module responsible for virtualizing CPU and memory, while QEMU is a user process responsible for virtualizing peripheral devices. KVM traps I/O requests to virtual devices and forwards them to QEMU for device emulation. The CPU extension, such as Intel-VT [15] or AMD-V [5], raises an exception named VM-exit when the VM running on KVM makes I/O requests to virtual devices (1.). Then, KVM determines the VM-exit reason (2.). KVM passes the I/O request information to QEMU where the VM-exit reason is I/O or MMIO, and QEMU emulates the request (3.). QEMU returns the result of emulation to KVM through `ioctl` interface (4.). Finally, KVM sets the result for the VM and resumes its execution (5.).

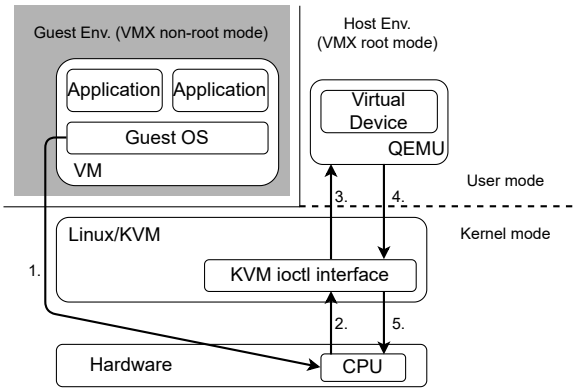

Figure 1: Overview of KVM+QEMU. I/O requests from the VM are operated as follows, (1.) VM-exit by I/O requests from the VM, (2.) KVM determines the VM-exit reason, (3.) KVM passes the I/O request to QEMU, (4.) QEMU returns the result of emulation to KVM, (5.) KVM resumes the VM's execution.

2.2 Vulnerabilities in Device Emulator

Vulnerabilities in the device emulators cause severe security issues because they run in the host context, and the attackers can consequently access resources on the host by exploiting their vulnerabilities. Device emulation is one of the hotbeds of the vulnerabilities of hypervisors.

Table 1 shows examples of vulnerabilities in device emulators. These vulnerabilities have been reported regardless of device types. The following two difficulties in implementing and maintaining device emulators make their vulnerabilities ubiquitous. First, detailed internal knowledge of devices is required to implement their emulators, but the vendors do not often disclose the device internal structures. Since the device specifications intend to help device driver developers, they mostly show the external interface of the devices rather than their internal structures. Second, the software implementation of the devices could make a vulnerability severe compared to the physical devices. Even though the device registers or internal buffers are isolated at the circuit level in hardware, they can affect each other in the device emulator due to software bugs. For example, a buffer overflow in the device emulators can overwrite other data in its process and become a critical security hole.

Performing an *illegal I/O sequence*, a series of illegal I/O requests, is a typical way to exploit vulnerabilities in the device emulators. For example, VENOM (CVE-2015-3456) [21] can be exploited with the illegal I/O sequence shown in Listing 1. VENOM is a vulnerability in QEMU's virtual floppy disk controller (FDC) and affects major hypervisors such as KVM, Xen [7], and Oracle Virtual Box [28]. VENOM allows guest VM users to cause a DoS or possibly execute arbitrary code in the host hypervisor. To exploit VENOM, attackers first issue the `READ_ID` command to transfer the virtual FDC into the command phase (line 6 in Listing 1). Then, the attackers repeatedly

Table 1: Vulnerabilities of Device Emulator. The following are all QEMU vulnerabilities. This table shows the target device emulator, the CVSS v2.0 Score, and the days from the time it was reported until the patch was committed for each CVE ID.

CVE ID	Device	CVSS v2.0 Score	Period [days]
CVE-2015-3456	FDC	7.7	8
CVE-2015-5279	NE2000	7.2	28
CVE-2016-4439	SCSI	4.6	4
CVE-2016-7909	PCnet	4.9	187
CVE-2020-11102	Tulip	6.8	55
CVE-2020-13361	ES1370	3.3	-3
CVE-2020-13800	ATI SVGA	4.9	1
CVE-2020-15863	XGMAC	4.4	18
CVE-2020-25085	SDCHI	4.4	35

```
1  #include <sys/io.h>
2  #define FIFO 0x3f5
3
4  int main() {
5      int i;
6      outb(0x0a,0x3f5); /* READ ID */
7      for (i=0;i<10000000;i++)
8          outb(0x42,0x3f5); /* push */
9  }
```

Listing 1: VENOM PoC code. Issue FDC's READ_ID command by `outb(0x0a,0x3f5)` in line 6. Write to the FIFO buffer repeatedly to cause buffer overflow in line 8 after it.

write data into the FIFO buffer in the virtual FDC through the port I/O (line 8). By writing repeatedly, the FIFO buffer overflows because the device emulator does not check the FIFO index properly. The VM results in a crash with this PoC.

