Project Zero

News and updates from the Project Zero team at Google

Tuesday, June 29, 2021

An EPYC escape: Case-study of a KVM breakout

Posted by Felix Wilhelm, Project Zero

Introduction

KVM (for Kernel-based Virtual Machine) is the de-facto standard hypervisor for Linux-based cloud environments. Outside of Azure, almost all large-scale cloud and hosting providers are running on top of KVM, turning it into one of the fundamental security boundaries in the cloud.

In this blog post I describe a vulnerability in KVM's AMD-specific code and discuss how this bug can be turned into a full virtual machine escape. **To the best of my knowledge, this is the first public writeup of a KVM guest-to-host breakout that does not rely on bugs in user space components such as QEMU. The discussed bug was assigned CVE-2021-29657, affects kernel versions v5.10-rc1 to v5.12-rc6 and was patched at the end of March 2021.** As the bug only became exploitable in v5.10 and was discovered roughly 5 months later, most real world deployments of KVM should not be affected. I still think the issue is an interesting case study in the work required to build a stable guest-to-host escape against KVM and hope that this writeup can strengthen the case that hypervisor compromises are not only theoretical issues.

I start with a short overview of KVM's architecture, before diving into the bug and its exploitation.

KVM

KVM is a Linux based open source hypervisor supporting hardware accelerated virtualization on x86, ARM, PowerPC and S/390. In contrast to the other big open source hypervisor Xen, KVM is deeply integrated with the Linux Kernel and builds on its scheduling, memory management and hardware integrations to provide efficient virtualization.

KVM is implemented as one or more kernel modules (kvm.ko plus kvm-intel.ko or kvm-amd.ko on x86) that expose a low-level IOCTL-based API to user space processes over the /dev/kvm device. Using this API, a user space process (often called VMM for Virtual Machine Manager) can create new VMs, assign vCPUs and memory, and intercept memory or IO accesses to provide access to emulated or virtualization-aware hardware devices. QEMU has been the standard user space choice for KVM-based virtualization for a long time, but in the last few years alternatives like LKVM, crosvm or Firecracker have started to become popular.

While KVM's reliance on a separate user space component might seem complicated at first, it has a very nice benefit: Each VM running on a KVM host has a 1:1 mapping to a Linux process, making it managable using standard Linux tools.

This means for example, that a guest's memory can be inspected by dumping the allocated memory of its user space process or that resource limits for CPU time and memory can be applied easily. Additionally, KVM can offload most work related to device emulation to the userspace component. Outside of a couple of performance-sensitive devices related to interrupt handling, all of the complex low-level code for providing virtual disk, network or GPU access can be implemented in userspace.

When looking at public writeups of KVM-related vulnerabilities and exploits it becomes clear that this design was a wise decision. The large majority of disclosed vulnerabilities and all publicly available exploits affect QEMU and its support for emulated/paravirtualized devices.

Even though KVM's kernel attack surface is significantly smaller than the one exposed by a default QEMU configuration or similar user space VMMs, a KVM vulnerability has advantages that make it very valuable for an attacker:

- Whereas user space VMMs can be sandboxed to reduce the impact of a VM breakout, no such option is available for KVM itself. Once an attacker is able to achieve code execution (or similarly powerful primitives like write access to page tables) in the context of the host kernel, the system is fully compromised.
- Due to the somewhat poor security history of QEMU, new user space VMMs like crosvm or Firecracker are written in Rust, a memory safe language. Of course, there can still be non-memory safety vulnerabilities or problems due to incorrect or buggy usage of the KVM APIs, but using Rust

Search This Blog

Search

Pages

- About Project Zero
- Working at Project Zero
- 0day "In the Wild"
- 0day Exploit Root Cause Analyses
- Vulnerability Disclosure FAQ

Archives

2021

- An EPYC escape: Case-study of a KVM breakout (Jun)
- Fuzzing iOS code on macOS at native speed (May)
- Designing sockfuzzer, a network syscall fuzzer for... (Apr)
- Policy and Disclosure: 2021 Edition (Apr)
- Who Contains the Containers? (Apr)
- In-the-Wild Series: October 2020 0-day discovery (Mar)
- Déjà vu-lnerability (Feb)
- A Look at iMessage in iOS 14 (Jan)
- Windows Exploitation Tricks: Trapping Virtual Memo... (Jan)
- The State of State Machines (Jan)
- Hunting for Bugs in Windows Mini-Filter Drivers (Jan)
- In-the-Wild Series: Android Post-Exploitation (Jan)
- In-the-Wild Series: Windows Exploits (Jan)
- In-the-Wild Series: Android Exploits (Jan)
- In-the-Wild Series: Chrome Infinity Bug (Jan)
- In-the-Wild Series: Chrome Exploits (Jan)
- Introducing the In-the-Wild Series (Jan)

2020

- An iOS hacker tries Android (Dec)
- An iOS zero-click radio proximity exploit odyssey (Dec)
- Oops, I missed it again! (Nov)
- Enter the Vault: Authentication Issues in HashiCor... (Oct)

effectively prevents the large majority of bugs that were discovered in C-based user space VMMs in the past.

- Finally, a pure KVM exploit can work against targets that use proprietary or heavily modified user space VMMs. While the big cloud providers do not go into much detail about their virtualization stacks publicly, it is safe to assume that they do not depend on an unmodified QEMU version for their production workloads. In contrast, KVM's smaller code base makes heavy modifications unlikely (and KVM's contributor list points at a strong tendency to upstream such modifications when they exist).

