Hardening Emulated Devices in OpenBSD's vmd(8) Hypervisor

Dave Voutila
OpenBSD
dv@openbsd.org

***Abstract*—The 2010s brought commoditization of hardware-assisted virtualization as now most consumer operating systems and computers ship with both support in hardware as well as Type-2 hypervisors. With hypervisors comes the need for emulated devices to provide virtual machines interfaces to the outside world, including network cards, disk controllers, and even hardware random number generators. However, hypervisors are still software programs and consequently subject to buffer overflow and stack smashing attacks like any other. Previous research has shown a common weak point in hypervisors to be these very emulated devices where exploits enable "guest to host escapes", the most famous being an exploit of an emulated floppy disk controller.**

This paper provides an experimental approach with OpenBSD's Type-2 hypervisor (vmd) for isolating emulated devices using the privilege dropping and separation capabilities available in OpenBSD 7.2 to mitigate techniques for turning memory bugs into guest-to-host hypervisor escapes.

***Index Terms*—OpenBSD, virtualization, security**

I. Introduction

One of the earliest documented guest-to-host escapes was made possible by a buffer overflow of an emulated video card in the Xen hypervisor (CVE-2008-1943 [1]). Since then, every major hypervisor whether open-source or commercial, has had something in common: a buffer overflow or unitialized memory bug in an emulated device allowing for exploitation by a malicious guest operating system. This raises two concerns:

- the attractiveness of attacking emulated devices in a hypervisor,
- the accessibility of modern techniques to exploit these memory bugs when found.

A. Emulated Devices as Targets in Hypervisors

Like with real computers, devices form the interface between a virtual machine and the outside world. While hardware-assisted virtualization allow CPU-intensive tasks to achieve bare-metal speeds, at some point the guest will need to perform I/O whether sending a network packet or writing a block to persistent storage. Given I/O becomes a noticeable performance bottleneck in virtualized systems, hypervisor authors often optimize in multiple ways:

- emulate a device in the host kernel to reduce overhead
- emulate all devices in the same process via threads
- pass-through access to a real, physical device

As a result, emulated devices often require or evolve to acquire elevated permissions and capabilities, making them a high-value target for an attacker.

The high-valued nature isn't the only problem: emulated devices must work with guest device *drivers*. Hypervisor authors must create virtual hardware from software and, while there exist virtualization-specific specifications like VirtIO [2], the nature of I/O requires moving guest-supplied data back and forth. A memory bug easily becomes "remotely" triggerable by a guest device driver or a even a packet coming in to the host from outside.

Given the above, it should come as little surprise that almost all published virtual machine "escapes" [3] have been as the result of exploiting emulated devices!

B. Return-Oriented Programming

At the same time researchers began to find the first guest-to-host exploits in hypervisors, other researchers found novel ways to go beyond simple code-injection techniques and use a program's code against itself.

Return-Oriented Programming (ROP) [3] provides a manner to defeat W$\oplus$X mitigations, commonplace in operating systems with the support of a hardware "no-execute" (NX) page protection bit. ROP attacks use either the program itself or runtime libraries like *libc* to execute arbitrary code assembled by finding and leveraging specificly useful machine instructions called "gadgets." These gadgets, when executed in a particular order (called a "chain"), allow an attacker to achieve arbitrary code execution.

Multiple approaches exist to help prevent a successful ROP attack, including Adress Space Randomization (ASLR) and Control Flow Integrity (CFI) [4].

C. Blind Return-Oriented Programming

In 2014, *Bittau et. al.* showcased an evolution of ROP, called "Blind Return-Oriented Programming" (BROP) [5], designed to overcome ASLR techniques and remotely exploit stack buffer overflows through information leakage. The authors' techniques of "stack reading" to defeat ASLR and remotely generate ROP chains (sequences of gadgets) to achieve arbitrary code execution emphasized the severity of stack or heap overflows in programs: *if it's remotely accessible and doesn't re-randomize itself on a restart, any remotely triggerable stack buffer overflow provides a BROP attack vector.*

How do these concerns apply to hypervisors? While there are no currently known successful BROP-based attacks on hypervisors, the concept of *information leakage* still applies. If an attacker can use a vulnerable emulated device to leak the hypervisor code into guest memory (assuming no program crash), it's possible to perform a ROP analysis.