This repeated writing attempt is rejected implicitly where a physical FDC is used. Since this I/O sequence does not happen during normal I/O operations for the physical FDC, we call this sort of I/O sequence as an illegal I/O sequence.

2.3 Problems in Vulnerability Response

Long *vulnerability windows* of hypervisors leave attackers much time to exploit. However, it takes a long time to fix the vulnerabilities by applying the patches. The period column in Table 1 shows the days that are taken to fix each vulnerability. These days are calculated from the date when the vulnerability was reported and the date when the patch was committed. We use Red Hat's bug tracker [13] to obtain the reported dates and qemu.org [3] to get the commit dates. Although the length of vulnerability windows varies depending on the complexity of each vulnerability, the result shows that each vulnerability takes several days to be fixed. Likewise, Ngoc et al. [25] report that fixing vulnerabilities takes one or two months for Xen and 71 days for KVM on average.

Fixing vulnerabilities in device emulators takes a long time for the following two reasons. First, developing a correct patch requires an in-depth analysis of the vulnerability. Second, each patch must be carefully checked to confirm it resolves the vulnerability and does not introduce new vulnerabilities. The hypervisor and its device emulators are complicated because of their code size; thus, careful code modification and regression testing are required to fix the vulnerabilities, even if the size of the code change is small. The long vulnerability window is inevitable as long as we rely on only patches to fix the vulnerabilities.

3 THREAT MODEL

Our threat model assumes that attackers have administrative privileges to operate VMs. In other words, the attackers can execute privileged commands and access virtual devices in guest VMs without any restrictions. The attackers gain administrative privileges in guest VMs by compromising them by exploiting vulnerabilities in guest applications or OSes. The attackers then attempt to exploit vulnerabilities in the device emulators; we assume no vulnerability in other hypervisor components is exploited in our threat model. In addition, our threat model does *not* assume that cloud providers, who have the administrative control over hypervisors, are malicious.

4 DESIGN AND IMPLEMENTATION

4.1 Nioh-PT Overview

In this paper, we propose Nioh-PT, a framework to protect hypervisors temporarily against vulnerabilities in device emulators during the vulnerability windows. As shown in Fig. 2, Nioh-PT resides between VMs and the hypervisor and is composed of the slim monitoring layer and the filtering engine. The monitoring layer intercepts all I/O requests from VMs. The filtering engine detects and stops malicious I/O requests from the intercepted I/O requests based on filtering rules provided by device emulator developers. Since Nioh-PT allows dealing with vulnerabilities without modifying the device emulators and it does not affect the internal states of VMs, it enables system administrators to develop countermeasures instantly when a vulnerability in a device emulator is reported.

Device emulator developers can specify what I/O sequences are illegal as a filtering rule with APIs provided by the I/O filtering engine. The filtering engine filters out the I/O requests that follow the sequences the device emulator developers designated. Rejecting illegal I/O requests does not affect normal device operations because malicious code issues I/O requests in a different flow from benign I/O requests. If an I/O sequence which is identical to a malformed I/O sequence is used in ordinary I/O requests, the problem would have been discovered during the test phase of the device emulator before it is reported as a security vulnerability. Therefore, it is highly expected that VMs can run without interference as long as they issue normal I/O requests only, even if illegal I/O sequences are prohibited.

Dividing Nioh-PT into the monitoring layer and the filtering engine makes it easy to deploy Nioh-PT because the filtering engine can be maintained independently of the device emulator. Adding or removing an I/O filtering rule does not require modifications to the device emulator, and thus can be done without long disruption of the service. However, it takes a long time for the device emulator developers to develop patches to fix vulnerabilities as mentioned in Section 2. In addition, applying the patch involves modifications to the device emulator and implies disruption of the service. This