With these advantages in mind, I decided to spend some time hunting for a KVM vulnerability that could be turned into a guest-to-host escape. In the past, I had some success with finding vulnerabilities in KVM's support for nested virtualization on Intel CPUs so reviewing the same functionality for AMD seemed like a good starting point. This is even more true, because the recent increase of AMD's market share in the server segment means that KVM's AMD implementation is suddenly becoming a more interesting target than it was in the last years.

Nested virtualization, the ability for a VM (called L1) to spawn nested guests (L2), was also a niche feature for a long time. However, due to hardware improvements that reduce its overhead and increasing customer demand it's becoming more widely available. For example, Microsoft is heavily pushing for Virtualization-based Security as part of newer Windows versions, requiring nested virtualization to support cloud-hosted Windows installations. KVM enables support for nested virtualization on both AMD and Intel by default, so if an administrator or the user space VMM does not explicitly disable it, it's part of the attack surface for a malicious or compromised VM.

AMD's virtualization extension is called SVM (for Secure Virtual Machine) and in order to support nested virtualization, the host hypervisor needs to intercept all SVM instructions that are executed by its guests, emulate their behavior and keep its state in sync with the underlying hardware. As you might imagine, implementing this correctly is quite difficult with a large potential for complex logic flaws, making it a perfect target for manual code review.

The Bug

Before diving into the KVM codebase and the bug I discovered, I want to quickly introduce how AMD SVM works to make the rest of the post easier to understand. (For a thorough documentation see AMD64 Architecture Programmer's Manual, Volume 2: System Programming Chapter 15.) SVM adds support for 6 new instructions to x86-64 if SVM support is enabled by setting the SVME bit in the EFER MSR. The most interesting of these instructions is *VMRUN*, which (as its name suggests) is responsible for running a guest VM. *VMRUN* takes an implicit parameter via the RAX register pointing to the page-aligned physical address of a data structure called "virtual machine control block" (VMCB), which describes the state and configuration of the VM.

The VMCB is split into two parts: First, the State Save area, which stores the values of all guest registers, including segment and control registers. Second, the Control area which describes the configuration of the VM. The Control area describes the virtualization features enabled for a VM, sets which VM actions are intercepted to trigger a VM exit and stores some fundamental configuration values such as the page table address used for nested paging.

If the VMCB is correctly prepared (and we are not already running in a VM), VMRUN will first save the host state in a memory region called the host save area, whose address is configured by writing a physical address to the VM_HSAVE_PA MSR. Once the host state is saved, the CPU switches to the VM context and VMRUN only returns once a VM exit is triggered for one reason or another.

An interesting aspect of SVM is that a lot of the state recovery after a VM exit has to be done by the hypervisor. Once a VM exit occurs, only RIP, RSP and RAX are restored to the previous host values and all other general purpose registers still contain the guest values. In addition, a full context switch requires manual execution of the VMSAVE/VMLOAD instructions which save/load additional system registers (FS, SS, LDTR, STAR, LSTAR …) from memory.

For nested virtualization to work, KVM intercepts execution of the VMRUN instruction and creates its own VMCB based on the VMCB the L1 guest prepared (called vmcb12 in KVM terminology). Of course, KVM can't trust the guest provided vmcb12 and needs to carefully validate all fields that end up in the real VMCB that gets passed to the hardware (known as vmcb02).

Most of the KVM's code for nested virtualization on AMD is implemented in arch/x86/kvm/svm/nested.c and the code that intercepts VMRUN instructions of nested guests is implemented in `nested_svm_vmrun`:

```
int nested_svm_vmrun(struct vcpu_svm *svm)
{
        int ret;
        struct vmcb *vmcb12;
        struct vmcb *hsave = svm->nested.hsave;
        struct vmcb *vmcb = svm->vmcb;
        struct kvm_host_map map;
        u64 vmcb12_gpa;


        vmcb12_gpa = svm->vmcb->save.rax; ** 1 **
        ret = kvm_vcpu_map(&svm->vcpu, gpa_to_gfn(vmcb12_gpa), &map); ** 2 **
        …
        ret = kvm_skip_emulated_instruction(&svm->vcpu);

        vmcb12 = map.hva;


        if (!nested_vmcb_checks(svm, vmcb12)) { ** 3 **
                vmcb12->control.exit_code    = SVM_EXIT_ERR;
                vmcb12->control.exit_code_hi = 0;
                vmcb12->control.exit_info_1  = 0;
                vmcb12->control.exit_info_2  = 0;
                goto out;
        }

        ...

        /*
         * Save the old vmcb, so we don't need to pick what we save, but can
         * restore everything when a VMEXIT occurs
         */
        hsave->save.es     = vmcb->save.es;
        hsave->save.cs     = vmcb->save.cs;
        hsave->save.ss     = vmcb->save.ss;
        hsave->save.ds     = vmcb->save.ds;
        hsave->save.gdtr   = vmcb->save.gdtr;
        hsave->save.idtr   = vmcb->save.idtr;
        hsave->save.efer   = svm->vcpu.arch.efer;
        hsave->save.cr0    = kvm_read_cr0(&svm->vcpu);
        hsave->save.cr4    = svm->vcpu.arch.cr4;
        hsave->save.rflags = kvm_get_rflags(&svm->vcpu);
        hsave->save.rip    = kvm_rip_read(&svm->vcpu);
        hsave->save.rsp    = vmcb->save.rsp;
        hsave->save.rax    = vmcb->save.rax;
        if (npt_enabled)
                hsave->save.cr3    = vmcb->save.cr3;
        else
                hsave->save.cr3    = kvm_read_cr3(&svm->vcpu);

        copy_vmcb_control_area(&hsave->control, &vmcb->control);

        svm->nested.nested_run_pending = 1;

        if (enter_svm_guest_mode(svm, vmcb12_gpa, vmcb12)) ** 4 **
                goto out_exit_err;

        if (nested_svm_vmrun_msrpm(svm))
                goto out;

out_exit_err:
        svm->nested.nested_run_pending = 0;

        svm->vmcb->control.exit_code    = SVM_EXIT_ERR;
        svm->vmcb->control.exit_code_hi = 0;
        svm->vmcb->control.exit_info_1  = 0;
        svm->vmcb->control.exit_info_2  = 0;

        nested_svm_vmexit(svm);

out:
        kvm_vcpu_unmap(&svm->vcpu, &map, true);

        return ret;
}
```