II. Defensive Concerns of the Ideal Hypervisor

Considering the state-of-the-art of ROP-based attacks, what would the "ideal" hypervisor consider? BROP has shown it's important to consider the following:

1) Information leakage allows for defeating ASLR.
2) ROP gadgets allow assembling any needed system call.
3) System calls allow lateral movement to take over a host.

The ideal hypervisor would maximize the complexity of defeating ASLR. While the trivial approach would be to simply not respawn after a crash (which isn't a common hypervisor trait anways), one can propose:

Ideal Trait 1 *An information leak in one vulnerable component of a hypervisor must not inform on other components of the hypervisor.*

And what about ROP gadgets? They're impossible to completely remove from the x86 architecture, so while they can be minimized by changes to compilers, in the ideal case our hypervisor would minimize their value:

Ideal Trait 2 *Compromising a component of a hypervisor must not allow for compromising other components of the hypervisor, i.e. a vulnerable network device must not allow for compromising other devices.*

Lastly, keeping with the principle of least privilege:

Ideal Trait 3 *Escaping a guest, via any means, must force the attacker to then exploit the hypervisor itself to gain control of the host.*

While we can harden emulated devices, the difficulty posed to the attacker should *stay constant or increase* even if they manage to take control of an emulated device.

III. OpenBSD's Hypervisor

Available since OpenBSD 5.9, released on 29 March, 2016, OpenBSD's hypervisor consists of three parts:

- `vmm(4)` - the in-kernel virtual machine monitor
- `vmd(8)` - the userland virtual machine daemon
- `vmctl(8)` - a utility for interacting with vmd

This paper focuses primarily on `vmd(8)` because it provides the emulated devices of interest (e.g. VirtIO network devices).

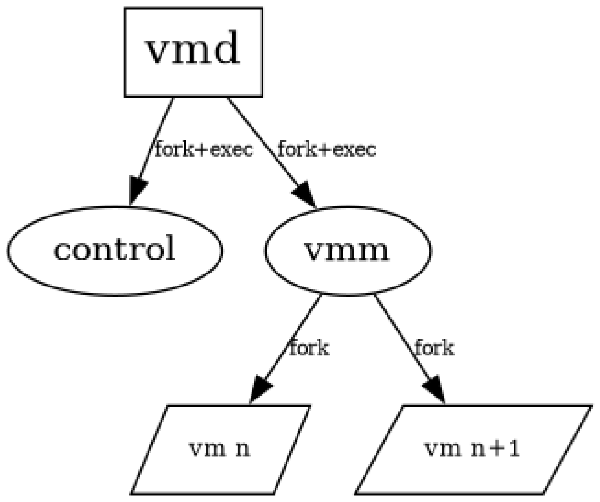

Fig. 1. vmd(8)'s first priv-sep design.

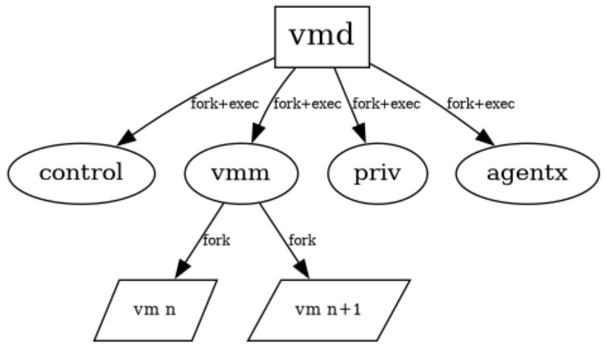

Fig. 2. vmd(8)'s priv-sep design as of OpenBSD 7.2.

A. *Privilege Separation in* `vmd`

While not in the original release of `vmd`, Reyk Floeter committed [6] a redesign to implement a *fork+exec* model like the one already used in other OpenBSD daemons such as `httpd(8)`.

Shortly thereafter, Reyk added [7] an additional "priv" process intended to run as root and facilitate privledged operations such as naming host-side `tap(4)` interfaces to match the name of the vm.

B. *Privilege Dropping in* `vmd`

In addition to separating `vmd` into multiple processes with dedicated responsibilities, it additionally uses methods to reduce the privilges of each process. As of OpenBSD 7.2, `vmd` incorporates three (3) primary mechanisms for dropping privilege:

1) `setresuid(2)/setresgid(2)` - for changing uid/gid
2) `chroot(2)` - for isolating file system access

TABLE I
EXISTING PRIVILEGE DROPPING IN `VMD(8)`

process	uid/gid	chroot	pledge(s)
parent	root	$PWD	stdio rpath wpath proc tty recvfd sendfd getpw chown fattr flock
control	_vmd	$PWD	stdio unix recvfd sendfd
agentx	_vmd	/	stdio recvfd unix
priv	root	$PWD	–
vmm	_vmd	/	stdio vmm sendfd recvfd proc
vm	_vmd	$PWD	stdio vmm recvfd

3) `pledge(2)` - for removing system call access

In short, the syscalls as early as feasible during program start to adjust privledges to those in the Table I.