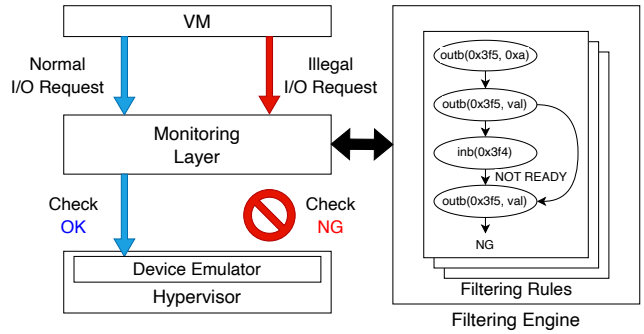

Figure 2: Overview of I/O Filtering Mechanism. Nioh-PT consists of a monitoring layer and a filtering engine. The monitoring layer pass the information of the I/O requests to the filtering engine. The filtering engine filters out illegal I/O requests specified as filtering rules, and the monitoring layer accepts or rejects them.

vulnerability window is serious because hypervisors are exposed to security threats during the window, which could lead to critical security incidents. To reduce the risk of a vulnerability before the patch is released, it is important to provide a defense mechanism that can be applied, modified and removed without changing the code of the hypervisor.

4.2 Slim Monitoring Layer

The monitoring layer intercepts I/O requests from VMs before they are emulated. When a VM is trying to read a value from the device emulator, the monitoring layer obtains the I/O address (MMIO or I/O port address), the access size, and the value read. When a VM is trying to write a value to the device emulator, it obtains the I/O address, the access size, and the value to be written. The monitoring layer passes the obtained information to the filtering engine so that it can validate the I/O sequence.

The monitoring layer is clearly separated from the filtering engine, and not affected by the modification of the filtering engine. The filtering engine evolves by incorporating new filtering rules to defend against new vulnerabilities. Those filtering rules are removed after the patches are applied to the hypervisor to fix those vulnerabilities. In spite of the continual update on the filtering engine, there is no need to update the monitoring layer, and thus the hypervisor can continue to run without any disruption.

4.3 I/O Request Filtering Engine

As mentioned in Section 4.1, the filtering engine determines whether I/O requests from VMs are illegal or not, based on the information from the monitoring layer. Device emulator developers create filtering rules by specifying illegal I/O sequences for each vulnerability with APIs provided by the filtering engine. If there are several ways to exploit a vulnerability, they can create multiple filtering rules to detect each exploitation method. The filtering engine keeps track of I/O requests issued by each VM. If it finds an exact match of the issued I/O sequence with an illegal I/O sequence specified by the filtering rules, it rejects the issued I/O request because it is considered malicious and the VM is attempting to compromise the hypervisor by exploiting the vulnerability in the device emulator.

```
1  let venom_readid = Filter::new()
2    .out(predicate::eq(0xa), 0x3f5)
3    .out(predicate::always(), 0x3f5)
4    .wait_until(
5      Filter::new().inb_update(predicate::function(|&x|
           x & 0x80 != 0), 0x3f4),
6      Filter::new().out(predicate::always(), 0x3f5));
```

Listing 2: Filtering rule for VENOM 1. Monitor READ_ID command issuance and the parameter writing in line 2-3. Prohibit the writing to the FIFO buffer until the VM confirms that the virtual FDC permits the access to the buffer in line 4-6.

Table 2 shows the list of APIs in Nioh-PT to describe filtering rules that specify illegal I/O sequences. An illegal I/O sequence is described by chaining I/O requests, which is represented by an invocation to a corresponding API in Table 2. If an I/O request writes a value to an I/O address, an API call to the `out` family in Table 2 is chained. A method in the `out` family takes two arguments: the first one (`value_condition`) specifies the value written to the I/O address specified in the second argument (`addr`). The first argument, `value_condition`, is specified by a predicate object in Rust [2]. It can express the value is equal to, less than, greater than any Rust expression, or any combination of those primitive predicates. If an I/O request reads a value from an I/O address, an API call to the `in` family is chained. Methods in the `in` family takes an I/O address (`addr`) from which a value is read, and optionally the value to be read (`value_condition`).

When manipulating an I/O device, it is common to wait until it reaches a certain state. For example, it is necessary to wait until a busy bit is cleared before requesting another I/O operation. Nioh-PT provides `wait_until` method to express a certain I/O sequence should be issued after another I/O sequence is issued. In other words, if a certain I/O sequence appears before another sequence, it is considered illegal. Method `wait_until` takes two arguments: the first one (`condition_filter`) specifies an I/O sequence that should be issued before the I/O sequence specified by the second argument (`next_filter`). It is considered illegal if an I/O sequence specified by `next_filter` comes before the I/O sequence specified by `condition_filter`.