The function first fetches the value of RAX out of the currently active vmcb (`svm->vcmb`) in 1 (numbers are marked in the code samples). For guests using nested paging (which is the only relevant configuration nowadays) RAX contains a guest physical address (GPA), which needs to be translated into a host physical address (HPA) first. `kvm_vcpu_map` (2) takes care of this translation and maps the underlying page to a host virtual address (HVA) that can be directly accessed by KVM.

Once the VMCB is mapped, `nested_vmcb_checks` is called for some basic validation in 3. Afterwards, the L1 guest context which is stored in `svm->vmcb` is copied into the host save area `svm->nested.hsave` before

- KTRW: The journey to build a debuggable iPhone (Oct)
- The story of Adobe Reader symbols (Oct)
- Windows Exploitation Tricks: Spoofing Name... (Sep)
- A very deep dive into iOS Exploit chains found in ... (Aug)
- In-the-wild iOS Exploit Chain 1 (Aug)
- In-the-wild iOS Exploit Chain 2 (Aug)
- In-the-wild iOS Exploit Chain 3 (Aug)
- In-the-wild iOS Exploit Chain 4 (Aug)
- In-the-wild iOS Exploit Chain 5 (Aug)
- Implant Teardown (Aug)
- JSC Exploits (Aug)
- The Many Possibilities of CVE-2019-8646 (Aug)
- Down the Rabbit-Hole... (Aug)
- The Fully Remote Attack Surface of the iPhone (Aug)
- Trashing the Flow of Data (May)
- Windows Exploitation Tricks: Abusing the User-Mode... (Apr)
- Virtually Unlimited Memory: Escaping the Chrome Sa... (Apr)
- Splitting atoms in XNU (Apr)
- Windows Kernel Logic Bug Class: Access Mode Mismat... (Mar)
- Android Messaging: A Few Bugs Short of a Chain (Mar)
- The Curious Case of Convexity Confusion (Feb)
- Examining Pointer Authentication on the iPhone XS (Feb)
- voucher_swap: Exploiting MIG reference counting in... (Jan)
- Taking a page from the kernel&#39;s book: A TLB issue ... (Jan)

2018

- On VBScript (Dec)
- Searching statically-linked vulnerable library fun... (Dec)
- Adventures in Video Conferencing Part 5: Where Do ... (Dec)
- Adventures in Video Conferencing Part 4: What Didn... (Dec)
- Adventures in Video Conferencing Part 3: The Even ... (Dec)
- Adventures in Video Conferencing Part 2: Fun with ... (Dec)
- Adventures in Video Conferencing Part 1: The Wild ... (Dec)
- Injecting Code into Windows Protected Processes us... (Nov)
- Heap Feng Shader: Exploiting SwiftShader in Chrome (Oct)
- Deja-XNU (Oct)
- Injecting Code into Windows Protected Processes us... (Oct)
- 365 Days Later: Finding and Exploiting Safari Bugs... (Oct)

KVM enters the nested guest context by calling `enter_svm_guest_mode` (4).

```
int enter_svm_guest_mode(struct vcpu_svm *svm, u64 vmcb12_gpa,
                         struct vmcb *vmcb12)
{
        int ret;

        svm->nested.vmcb12_gpa = vmcb12_gpa;
        load_nested_vmcb_control(svm, &vmcb12->control);
        nested_prepare_vmcb_save(svm, vmcb12);
        nested_prepare_vmcb_control(svm);

        ret = nested_svm_load_cr3(&svm->vcpu, vmcb12->save.cr3,
                                  nested_npt_enabled(svm));
        if (ret)
                return ret;

        svm_set_gif(svm, true);

        return 0;
}

static void load_nested_vmcb_control(struct vcpu_svm *svm,
                                     struct vmcb_control_area *control)
{
        copy_vmcb_control_area(&svm->nested.ctl, control);

        ...
}
```

Looking at `enter_svm_guest_mode` we can see that KVM copies the vmcb12 control area directly into svm->nested.ctl and does not perform any further checks on the copied value.
Readers familiar with double fetch or Time-of-Check-to-Time-of-Use vulnerabilities might already see a potential issue here: The call to `nested_vmcb_checks` at the beginning of `nested_svm_vmrun` performs all of its checks on a copy of the VMCB that is stored in guest memory. This means that a guest with multiple CPU cores can modify fields in the VMCB after they are verified in `nested_vmcb_checks`, but before they are copied to svm->nested.ctl in `load_nested_vmcb_control`.

