Breaking Turtles All the Way Down: An Exploitation Chain to Break out of VMware ESXi

Hanqing Zhao [‡†], Yanyu Zhang[†], Kun Yang[*], Taesoo Kim [‡]

[†]*Chaitin Security Research Lab,*
[‡]*Georgia Institute of Technology,*
[*]*Tsinghua University*

Abstract

VMware ESXi is an enterprise-class, bare-metal hypervisor dedicated to providing the state-of-the-art private-cloud infrastructures. Accordingly, the design and implementation of *ESXi* is of our community's interest, yet lacking a thorough evaluation of its security internals. In this paper, we give a comprehensive analysis of the guest-to-host attack surfaces of *ESXi* and its recent security mitigation (i.e., the vSphere sandbox). In particular, we introduce an effective and reliable approach to chain multiple vulnerabilities for exploitation and demonstrate our approach by leveraging two new bugs (i.e., uninitialized stack usages), namely, CVE-2018-6981 and CVE-2018-6982. Our exploit chain is the first public demonstration of a virtual machine escape against *VMware ESXi*.

1 Introduction

Cloud computing has become the most popular choice for today's online services. The foundation that enables cloud computing is virtualization technology, effectively allowing economy of scale. Some examples include *QEMU-KVM*, *Xen*, *Hyper-V*, and VMware Families, which are the basic building blocks of most public and private cloud infrastructures.

Unfortunately, its large and complicated code base of virtual machines inevitably includes software vulnerabilities such as memory corruptions. Moreover, unlike other software bugs, the bugs in virtualization infrastructures could lead to a break-in of the security boundary between guest and host; thus, the security of the entire cloud environment of the cloud provider might be subverted.

These unique propositions of cloud infrastructures make them an attractive target for attackers, but our communities lack, an in-depth analysis and evaluation of their security. Research communities have made several attempts to do this: one example that received considerable attention is a security bug of *QEMU-KVM*, called VENOM (known as CVE-2015-3456 [8]), which attempted to exploit a bug in a virtual, emulated floppy drive. Other vulnerabilities such as CVE-2015-5165 and CVE-2015-7504 have been disclosed and exploited in public PoC demonstrations [17]. However, most of them, as far as we know, rely on strict preconditions (e.g., supporting a decade-old floppy driver by the cloud provider), which fail to demonstrate exploitation under the default configuration of the underlying virtualization infrastructures. Admittedly, there have been a few public demonstrations against *VMware Workstation* in the Pwn2Own contest, but their approaches and details of exploitation have not been disclosed to the public.

The situation for enterprise-class, commercial hypervisors, such as *VMware ESXi* or *Hyper-V*, is even worse: they are much more attractive targets to attackers, but their internals still remain opaque to the security communities. Such security-by-obscurity approaches taken by commercial type-1 hypervisor require that many practical challenges be addressed. First, the complex internal design and machinery to implement a type-1 hypervisors is not trivial to understand in depth. Second, hypervisors are intensively protected by custom in-house protection schemes, limiting the capability for dynamic analysis. Lastly, non-trivial reverse engineering skills and efforts are required to understand and even navigate system binaries without proper source codes and symbols.

In this paper, we share the lessons learned from our in-depth, security analysis of *ESXi*. We first introduce the attack surfaces of *ESXi* by systematically evaluating attack scenarios. Second, we provide an exhaustive analysis of two previously unknown vulnerabilities we found in *ESXi*. Last, by leveraging these vulnerabilities, we elaborate a new exploit technique to bypass the deployed mitigation, the vSphere sandbox in *ESXi*. The constructed exploit is reliable and persistent; it dynamically adopts the version of *ESXi* and survives across the system reboot, causing persistent damage to cloud providers.

Overall, this paper makes the following contributions:

1. A systematic review of attack surfaces, mitigations, and vSphere sandbox of *VMware ESXi*.

2. An in-depth demonstration and analysis of the vulner-

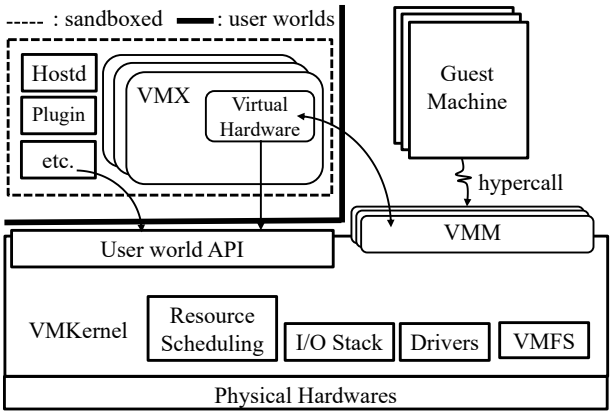

Figure 1: The architecture of *VMware ESXi* [5]. `VMKernel`, a POSIX-like OS, is designed to multiplex the virtual machines and provides some core fundamentals such as resource scheduling, I/O stacks, file system (`VMFS`), and drivers. The guest machines can communicate with the host through hypercall, which includes normal hypercalls such as `VM-Exit` and special hypercalls such as `backdoor` and `VMCI`. The term "user world" refers to a process running in the `VMkernel`. A significant user world is "`VMX`," a Ring-3 process, because it contains RPC handlers and virtual hardwares. (Note that the `VMX` process and some other process sandboxed by the vSphere sandbox.) The virtual machine monitor (VMM) is a process that provides the execution environment for guest virtual machines.

abilities used in the first virtual machine escape of *VMware ESXi*.

3. State-of-the-art and reusable exploitation techniques for manipulating memory layout, constructing an arbitrary-address-write primitive, and achieving persistent exploitation in *VMware ESXi*.