Listing 2 shows an example of the filtering rule that defends against the VENOM vulnerability. As described in Section 2, VENOM overflows the FIFO buffer in FDC by writing to it repeatedly after `READ_ID` command is issued. Once `READ_ID` command is issued, no value should be written to the buffer until the device register, mapped to I/O address `0x3f4` in this example, indicates the FIFO buffer is accessible again. In Listing 2, `READ_ID` command is monitored in lines 2-3. This filtering rule prohibits the writing to the FIFO buffer until it is confirmed that the FDC permits the access to the buffer in line 4-7.

4.4 Implementation

We have implemented Nioh-PT prototype for KVM [16] version 4.15.0-153-generic and QEMU [8] version 2.9.50 on Intel x86_64. Nioh-PT is implemented in Rust because it guarantees type- and

Table 2: API List. This table shows the APIs to describe filtering rules that specify illegal I/O sequences.

family	API	description
create	new()	Create a filter that has no rules. Every I/O request is negated.
out	outb(value_condition, addr) outw(value_condition, addr) outl(value_condition, addr) out(value_condition, addr)	An I/O request to write a value specified `value_condition` to the I/O address specified by `addr`. Use an appropriate API depending on the writing size (`outb`, `outw`, `outl`). Use `out` if the size is not relevant.
in	inb(addr) inw(addr) inl(addr) in(addr)	An I/O request to read from the specified I/O address (`addr`). Use an appropriate API depending on the reading size (`inb`, `inw`, `inl`). Use `in` if the size is not relevant.
	inb_update(value_condition, addr) inw_update(value_condition, addr) inl_update(value_condition, addr) in_update(value_condition, addr)	An I/O request to read from `addr`. Argument `value_condition` specifies the value read from the I/O address.
wait	wait_until(condition_filter, next_filter)	An I/O sequence specified by `next_filter` must wait until the I/O sequence specified by `condition_filter` is issued.

memory-safety at the compile time [31]. This characteristic of Rust eliminates the possibility of dangling pointers and buffer overflows without unacceptable overheads. Nioh-PT is a dynamic library linked to QEMU. QEMU emulates I/O devices by reading from or writing to a memory region called `MemoryRegion`. The monitoring layer of Nioh-PT get the contents of I/O requests from VMs. Fig. 3 shows the control flow of I/O processing from a VM to a virtual device via Nioh-PT. The VM sends an I/O requests to KVM (1.). The monitoring layer gets the information about the requested I/O (e.g. read/write operation, the accessed I/O address, the value it attempts to write) (2.). The monitoring layer passes the information to the filtering engine (3.). The filtering engine decides whether the I/O request is illegal or not, according to all the registered filtering rules (4.). If it is illegal, the filter discard the request. Otherwise, the request is passed to the device emulator (5.).

When a filtering rule is registered to the filtering engine, the filtering engine builds an internal data structure that represents the illegal I/O sequence. The filtering engine manages a pointer for each filtering rule that points to the current location in the corresponding sequence. When I/O request information is transferred from the monitoring layer, the filtering engine searches for a match with the I/O requests in the illegal I/O sequences. If there is a match, the pointer is advanced to point to the next I/O request. Since multiple filters can be registered at the same time, all the pointers are advanced if the transferred I/O request matches with multiple I/O sequences. If the pointer comes to the end of the I/O sequence, the filtering engine judges it is illegal, raises a warning, and stops transferring the I/O request to the virtual device. Note that the pointer is reset if the condition is met in `wait_until` because the I/O sequence is considered benign in this case.

5 EVALUATION

In this section, we first analytically examine the security guarantees provided by Nioh-PT against real-world vulnerabilities in QEMU. Then, we show our prototype can defend against CVE-2015-3456 (VENOM) by using the PoC code. Finally, we measure the performance overhead introduced by Nioh-PT. The experimental environment is shown in Table 3.

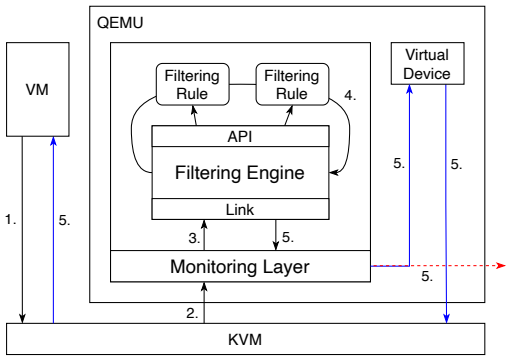

Figure 3: The flow of processing I/O requests. The I/O requests from the VMs are processed in Nioh-PT as follows, (1.) The VM requests I/O to operate the device, (2.) The monitoring layer gets the information of the I/O request, (3.) The monitoring layer pass it to the filtering engine, (4.) The filtering engine filters checks whether the I/O request is illegal, (5.) The monitoring layer accepts or rejects according to the judge of the filtering engine.