Let's look at `nested_vmcb_checks` to see what kind of checks we can bypass with this approach:

```
static bool nested_vmcb_check_controls(struct vmcb_control_area *control)
{
        if ((vmcb_is_intercept(control, INTERCEPT_VMRUN)) == 0)
                return false;

        if (control->asid == 0)
                return false;

        if ((control->nested_ctl & SVM_NESTED_CTL_NP_ENABLE) &&
            !npt_enabled)
                return false;

        return true;
}
```

At first glance this looks pretty harmless. `control->asid` isn't used anywhere and the last check is only relevant for systems where nested paging isn't supported. However, the first check turns out to be very interesting.

For reasons unknown to me, SVM VMCBs contain a bit that enables or disables interception of the VMRUN instruction when executed inside a guest. Clearing this bit isn't actually supported by hardware and results in an immediate VMEXIT, so the check in `nested_vmcb_check_controls` simply replicates this behavior. When we race and bypass the check by repeatedly flipping the value of the INTERCEPT_VMRUN bit, we can end up in a situation where svm->nested.ctl contains a 0 in place of the INTERCEPT_VMRUN bit. To understand the impact we first need to see how nested vmexit's are handled in KVM:

The main SVM exit handler is the function `handle_exit` in arch/x86/kvm/svm.c, which is called whenever a VMexit occurs. When KVM is running a nested guest, it first has to check if the exit should be handled by itself or the L1 hypervisor. To do this it calls the function `nested_svm_exit_handled` (5) which will return NESTED_EXIT_DONE if the vmexit will be handled by the L1 hypervisor and no further processing by the L0 hypervisor is needed:

```
 static int handle_exit(struct kvm_vcpu *vcpu, fastpath_t exit_fastpath)
{
        struct vcpu_svm *svm = to_svm(vcpu);
        struct kvm_run *kvm_run = vcpu->run;
        u32 exit_code = svm->vmcb->control.exit_code;

        …

        if (is_guest_mode(vcpu)) {
                int vmexit;

                trace_kvm_nested_vmexit(exit_code, vcpu, KVM_ISA_SVM);

                vmexit = nested_svm_exit_special(svm);

                if (vmexit == NESTED_EXIT_CONTINUE)
                        vmexit = nested_svm_exit_handled(svm); ** 5 **

                if (vmexit == NESTED_EXIT_DONE)
                        return 1;
        }
}

static int nested_svm_intercept(struct vcpu_svm *svm)
{
        // exit_code==INTERCEPT_VMRUN when the L2 guest executes vmrun
        u32 exit_code = svm->vmcb->control.exit_code;
        int vmexit = NESTED_EXIT_HOST;

        switch (exit_code) {
        case SVM_EXIT_MSR:
                vmexit = nested_svm_exit_handled_msr(svm);
                break;
        case SVM_EXIT_IOIO:
                vmexit = nested_svm_intercept_ioio(svm);
                break;
        …
        default: {
                if (vmcb_is_intercept(&svm->nested.ctl, exit_code)) ** 7 **
                        vmexit = NESTED_EXIT_DONE;
        }
        }

        return vmexit;
}

int nested_svm_exit_handled(struct vcpu_svm *svm)
{
        int vmexit;

        vmexit = nested_svm_intercept(svm); ** 6 **

        if (vmexit == NESTED_EXIT_DONE)
                nested_svm_vmexit(svm); ** 8 **

        return vmexit;
}
```

`nested_svm_exit_handled` first calls `nested_svm_intercept (6)` to see if the exit should be handled. When we trigger an exit by executing VMRUN in a L2 guest, the default case is executed (7) to see if the INTERCEPT_VMRUN bit in svm->nested.ctl is set. Normally, this should always be the case and the function returns NESTED_EXIT_DONE to trigger a nested VM exit from L2 to L1 and to let the L1 hypervisor handle the exit (8). (This way KVM supports infinite nesting of hypervisors).

However, if the L1 guest exploited the race condition described above svm->nested.ctl won't have the INTERCEPT_VMRUN bit set and the VM exit will be handled by KVM itself. This results in a second call to `nested_svm_vmrun` while still running inside the L2 guest context. `nested_svm_vmrun` isn't written to handle this situation and will blindly overwrite the L1 context stored in `svm->nested.hsave` with data from the currently active `svm->vmcb` which contains data for the L2 guest:

```
      /*
         * Save the old vmcb, so we don't need to pick what we save, but can
         * restore everything when a VMEXIT occurs
         */
        hsave->save.es     = vmcb->save.es;
        hsave->save.cs     = vmcb->save.cs;
        hsave->save.ss     = vmcb->save.ss;
        hsave->save.ds     = vmcb->save.ds;
        hsave->save.gdtr   = vmcb->save.gdtr;
        hsave->save.idtr   = vmcb->save.idtr;
        hsave->save.efer   = svm->vcpu.arch.efer;
        hsave->save.cr0    = kvm_read_cr0(&svm->vcpu);
        hsave->save.cr4    = svm->vcpu.arch.cr4;
        hsave->save.rflags = kvm_get_rflags(&svm->vcpu);
        hsave->save.rip    = kvm_rip_read(&svm->vcpu);
        hsave->save.rsp    = vmcb->save.rsp;
        hsave->save.rax    = vmcb->save.rax;
        if (npt_enabled)
                hsave->save.cr3    = vmcb->save.cr3;
        else
                hsave->save.cr3    = kvm_read_cr3(&svm->vcpu);

        copy_vmcb_control_area(&hsave->control, &vmcb->control);
```

This becomes a security issue due to the way Model Specific Register (MSR) intercepts are handled for nested guests:
SVM uses a permission bitmap to control which MSRs can be accessed by a VM. The bitmap is a 8KB data structure with two bits per MSR, one of which controls read access and the other write access. A 1 bit in this position means the access is intercepted and triggers a vm exit, a 0 bit means the VM has direct access to the MSR. The HPA address of the bitmap is stored in the VMCB control area and for normal L1 KVM guests, the pages are allocated and pinned into memory as soon as a vCPU is created.