C. *Existing Weaknesses in* `vmd`

Even with the existing PrivSep design and mitigations, the following remains true for `vmd` as of OpenBSD 7.2:

1) All devices are emulated in the same guest vm process.
2) Guest vm process creation relies only on `fork(2)`, meaning address layouts are the same across guest vms.

Given our proposed ideal traits in section II, there are identifiable gaps in the current design of `vmd` with respect to device emulation. Consequently, *a compromised device in one machine exposes all guests under* `vmd` *to risk.*

IV. HARDENING A `VMD` DEVICE

We'll use the "ideal traits" to design our hardening methodology. Implementing each trait requires using multiple capabilities of OpenBSD. Let's look at them in order.

A. *Maximizing Randomness*

The first step in implementing *Trait 1* is solving for the 2nd weakness outlined above.

To make each guest have their own address space layout, it should be as simple as performing the `exec(2)` part of "fork+exec," right? `vmd` poses some challenges towards implementation *because of its existing PrivSep design!*

Firstly, the *vmm* process responsible for spawning new vm processes doesn't utilize the "exec" `pledge(2)` promise. This means the `execvp(2)` syscall is prohibited. But that's an easy change; just add the promise, right?

Unfortunately, because *vmm* utilizes `chroot(2)`, it won't have access to the `vmd(8)` executable in the file system. Luckily, we can leverage `unveil(2)` and the known path to the vmd executable to approximate the same outcome of minimizing filesystem access.

As a consequence, we need to make the following changes:

1) Remove *vmm*'s `chroot(2)`.

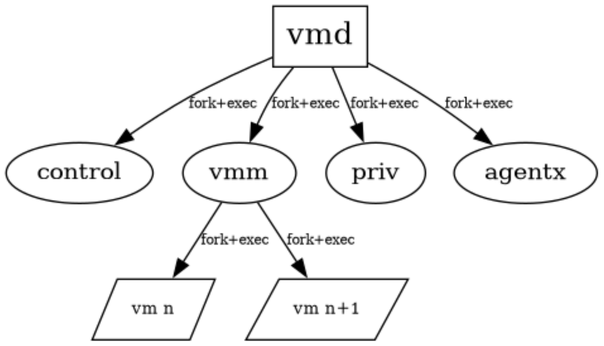

Fig. 3. vmd(8)'s priv-sep after adding fork+exec for vm's.

2) `unveil(2)` /usr/sbin/vmd in executable mode.
3) Add "exec" to *vmm*'s promises.
4) Set the child-side of the `socketpair(2)` file descriptors to not close-on-exec.
5) Implement message passing to bootstrap the new *vm* process since we can no longer rely on existing global variables.

The last part (message passing) requires the most effort. For now, the following approach is used:

1) Add a new `getopt(3)` argument to indicate we're launching a new *vm*-based process and pass that argument (`-V`) when exec'ing.
2) Pass the file descriptor integer for the child-side of the `socketpair(2)`.
3) Use synchronous message passing to send the child *vm* process its configuration values.

At this point, the privledge separation diagram looks like:

B. *Minimizing the Impact – Isolating a Device*

`vmd` has had its share of security errata and, like most hypervisors, most have been related to emulating a network device. How can we isolate devices?

The most obvious approach is to simply make each device its own process, each with its own address space like we did with the *vm* process in section IV-A. Ultimately, we want to achieve a design illustrated in figure IV-B.

For this, we'll take one of the more complicated and higher value targets of the emulated devices in `vmd`: the emulated `vio(4)` network device. This poses multiple challenges.

1) Sharing Guest Physical Memory: The first challenge, and perhaps the most critical, is that the device needs the ability to read and write directly to guest memory. The guest memory was already allocated and mapped by the vm process, so we need to share the pages between vm process and device process.