5.1 Security Analysis

We develop 11 I/O filtering rules against four real-world vulnerabilities exploited through the port I/O interface. Since the vulnerabilities may have several paths to exploit, we develop more filtering rules than the vulnerabilities. In the rest of this section, we provide the security analysis of six vulnerabilities, including the four vulnerabilities prevented by the filtering rules. The size of I/O filtering rules depends on the number of I/O requests to exploit vulnerabilities. We develop them with seven lines of code (LoC) on average, and 16 LoC at the maximum.

VENOM. As mentioned in Section 2, VENOM is a vulnerability of QEMU's FDC emulator [21]. This vulnerability allows attackers

Table 3: Configuration in Experiments

Host OS	Ubuntu 18.04.5 LTS
Linux Kernel	4.15.0-153-generic
Host CPU	Intel Xeon Silver 4210 CPU @ 2.20GHz
Host Core	10
Host Memory	32 GB
QEMU Version	2.9.50
Guest OS	Ubuntu 18.04.6 LTS
Linux Kernel	4.15.0-166-generic
vCPUs	4
Guest Memory	8 GB

to crash the VM or execute arbitrary code in the host machine by a buffer overflow in the FDC's FIFO buffer. VENOM can be exploited with `READ_ID`, as shown in Section 2. Another FDC command, `DRIVE_SPECIFICATION_COMMAND`, can be used to exploit VENOM. `DRIVE_SPECIFICATION_COMMAND` takes five parameters to execute. Since the FDC emulator does not reset the FIFO buffer index unless the MSB of the fifth argument is set, many write requests to the FIFO buffer at line 8 in Listing 1 causes a buffer overflow [9]. We can specify the filtering rule as shown in Listing 3.

To confirm the effectiveness of Nioh-PT with the filtering rules against VENOM, we conducted the following two experiments. First, we ran the following operations with a virtual floppy disk to verify that the filtering rules allow Nioh-PT to detect a malicious I/O sequence.

(1) Mount a floppy disk on the VM
(2) Execute VENOM PoC code in Listing 1

Next, we performed the following steps to check if normal I/O requests to the virtual FDC are not interfered by the filtering rules.

(1) Mount a floppy disk on the VM
(2) Create a new file on the floppy disk and write 512 KB to it
(3) Add 512 KB to the created file
(4) Delete the file
(5) Unmound the floppy disk from the VM

As a result, only the exploit code was detected by the filtering rules. Nioh-PT with the filtering rules can protect the hypervisor against the malicious I/O requests without interfering with the regular I/O requests to the virtual FDC.

```
1  let venom_drivespec = Filter::new()
2    .out(predicate::eq(0x8e), 0x3f5)
3    .out(predicate::always(), 0x3f5)
4    .out(predicate::always(), 0x3f5)
5    .out(predicate::always(), 0x3f5)
6    .out(predicate::always(), 0x3f5)
7    .out(predicate::function(|&x| x & 0x80 == 0), 0x3f5)
         ;
```

Listing 3: Filtering rule for VENOM 2. Monitor FDC's `DRIVE_SPECIFICATION_COMMAND` command issuance in line 2. Reject the write operation to the buffer if the MSB is not set of the fifth parameter of the command in line 7.

```
1  let cve_2015_5279_pstart = Filter::new()
2    .out(predicate::lt(0x40), 0xc000)
3    .wait_until(
4      Filter::new().out(predicate::ge(0x40), 0xc000),
5      Filter::new().out(predicate::function(|&x| x << 8
           > 49152 as u32), 0xc001));
```

Listing 4: Filtering rule for CVE-2015-5279. This code is for defending the attack by using PSTART. Check the preparation to write to PSTART register first in line 2. Reject the write operation to PSTART register if the value is larger than 49152 in line 3-5.

```
1  let cve_2016_7909_word = Filter::new()
2    .outw(
3      predicate::eq(0x4c)
4        .or(predicate::eq(0xcc))
5        .or(predicate::eq(0x4e))
6        .or(predicate::eq(0xce)),
7      0xc212)
8    .wait_until(
9      Filter::new().outw(
10       predicate::eq(0x4c)
11         .or(predicate::eq(0xcc))
12         .or(predicate::eq(0x4e))
13         .or(predicate::eq(0xce))
14         .not(),
15       0xc212),
16     Filter::new().out(predicate::eq(0), 0xc210));
```

Listing 5: Filtering rule for CVE-2016-7909. Monitor if the accessed device register is CSR76 or CSR78 in line 2-7. Prohibit the writing 0 until the accessed device register change in line 8-16.