For a nested guest, the MSR permission bitmap is stored in `svm->nested.msrpm` and its physical address is copied into the active VMCB (in `svm->vmcb->control.msrpm_base_pa`) while the nested guest is running. Using the described double invocation of `nested_svm_vmrun`, a malicious guest can copy this value into the `svm->nested.hsave` VMCB when `copy_vmcb_control_area` is executed. This is interesting because the KVM's hsave area normally only contains data from the L1 guest context so `svm->nested.hsave.msrpm_base_pa` would normally point to the pinned vCPU-specific MSR bitmap pages.

This edge case becomes exploitable thanks to a relatively recent change in KVM:
Since commit "2fcf4876: KVM: nSVM: implement on demand allocation of the nested state" from last October, svm->nested.msrpm is dynamically allocated and freed when a guest changes the SVME bit of the MSR_EFER register:

```
int svm_set_efer(struct kvm_vcpu *vcpu, u64 efer)
{
        struct vcpu_svm *svm = to_svm(vcpu);
        u64 old_efer = vcpu->arch.efer;
        vcpu->arch.efer = efer;

        if ((old_efer & EFER_SVME) != (efer & EFER_SVME)) {
                if (!(efer & EFER_SVME)) {
                        svm_leave_nested(svm);
                        svm_set_gif(svm, true);
                        ...                       /*
                         * Free the nested guest state, unless we are in SMM.
                         * In this case we will return to the nested guest
                         * as soon as we leave SMM.
                         */
                        if (!is_smm(&svm->vcpu))
                                svm_free_nested(svm);

                } ...
}
}
```

For the "disable SVME" case, KVM will first call `svm_leave_nested` to forcibly leave potential nested guests and then free the `svm->nested` data structures (including the backing pages for the MSR permission bitmap) in `svm_free_nested`. As `svm_leave_nested` believes that `svm->nested.hsave` contains the saved context of the L1 guest, it simply copies its control area to the real VMCB:

```
void svm_leave_nested(struct vcpu_svm *svm)
{
        if (is_guest_mode(&svm->vcpu)) {
                struct vmcb *hsave = svm->nested.hsave;
                struct vmcb *vmcb = svm->vmcb;

                ...
                copy_vmcb_control_area(&vmcb->control, &hsave->control);
                ...
        }
}
```

But as mentioned before, `svm->nested.hsave->control.msrpm_base_pa` can still point to `svm->nested->msrpm`. Once `svm_free_nested` is finished and KVM passes control back to the guest, the CPU will use the freed pages for its MSR permission checks. This gives a guest unrestricted access to host

- In-Console-Able (May)
- A Tale of Two Exploits (Apr)
- Taming the wild copy: Parallel Thread Corruption (Mar)
- Exploiting the DRAM rowhammer bug to gain kernel p... (Mar)
- Feedback and data-driven updates to Google's discl... (Feb)
- (^Exploiting)\s*(CVE-2015-0318)\s*(in)\s*(Flash$) (Feb)
- A Token's Tale (Feb)
- Exploiting NVMAP to escape the Chrome sandbox - CV... (Jan)
- Finding and exploiting ntpd vulnerabilities (Jan)

2014

- Internet Explorer EPM Sandbox Escape CVE-2014-6350 (Dec)
- pwn4fun Spring 2014 - Safari - Part II (Nov)
- Project Zero Patch Tuesday roundup, November 2014 (Nov)
- Did the "Man With No Name" Feel Insecure? (Oct)
- More Mac OS X and iPhone sandbox escapes and kerne... (Oct)
- Exploiting CVE-2014-0556 in Flash (Sep)
- The poisoned NUL byte, 2014 edition (Aug)
- What does a pointer look like, anyway? (Aug)
- Mac OS X and iPhone sandbox escapes (Jul)
- pwn4fun Spring 2014 - Safari - Part I (Jul)
- Announcing Project Zero (Jul)

MSRs if the pages are reused and partially overwritten with zeros.

To summarize, a malicious guest can gain access to host MSRs using the following approach:

1. Enable the SVME bit in MSR_EFER to enable nested virtualization
2. Repeatedly try to launch a L2 guest using the VMRUN instruction while flipping the INTERCEPT_VMRUN bit on a second CPU core.
3. If VMRUN succeeds, try to launch a “L3” guest using another invocation of VMRUN. If this fails, we have lost the race in step 2 and must try again. If VMRUN succeeds we have successfully overwritten `svm->nested.hsave` with our L2 context.
4. Clear the SVME bit in MSR_EFER while still running in the “L3” context. This frees the MSR permission bitmap backing pages used by the L2 guest who is now executing again.
5. Wait until the KVM host reuses the backing pages. This will potentially clear all or some of the bits, giving the guest access to host MSRs.

When I initially discovered and reported this vulnerability, I was feeling pretty confident that this type of MSR access should be more or less equivalent to full code execution on the host. While my feeling turned out to be correct, getting there still took me multiple weeks of exploit development. In the next section I’ll describe the steps to turn this primitive into a guest-to-host escape.