This change primarily occurs in the vm process. When allocating the memory for the backing guest memory ranges, we can use `shm_mkstemp(3)` to create a temporary shared

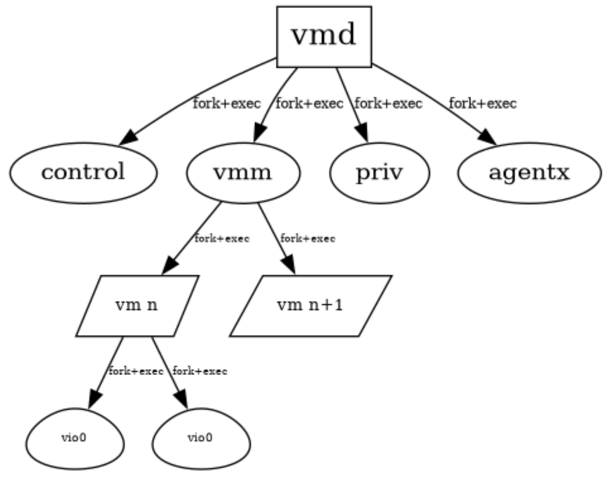

Fig. 4. Isolating multiple VirtIO nics for a vm.

memory object to use when calling `mmap(2)`. Since the mapping is no longer anonymous, we can make sure that the file descriptor is not set to close-on-exec. All that's left is to incorporate the file descriptor into a configuration message to send to the device after `execvp(2)`.

2) Bootstrapping the Device Process: We need to update the *vm* process to do the `fork(2)/execvp(2)` dance. That's easy enough as previously we added the `-V` flag and can use the same pattern of passing configuration data after re-exec. The configuration message needs to contain the value of the shared memory file descriptor and the already open `tap(4)` file descriptor.

Like before, we establish a synchronous communication channel and pass the file descriptor value via the program arguments so the new device process can start communicating and receiving the configuration data.

However, unlike the *vmm-to-vm* communication channel, we need an additional asynchronous one to allow for event-based communication.

3) Communicating with the Device: The vm process has two threads that may need to communicate with the device: the vcpu thread that will emulate IN/OUT instructions to the PCI registers and the event loop thread handling asynchronous events.

For the vcpu thread, we need a *synchronous* channel. When emulating a PCI register read, the vcpu will expect a response immediately. (Writes are trivial as no response is needed.) Since this is the vcpu thread, it cannot leverage any async scheduling of *imsg*'s since it doesn't own the `event(3)` *event_base* lest it corrupt existing events!

For the event threads, we need an *asynchronous* channel. One message in particular is relayed from the *vmm* process to the *vm* process: an update to the host-side MAC. (This occurs at some time post vm-launch.) Other async messages include pausing and unpausing the device when the vm is being paused or unpaused.

In either case, *imsg* functions are used to guarantee atomic delivery.

4) Communicating with the Network: A functioning `vio(4)` device requires the following for operation:

- Two virtqueues (*TX* and *RX*).
- An open file descriptor to the host's `tap(4)` device.

The same mechanisms already in place for using an event loop for reading off the `tap(4)` can be reused in this case. The only difference is now when the device needs to assert or deassert the IRQ, it needs to broadcast a message to the vm process using the asynchronous channel.

C. Escaping into a Void – Reducing the Device Surface Area

If all else fails, escaping a *device* should leave the attacker needing to now find additional exploits to elevate privilege. Let's assume a bug in the `vio(4)` device allows a guest-to-host escape. What are the next potential escalation paths an attacker will want to chain together to get root on the host?

1) Securing the File System: One trivial target is the filesystem, either to exfiltrate sensitive data (like private keys), exploit race conditions, or trick another program to execute something. The device isn't running as root, so we can't simply `chroot(2)` to `/var/empty`, but we can leverage `unveil(2)` and achieve a similar result.

D. Removing System Calls

We want to prevent lateral movement and that means preventing system calls. Thankfully, this is trivial with `pledge(2)` and, moreover, we can *further reduce privileges*. Using just the "stdio" promise, we reduce to just a minimal subset of syscalls we need for reading and writing our open file descriptors and managing our events.

This does mean that an attacker escaping the guest and controlling the device process can read(2)/write(2) but the possible targets are limited to the existing file descriptors (communication channels with the vm and the host `tap(4)`). However, no new sockets can be created nor can any files in the file system be opened.

Any privilege escalation will need to exploit a smaller surface area than exposed by the vm process and need to rely on kernel bugs, most likely in this limited area.