CVE-2015-5279. This vulnerability resides in the QEMU's NE2000 network card emulator [22]. It allows attackers to cause a DoS attack or execute arbitrary code through receiving packets. To exploit this vulnerability, an attacker sets inappropriate values in the virtual device registers PSTART, PSTOP, BNRY, and CURR. The device emulator uses the four device registers' values to calculate the receive buffer index. The inappropriate values in these device registers cause out-of-bounds memory access in `ne2000_receive()` function when it receives packets. It is necessary to issue a write request after designating which device register to access through an I/O request to write desired values to a specific device register. We can specify the filtering rule as shown in Listing 4.

CVE-2016-7909. This vulnerability resides in the QEMU's AMD PCnet-PCI II Single-Chip Full-Duplex Ethernet Controller such as Am79C970A [19]. It allows attackers to cause a DoS attack via receiving packets. To exploit this vulnerability, an attacker sets CSR76 or CSR78 device registers to 0. It causes infinite loop through receiving packets via `pcnet_receive()`. To write the desired value to the specific device register, it is necessary to specify the destination register via a port I/O request and then issue a port I/O request to write the value. In normal operation, 0 is never written to CSR76 and CSR78. We can specify the filtering rule as shown in Listing 5.

CVE-2020-13361. This vulnerability resides in the QEMU's ENSONIQ AudioPCI ES1370 emulator [20]. It allows attackers to cause out-of-bounds accesses by setting an inapporopriate value to the virtual device register FRAMECNT. FRAMECNT has two 16 bits values together in 32 bits device register. If the value of the lower 16

bits is smaller than the upper 16 bits, the index based on their values will exceed the buffer size in the virtual device, and then causes out-of-bounds memory through calling `es1370_transfer_audio()`. To write the desired value to FRAMECNT, an attacker issues a write I/O request after specifying a memory page to access via the port I/O. We can specify the filtering rule as shown in Listing 6.

```
1  let cve_2020_13361_dac1 = Filter::new()
2    .out(predicate::eq(0xc), 0xc10c)
3    .wait_until(
4      Filter::new().out(predicate::ne(0xc), 0xc10c),
5      Filter::new().outl(predicate::function(|&x| (x >>
           16 & 0xffff as u32) > x & 0xffff), 0xc134));
```

Listing 6: Filtering rule for CVE-2020-13361. Monitor the specification of the memory page to write to set a value to FRAMECNT in line 2. Prohibit the value that meets the condition until another memory page is set in line 3-5.

CVE-2020-13800. This vulnerability resides in the QEMU's ATI SVGA emulator [24]. It allows attackers to cause infinite recursion by setting a crafted value to MM_INDEX. MM_INDEX works as an index of memory access in the virtual device. `ati_mm_read()` and `ati_mm_write()` call them recursively depending on the value of MM_INDEX. The two functions make recursive calls infinitely where the value of MM_INDEX is seven or less. These values are not set to MM_INDEX during the normal I/O operations. It is possible to prevent exploiting this vulnerability by monitoring setting to MM_INDEX and the request that causes recursion.

CVE-2020-15863. This vulnerability resides in the QEMU's XGMAC Ethernet Controller [23]. It allows attackers to cause a buffer overflow and crash the QEMU process on the host. When it sends packets, the contents are copied from memory to a buffer for transmission. By crafting an in-memory data structure that does not have a sign indicating the end of the packet, an attacker can make out-of-bounds write to the buffer during the copy, which leads to a DoS attack or execution of privileged code. Since this vulnerability does not require I/O requests to exploit, Nioh-PT does not help to protect the hypervisor against exploiting this.

5.2 Performance Evaluation

To measure the overhead introduced by Nioh-PT, we compare the throughput of the bare QEMU, QEMU + Nioh-PT with no filtering rules, and QEMU + Nioh-PT with the 11 filtering rules. Filtering rules can be made for a variety of devices and the filtering engine checks all I/O sequences performed by VM. We measure the overhead of virtual disk as a representative example. For the experience, we use I/O intensive workloads for a virtual disk, `fileserver`, `varmail`, and `webserver` from Filebench [1], which contains a variety of workloads for the filesystem and storage. Fig. 4 shows the throughput normalized by the bare QEMU results for each workload.