The Exploit

Assuming our guest can get full unrestricted access to any MSR (which is only a question of timing thanks to init_on_alloc=1 being the default for most modern distributions), how can we escalate this into running arbitrary code in the context of the KVM host? To answer this question we first need to look at what kind of MSRs are supported on a modern AMD system. Looking at the BIOS and Kernel Developer’s Guide for recent AMD processors we can find a wide range of MSRs starting with well known and widely used ones such as EFER (the Extended Feature Enable Register) or LSTAR (the syscall target address) to rarely used ones like SMI_ON_IO_TRAP (can be used to generate a System Management Mode Interrupt when specific IO port ranges are accessed).
Looking at the list, several registers like LSTAR or KERNEL_GSBASE seem like interesting targets for redirecting the execution of the host kernel. Unrestricted access to these registers is actually enabled by default, however they are automatically restored to a valid state by KVM after a vmexit so modifying them does not lead to any changes in host behavior.

Still, there is one MSR that we previously mentioned and that seems to give us a straightforward way to achieve code execution: The VM_HSAVE_PA that stores the physical address of the host save area, which is used to restore the host context when a vmexit occurs. If we can point this MSR at a memory location under our control we should be able to fake a malicious host context and execute our own code after a vmexit.

While this sounds pretty straightforward in theory, implementing it still has some challenges:

- AMD is pretty clear about the fact that software should not touch the host save area in any way and that the data stored in this area is CPU-dependent: “*Processor implementations may store only part or none of host state in the memory area pointed to by VM_HSAVE_PA MSR and may store some or all host state in hidden on-chip memory. Different implementations may choose to save the hidden parts of the host’s segment registers as well as the selectors. For these reasons, software must not rely on the format or contents of the host state save area, nor attempt to change host state by modifying the contents of the host save area.*” (AMD64 Architecture Programmer’s Manual, Volume 2: System Programming, Page 477). To strengthen the point, the format of the host save area is undocumented.
- Debugging issues involving an invalid host state is very tedious as any issue leads to an immediate processor shutdown. Even worse, I wasn’t sure if rewriting the VM_HSAVE_PA MSR while running inside a VM can even work. It’s not really something that should happen during normal operation so in the worst case scenario, overwriting the MSR would just lead to an immediate crash.
- Even if we can create a valid (but malicious) host save area in our guest, we still need some way to identify its host physical address (HPA). Because our guest runs with nested paging enabled, physical addresses that we can see in the guest (GPAs) are still one address translation away from their HPA equivalent.

After spending some time scrolling through AMD’s documentation, I still decided that VM_HSAVE_PA seems to be the best way forward and decided to tackle these problems one by one.

After dumping the host save area of a normal KVM guest running on an AMD EPYC 7351P CPU, the first problem goes away quickly: As it turns out, the host save area has the same layout as a normal VMCB with only a couple of relevant fields initialized. Even better, the initialized fields include all the saved host information documented in the AMD manual so the fear that all interesting host state is stored in on-chip memory seems to be unfounded.

Saving Host State. To ensure that the host can resume operation after #VMEXIT, VMRUN saves at least the following host state information:

- CS.SEL, NEXT_RIP—The CS selector and rIP of the instruction following the VMRUN. On #VMEXIT the host resumes running at this address.
- RFLAGS, RAX—Host processor mode and the register used by VMRUN to address the VMCB.
- SS.SEL, RSP—Stack pointer for host

- CRO, CR3, CR4, EFER—Paging/operating mode for host
- IDTR, GDTR—The pseudo-descriptors. VMRUN does not save or restore the host LDTR.
- ES.SEL and DS.SEL.

Under the mistaken assumption that I solved the problem of creating a fake but valid host save area, I decided to look into building an infoleak that gives me the ability to translate GPAs to HPAs. A couple hours of manual reading led me to an AMD-specific performance monitoring feature called Instruction Based Sampling (IBS). When IBS is enabled by writing the right magic invocation to a set of MSRs, it samples every Nth instruction that is executed and collects a wide range of information about the instruction. This information is logged in another set of MSRs and can be used to analyze the performance of any piece of code running on the CPU. While most of the documentation for IBS is pretty sparse or hard to follow, the very useful open source project AMD IBS Toolkit contains working code, a readable high level description of IBS and a lot of useful references.

IBS supports two different modes of operation, one that samples Instruction fetches and one that samples micro-ops (which you can think of as the internal RISC representation of more complex x64 instructions). Depending on the operation mode, different data is collected. Besides a lot of caching and latency information that we don't care about, fetch sampling also returns the virtual address and physical address of the fetched instruction. Op sampling is even more useful as it returns the virtual address of the underlying instruction as well as virtual and physical addresses accessed by any load or store micro op.

Interestingly, IBS does not seem to care about the virtualization context of its user and every physical address returned by it is an HPA (of course this is not a problem outside of this exploit as guest accesses to the IBS MSR's will normally be restricted). The wide range of data returned by IBS and the fact that it's completely driven by MSR reads and writes make it the perfect tool for building infoleaks for our exploit.