V. SECURITY! BUT AT WHAT COST?

If Meltdown and Spectre taught the average user anything it's that security often comes at the price of performance. What impact do the proposed architectural changes to `vmd` have on things like network performance from the point of view of the guest?

TABLE II
IPERF3 PERFORMANCE TEST

Host	Guest	Bitrate (Gbps)	Δ (%)
-current	OpenBSD 7.2	0.86	
prototype	OpenBSD 7.2	1.40	63%
-current	Alpine Linux 3.17	1.30	
prototype	Alpine Linux 3.17	1.14	-14%

A. A Simple TCP Benchmark

While this research doesn't aim to perform a full performance evaluation at this time, one known area of poor network performance is when using a TCP performance test program and having the guest act as the client. Anecdotally, this often shows noticeably worse performance than the reverse (having the host act as the client).

Utilizing a Lenovo X1 Carbon laptop (10th generation model) with the an Intel i7-1270P CPU, the observations in Table II were observed using `iperf3(1)` from (chosen as it's available on both OpenBSD and Alpine). Both guests were allocated 8 GiB of memory and the recorded result is the best average throughput reported across 3 runs of 60 second duration.

B. Interpretation of Results

Since `vmd(8)` does not support multi-processor guests, it isn't too surprising that we can see a performance improvement from this design for OpenBSD guests. As a consequence of the message-based approach, some of the network IO can occur without blocking either the vcpu thread or event handling thread (responsible for the `libevent` event handler) in the main vm process. For instance, in OpenBSD 7.2, writes by a vcpu to an emulated PCI register will block while the device emulates them and potentially injects an interrupt, resulting in a syscall via `ioctl(2)`.

The potential performance regression in Alpine Linux is not substantial, but does warrant further investigation.

VI. Future Work and Considerations

The prototype has rough edges, specifically around robustness of lifecycle management. Improving child process handling in the event of program termination as well as preventing possible messaging deadlocks at launch would improve viability.

While the performance improvement of networking throughput in OpenBSD guests was a pleasant surprise, the lack of improvement in Alpine Linux guests shows there's still potential. The single thread design of the device process is easier to debug, but prevents simultaneous transmit and receive processing.

In addition, relying on syscalls and trips through the kernel to communicate adds extra overhead. While unmeasured at this point, if it's deemed a bottleneck then exploring messaging via a shared page of memory and newer Intel process features like `TPAUSE` instructions might reduce latency.

Lastly, extending the design to VirtIO block devices would be ideal.

VII. Conclusions

OpenBSD contains all the necessary tools for implementing an advanced hypervisor design that improves security without complicating user experience. While this paper doesn't explore existing approaches from systems such as *QEMU*, the idea of isolating devices has been attempted by multiple hypervisors, but has yet to become the *default* behavior.

The outlined design and approach for `vmd(8)` presents a viable way to bring the isolation needed to take another step towards the "ideal hypervisor" without the expense of operator or user complexity. This approach keeps with the spirit and design of `pledge(2)` and `unveil(2)`: it's the developer's responsibility to study and improve the program, *not the user's.*

Acknowledgment

The author would like to thank the AsiaBSDCon program committee for the opportunity to present on this research. Many thanks also to Mike Larkin (`mlarkin@`) for his guidance and mentorship while hacking on `vmd(8)`. Lastly, the author's spouse deserves immense credit for supporting the time and obsession required for attempting this research.

References

[1] Red Hat Bugzilla, https://bugzilla.redhat.com/show_bug.cgi?id=443078, accessed December 2023.

[2] Virtual I/O Device (VIRTIO) Version 1.1. Edited by Michael S. Tsirkin and Cornelia Huck, 11 April 2019, OASIS Committee Specification 01, https://docs.oasis-open.org/virtio/virtio/v1.1/cs01/virtio-v1.1-cs01.html.

[3] Wikipedia, "Virtual machine escape", https://en.wikipedia.org/wiki/Virtual_machine_escape#Previous_known_vulnerabilities, retrieved 26 December, 2022.

[4] V. Pappas, "Defending against Return-Oriented Programming", Columbia University, 2015.

[5] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)", Proceedings of the CCS 2007, ACM Press, pp. 552-61, 2007.

[6] OpenBSD GitHub mirror, https://github.com/openbsd/src/commit/bcc679a146056243a2fd52a28182621f893fed4b

[7] OpenBSD GitHub mirror, https://github.com/openbsd/src/commit/5921535c0be28fd3cf226c9c6a0aa8bb71699acb