Nioh-PT introduces the overhead by up to 8% in `fileserver`. Since I/O virtualization involves the two high-cost context switches, between the guest OS and the host and KVM and QEMU, the overhead introduced by Nioh-PT becomes relatively small. Thanks to the Rust's compiling time check, Nioh-PT does not incur run-time

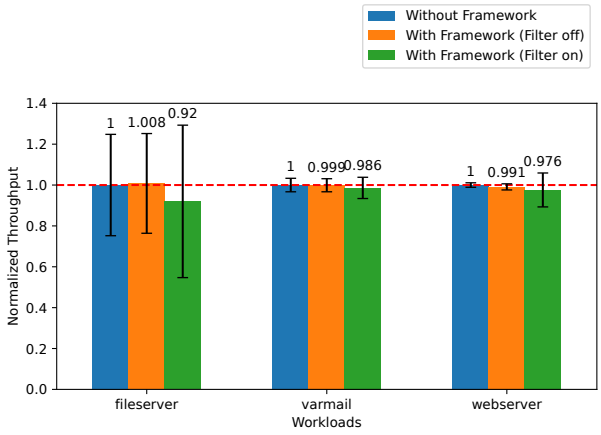

Figure 4: Performance Overhead. This compares the bare QEMU, with Nioh-PT having 0 filters and with Nioh-PT having 11 filters for 4 vulnerabilities based on the bare QEMU.

overhead for memory management like garbage collection. This makes the filtering time by Nioh-PT more constant.

6 RELATED WORK

Protecting hypervisors against malicious guest I/O requests Nioh [27] leverages device specifications to filter out illegal I/O requests from guest VMs. Nioh manages virtual device states as an automaton and rejects illegal I/O requests that do not follow the device specifications. This allows Nioh to protect hypervisors against attacks with illegal I/O requests, including zero-day attacks. However, building I/O filters for Nioh is time-consuming and error-prone because it requires translation from the device specifications written in natural languages into the automatons for Nioh. Moreover, the device specifications may not contain enough information to build the automatons because they focus on describing the device interfaces to develop device drivers. In contrast, the filtering rules for Nioh-PT can be specified without knowledge of the device internals to reduce hypervisor vulnerability windows.

Reducing attack surfaces The large code base and complexity of hypervisors pose security issues. Several researches aim to reduce the attack surfaces of hypervisors by minimizing their trusted computing base (TCB). Min-V [26] focuses on the fact that VMs use limited virtual devices at run-time in cloud environments. To minimize the TCB, Min-V analyzes which virtual devices are required only at boot time and eliminates them from VMs at run-time. De-privileging hypervisors by introducing a more privileged security monitor [10, 18, 34, 37–39], employing a microkernel approach [17, 35], or removing them at run-time [36] can help to reduce hypervisors' attack surfaces. Firecracker [4] and crossvm [12] re-build the part of KVM+QEMU for their specific purposes. Thanks to their simple designs, these works can reduce the attack surfaces of the hypervisor. Most works require a large modification of the hypervisor code base to change the architecture for the reduction in the attack surfaces. In contrast, Nioh-PT requires only modest hypervisor changes to introduce. Hypervisor fuzzing [6, 14, 29, 30, 32, 33] and testing with symbolic

execution [11] can reduce the attack surfaces by uncovering vulnerabilities in hypervisors. These methods complement Nioh-PT because Nioh-PT can provide temporal protection against uncovered vulnerabilities by fuzzing.

Minimizing hypervisor vulnerability windows HyperTP [25] proposes the hypervisor transplant to address hypervisor vulnerability windows. HyperTP enables replacing quickly the current data center hypervisor that has a severe security issue with a different hypervisor. Nioh-PT focuses on vulnerability windows in device emulation that the code base tend to be shared by multiple hypervisors [21].

7 CONCLUSION

It is a common approach to apply patches to fix vulnerabilities in device emulation of hypervisors in order to defend against the exploitation. Since it takes long time to develop the patches, this approach gives attackers a long grace period, or a *vulnerability window*, to exploit the vulnerabilities. We have shown that issuing illegal I/O sequences is a typical method of exploiting vulnerabilities in device emulation. Those illegal I/O sequences are not issued during normal I/O operations. Nioh-PT, proposed in this paper, is a framework to protect KVM+QEMU against exploitation of device emulation vulnerabilities. Given filtering rules which specify illegal I/O sequences, Nioh-PT filters out those illegal sequences to protect the hypervisors. Since the filtering rules can be specified without the internal knowledge on the hypervisors, it is expected to reduce the vulnerability windows. Nioh-PT with 11 filtering rules introduces negligible overhead for the filesystem and storage benchmarks by up to 8%.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was supported by JST AIP Acceleration Research JPMJCR22U3, Japan.