Building a GPA -> HPA leak boils down to enabling IBS ops sampling, executing a lot of instructions that access a specific memory page in our VM and reading the IBS_DC_PHYS_AD MSR to find out its HPA:

```
// This function leaks the HPA of a guest page using
// AMD's Instruction Based Sampling. We try to sample
// one of our memory loads/writes to *p, which will
// store the physical memory address in MSR_IBC_DH_PHYS_AD
static u64 leak_guest_hpa(u8 *p) {
  volatile u8 *ptr = p;
  u64 ibs = scatter_bits(0x2, IBS_OP_CUR_CNT_23) |
            scatter_bits(0x10, IBS_OP_MAX_CNT) | IBS_OP_EN;

  while (true) {
    wrmsr(MSR_IBS_OP_CTL, ibs);
    u64 x = 0;

    for (int i = 0; i < 0x1000; i++) {

      x = ptr[i];
      ptr[i] += ptr[i - 1];
      ptr[i] = x;

      if (i % 50 == 0) {
        u64 valid = rdmsr(MSR_IBS_OP_CTL) & IBS_OP_VAL;
        if (valid) {
          u64 op3 = rdmsr(MSR_IBS_OP_DATA3);
          if ((op3 & IBS_ST_OP) || (op3 & IBS_LD_OP)) {
            if (op3 & IBS_DC_PHY_ADDR_VALID) {
              printf("[x] leak_guest_hpa: %lx %lx %lx\n", rdmsr(MSR_IBS_OP_RIP),
                     rdmsr(MSR_IBS_DC_PHYS_AD), rdmsr(MSR_IBS_DC_LIN_AD));
              return rdmsr(MSR_IBS_DC_PHYS_AD) & ~(0xFFF);
            }
          }

          wrmsr(MSR_IBS_OP_CTL, ibs);
        }
      }
    }

    wrmsr(MSR_IBS_OP_CTL, ibs & ~IBS_OP_EN);
  }
}
```

Using this infoleak primitive, I started to create a fake host save area by preparing my own page tables (for pointing CR3 at them), interrupt descriptor tables and segment descriptors and pointing RIP to a primitive shellcode that would write to the serial console. Of course, my first tries immediately crashed the whole system and even after spending multiple days to make sure everything was set up correctly, the system would crash immediately once I pointed the hsave MSR at my own location.

After getting frustrated with the total lack of progress, watching my server reboot for the hundredth time, trying to come up with a different exploitation strategy for two weeks and learning about the surprising regularity of physical page migrations on Linux, I realized that I made an important mistake. Just because the CPU initializes all the expected fields in the host save area, it is not safe to assume that these fields are actually used for restoring the host context. Slow trial and error led to the discovery that my AMD EPYC CPU ignores everything in the host save area besides the values of the RIP, RSP and RAX registers.

While this register control would make a local privilege escalation straightforward, escaping the VM boundary is a bit more complicated. RIP and RSP control make launching a kernel ROP chain the next logical step, but this requires us to first break the host kernel's address randomization and to find a way to store controlled data at a known host virtual address (HVA).

Fortunately, we have IBS as a powerful infoleak building primitive and can use it to gather all required information in a single run:

- Leaking the host kernel's (or more specifically kvm-amd.ko's) base address can be done by enabling IBS sampling with a small sampling interval and immediately triggering a VM exit. When VM execution continues, the IBS result MSRs will contain the HVA of instructions executed by KVM during the exit handling.
- The most powerful way to store data at a known HVA is to leak the location of the kernel's linear mapping (also known as physmap), a 1:1 mapping of all physical pages on the system. This gives us a GPA->HVA translation primitive by first using our GPA->HPA infoleak from above and then adding the HPA to the physmap base address. Leaking the physmap is possible by sampling micro ops in the host kernel until we find a read or write operation, where the lower ~30 bits of the accessed virtual address and physical address are identical.

Having all these building blocks in place, we could now try to build a kernel ROP chain that executes some interesting payload. However, there is one important caveat. When we take over execution after a vmexit, the system is still in a somewhat unstable state. As mentioned above, SVM's context switching is very minimal and we are at least one VMLOAD instruction and reenabling of interrupts away from a usable system. While it is surely possible to exploit this bug and to restore the original host context using a sufficiently complex ROP chain, I decided to find a way to run my own code instead.

A couple of years ago, the Linux physmap was still mapped executable and executing our own code would be as simple as jumping to a physmap mapping of one of our guest pages. Of course, that is not possible anymore and the kernel tries hard to not have any memory pages mapped as writable and executable. Still, page protections only apply to virtual memory accesses so why not use an instruction that directly writes controlled data to a physical address? As you might remember from our initial discussion of SVM earlier in this chapter, SVM supports an instruction called VMSAVE to store hidden guest state (or host state) in a VMCB. Similar to VMRUN, VMSAVE takes a physical address to a VMCB stored in the RAX register as an implicit argument. It then writes the following register state to the VMCB:

- FS, GS, TR, LDTR
- KernelGsBase
- STAR, LSTAR, CSTAR, SFMASK
- SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP

For us, VMSAVE is interesting for a couple of reasons:

- It is used as part of KVM's normal SVM exit handler and can be easily integrated into a minimal ROP chain.
- It operates on physical addresses, so we can use it to write to an arbitrary memory location including KVM's own code.
- All written registers still contain the guest values set by our VM, allowing us to control the written content with some restrictions

VMSAVE's biggest downside as an exploitation primitive is that RAX needs to be page aligned, reducing our control of the target address. VMSAVE writes to the memory offsets 0x440-0x480 and 0x600-0x638 so we need to be careful about not corrupting any memory that's in use.
In our case this turns out to be a non-issue, as KVM contains a couple of code pages where functions that are rarely or never used (e.g cleanup_module or SEV specific code) are stored at these offsets.