REFERENCES

[1] 2020. Filebench. https://github.com/filebench/filebench
[2] 2022. Crate predicates. https://docs.rs/predicates/latest/predicates/
[3] 2022. git://git.qemu.org/qemu.git. https://git.qemu.org/?p=qemu.git
[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. USENIX Association, 419–434.
[5] AMD. 2021. AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/system/files/TechDocs/24593.pdf
[6] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU Validation. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, 311–327.
[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*. ACM, 164–177.
[8] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association.
[9] CrowdStrike. 2015. VENOM Vulnerability Details. https://www.crowdstrike.com/blog/venom-vulnerability-details/
[10] Liang Deng, Peng Liu, Jun Xu, Ping Chen, and Qingkai Zeng. 2017. Dancing with Wolves: Towards Practical Event-Driven VMM Monitoring. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. ACM, 83–96.
[11] Pedro Fonseca, Xi Wang, and Arvind Krishnamurthy. 2018. MultiNyx: A Multi-Level Abstraction Framework for Systematic Analysis of Hypervisors. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, Article 23.
[12] Google. 2022. crosvm - The Chrome OS Virtual Machine Monitor. https://chromium.googlesource.com/chromiumos/platform/crosvm/
[13] Red Hat. 2022. Red Hat Bugzilla - Main Page. https://bugzilla.redhat.com/
[14] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. 2017. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 3–25.
[15] Intel. [n. d.]. Intel Virtualization Technology (Intel VT). https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html
[16] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'07)*.
[17] Shih-Wei Li, John S. Koh, and Jason Nieh. 2019. Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1357–1374.
[18] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2020. (Mostly) Exitless VM protection from untrusted hypervisor through disaggregated nested virtualization. In *Proceedings of the 29th USENIX Conference on Security Symposium*. USENIX Association, 1695–1712.
[19] National Institute of Standards and Technology. 2020. CVE-2016-7909. https://nvd.nist.gov/vuln/detail/CVE-2016-7909
[20] National Institute of Standards and Technology. 2020. CVE-2020-13361. https://nvd.nist.gov/vuln/detail/CVE-2020-13361
[21] National Institute of Standards and Technology. 2021. CVE-2015-3456. https://nvd.nist.gov/vuln/detail/CVE-2015-3456
[22] National Institute of Standards and Technology. 2021. CVE-2015-5279. https://nvd.nist.gov/vuln/detail/CVE-2015-5279
[23] National Institute of Standards and Technology. 2021. CVE-2020-15863. https://nvd.nist.gov/vuln/detail/CVE-2020-15863
[24] National Institute of Standards and Technology. 2022. CVE-2020-13800. https://nvd.nist.gov/vuln/detail/CVE-2020-13800
[25] Tu Dinh Ngoc, Boris Teabe, Alain Tchana, Gilles Muller, and Daniel Hagimont. 2021. Mitigating vulnerability windows with hypervisor transplant. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. ACM, 162–177.
[26] Anh Nguyen, Himanshu Raj, Shravan Rayanchu, Stefan Saroiu, and Alec Wolman. 2012. Delusional boot: securing hypervisors without massive re-engineering. In *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys '12)*. ACM, 141–154.
[27] Junya Ogasawara and Kenji Kono. 2017. Nioh: Hardening The Hypervisor by Filtering Illegal I/O Requests to Virtual Devices. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, 542–552.
[28] ORACLE. 2022. Oracle VM VirutalBox. https://www.virtualbox.org
[29] Tavis Ormandy. 2007. An empirical study into the security exposure to hosts of hostile virtualized environments *(CanSecWest '07)*.
[30] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. 2021. V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. ACM, 2197–2213.
[31] Rust Team. 2022. Rust. https://www.rust-lang.org/
[32] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2597–2614.
[33] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
[34] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen. In *24th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
[35] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, 209–222.
[36] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. 2011. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, 401–412.
[37] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. 2012. Isolating Commodity Hosted Hypervisors with HyperLock. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, 127–140.
[38] Chiachih Wu, Zhi Wang, and Xuxian Jiang. 2013. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *20th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
[39] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, 203–216.