While we don't have full control over the written data and valid register values are somewhat restricted, it is still possible to write a minimal stage0 shellcode to an arbitrary page in the host kernel by filling guest MSRs with the right values. My exploit uses the STAR, LSTAR and CSTAR registers for this which are written to the physical offsets 0x400, 0x408 and 0x410. As all three registers need to contain canonical addresses, we can only use parts of the registers for our shellcode and use relative jumps to skip the unusable parts of the STAR and LSTAR MSRs:

```
// mov cr0, rbx; jmp
wrmsr(MSR_STAR, 0x00000003ebc3220f);
// pop rdi; pop rsi; pop rcx; jmp
wrmsr(MSR_LSTAR, 0x00000003eb595e5fULL);
// rep movsb; pop rdi; jmp rdi;
wrmsr(MSR_CSTAR, 0xe7ff5fa4f3);
```

The above code makes use of the fact that we control the value of the RBX register and the stack when we return to it as part of our initial ROP chain. First, we copy the value of RBX (0x80040033) into CR0, which disables Write Protection (WP) for kernel memory accesses. This makes all of the kernel code writable on this CPU allowing us to copy a larger stage1 shellcode to an arbitrary unused memory location and jump to it.

Once the WP bit in cr0 is disabled and the stage1 payload executes, we have a wide range of options. For my proof-of-concept exploit I decided on a somewhat boring but easy-to-implement approach to spawn a random user space command: The host is still in a very weird state so our stage1 payload can't directly call into other kernel functions, but we can easily backdoor a function pointer which will be called at some later point in time. KVM uses the kernel's global workqueue feature to regularly synchronize a VM's clock between different vCPUs. The function pointer responsible for this work is stored in the (per VM) kvm->arch data structure as kvm->arch.kvmclock_update_work. The stage1 payload overrides this function pointer with the address of a stage2 payload. To put the host into a usable state it then sets the VM_HSAVE_PA MSR back to its original value and restores RSP and RIP to call the original vmexit handler.

The final stage2 payload executes at some later point in time as part of the kernel global work queue and uses the call_usermodehelper to run an arbitrary command with root privileges.

Let's put all of this together and walk through the attacks step-by-step:

1. Prepare the stage0 payload by splitting it up and setting the right guest MSRs.
2. Trigger the TOCTOU vulnerability in nested_svm_vmrun and free the MSR permission bitmap by disabling the SVME bit in the EFER MSR.
3. Wait for the pages to be reused and initialized to 0 to get unrestricted MSR access.
4. Prepare a fake host save area, a stack for the initial ROP chain and a staging memory area for the stage1 and stage2 payloads.
5. Leak the HPA of the host save area, the HVA addresses of the stack and staging page and the kvm-amd.ko's base address using the different IBS infoleaks.
6. Redirect execution to the VMSAVE gadget by setting RIP, RSP and RAX in the fake host save area, pointing the VM_HSAVE_PA MSR at it and triggering a VM exit.
7. VMSAVE writes the stage0 payload to an unused offset in kvm-amd's code segment, when the gadget returns stage0 gets executed.
8. stage0 disables Write Protection in CR0 and overwrites an unused executable memory location with the stage1 and stage2 payloads, before jumping to stage1.
9. stage1 overwrites kvm->arch.kvmclock_update_work.work.func with a pointer to stage2 before restoring the original host context.
10. At some later point in time kvm->arch.kvmclock_update_work.work.func is called as part of the global kernel work_queue and stage2 spawns an arbitrary command using call_usermodehelper.

Interested readers should take a look at the heavily documented proof-of-concept exploit for the actual implementation.

Conclusion

This blog post describes a KVM-only VM escape made possible by a small bug in KVM's AMD-specific code for supporting nested virtualization. Luckily, the feature that made this bug exploitable was only included in two kernel versions (v5.10, v5.11) before the issue was spotted, reducing the real-life impact of the vulnerability to a minimum. The bug and its exploit still serve as a demonstration that highly exploitable security vulnerabilities can still exist in the very core of a virtualization engine, which is almost certainly a small and well audited codebase. While the attack surface of a hypervisor such as KVM is relatively small from a pure LoC perspective, its low level nature, close interaction with hardware and pure complexity makes it very hard to avoid security-critical bugs.

While we have not seen any in-the-wild exploits targeting hypervisors outside of competitions like Pwn2Own, these capabilities are clearly achievable for a well-financed adversary. I've spent around two months on this research, working as an individual with only remote access to an AMD system. Looking at the potential ROI on an exploit like this, it seems safe to assume that more people are working on similar issues right now and that vulnerabilities in KVM, Hyper-V, Xen or VMware will be exploited in-the-wild sooner or later.

What can we do about this? Security engineers working on Virtualization Security should push for as much attack surface reduction as possible. Moving complex functionality to memory-safe user space components is a big win even if it does not help against bugs like the one described above. Disabling unneeded or unreviewed features and performing regular in-depth code reviews for new changes can further reduce the risk of bugs slipping by.

Hosters, cloud providers and other enterprises that are relying on virtualization for multi-tenancy isolation should design their architecture in way that limits the impact of an attacker with an VM escape exploit:

- Isolation of VM hosts: Machines that host untrusted VMs should be considered at least partially untrusted. While a VM escape can give an attacker full control over a single host, it should not be easily possible to move from one compromised host to another. This requires that the control plane and backend infrastructure is sufficiently hardened and that user resources like disk images or encryption keys are only exposed to hosts that need them. One way to limit the impact of a VM escape even further is to only run VMs of a specific customer or of a certain sensitivity on a single machine.

- Investing in detection capabilities: In most architectures, the behavior of a VM host should be very predictable, making a compromised host stick out quickly once an attacker tries to move to other systems. While it’s very hard to rule out the possibility of a vulnerability in your virtualization stack, good detection capabilities make life for an attacker much harder and increase the risk of quickly burning a high-value vulnerability. Agents running on the VM host can be a first (but bypassable) detection mechanism, but the focus should be on detecting unusual network communication and resource accesses.

Posted by Ryan at 8:58 AM

No comments:

Post a Comment

Home Older Post