Threat Model. In this research, we assume that an adversary can execute arbitrary codes in the user space and kernel space on the guest OS.

2 Background

2.1 The Architecture of the ESXi

As Figure 1 illustrates, *VMware ESXi* integrates its operating system (OS) called VMkernel, providing the functionalities of resource scheduling, I/O stacks, network stacks, storage stacks, and device drivers, and all processes are running on top of it. VMkernel also implements a simple in-memory file system to hold staged patches, configurations, and system logs.

To communicate with the hypervisor, the guest taps into the `VMM` through `VM-Exit` in most circumstances. Notably, VMware also introduced another hypercall mechanism called `backdoor`. Interestingly, although it is named "`backdoor`," it is merely a communication channel between the guest and the

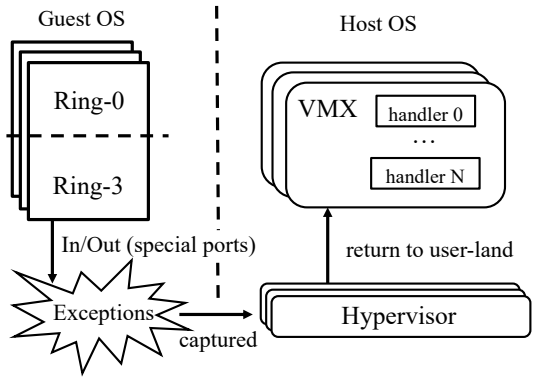

Figure 2: The backdoor remote procedure call. Under default I/O privilege level, a Ring-3 program should not be able to issue I/O operations. As a result, the `int` or `out` instruction should cause the Ring-3 process to fault and crash. However, in this scene, the hypervisor captures the fault and handles it, supporting the `backdoor` RPC mechanism.

hypervisor. Also, it has been widely applied to VMware products, e.g., some functionalities in `open-vm-tools` [18] such as drag-and-drop, copy-and-paste are implemented on top of the `backdoor` mechanism. As Figure 2 illustrates, the guest machine that runs in Ring-3 of a protected-mode OS executes the `in` or `out` instruction with a specific port, which raises a corresponding exception. Normally, this results in the crash of the process. However, in a VMware virtual machine, the hypervisor captures the exception and dispatches it to a proper handler on the host OS. Therefore, there are no exceptions in the end. Compared to normal hypercall, which usually requires the $\texttt{CPL} \leqslant \texttt{IOPL}$, some `backdoor` requests can be issued from Ring-3 directly. Consequently, a channel for the communication between guest and host can be established. For one example, as Figure 3 delineates, `backdoor` can be leveraged to send data from the guest to the host. By putting required parameters into specific registers, like a simple function call, a process running in the protected mode can invoke the RPC directly. In the sample, we first create a new "RPCI" channel and retrieve the channel number. Then, we can send data to the host through this channel. Based on this mechanism, some high-level and complicated protocols could be developed. Furthermore, in this paper, we also use this feature to reliably manipulate the memory layout. The details are discussed in §3.

2.2 Virtual Machine Escape

VM escape is a process of breaking out of a virtual machine from a guest OS, so the guest VM can launch an arbitrary execution with the privilege of the host operating system [21]. Specifically, *ESXi* completely isolates the guest operating systems from each other by leveraging hardware virtualization technologies, such as Intel VT or AMD-V. Any privileged instructions from the guest operating systems will be captured

```
; Creating a new channel
asm(
    "movl $0x564d5868,%%eax\n\t" ; magic bytes: 'VMXh'
    "movl $0xc9435052,%%ebx\n\t" ; magic bytes for RPCI
    "movl $0x1e,%%ecx\n\t"       ; MESSAGE_TYPE_OPEN
    "movl $0x5658,%%edx\n\t"     ; special I/O port
    "out %%eax,%%dx\n\t"
    "movl %%edx, %%eax\n\t" ; ret channel number (EDX(HI))
    "movl %%ecx, %%ebx\n\t"      ; success or failure
    ...
);

; Sending data through a specific channel
asm(
    "movl %0, %%edx\n\t"         ; channel number (EDX(HI))
    "movl $0x41414141,%%ebx\n\t" ; 4 bytes to be sent
    "movl $0x564d5868,%%eax\n\t" ; magic bytes: 'VMXh'
    "movl $0x0002001e,%%ecx\n\t" ; MESSAGE_TYPE_SEND
    "movw $0x5658,%%dx\n\t"      ; special I/O port
    "out %%eax,%%dx\n\t"
    "movl %%ecx, %%eax"          ; success or failure
    ...
);
```

Figure 3: By using "RPCI," a sort of backdoor-based mechanism, guest machines can communicate with the host OS. By reading or writing a special I/O port (0x5658/ 0x5659), a process running in Ring-3 can invoke the RPC directly.

and sanitized by the hypervisor. In normal circumstances, the guest cannot execute codes or affect security-critical behaviors such as system configuration, and network connections of other guests or the host. By exploiting the vulnerabilities in the *ESXi*, the adversaries can cross the security boundary between the guest and the host, to execute arbitrary codes on the host operating system (i.e., virtual machine escape in the *ESXi*).

2.3 Security Analysis of the ESXi

Attack Surfaces. The virtualization layer is the most significant part of the lifetime of the guest operating system. Any interaction between the guest OS and the hypervisor is a potential attack vector that could be exploited by adversaries. Generally, the guest can communicate with the host of *ESXi* in several ways:

1. VMKernel and Core Virtualization Infrastructures: There are some fundamentals such as `VM-Exit` handlers, memory management, and memory virtualization infrastructures offered by the hypervisor running in the kernel space. An adversary who attacks the hypervisor successfully could take over the kernel of the host operating system directly.

2. Virtual Hardware: To support the I/O virtualization, VMware designed a batch of virtual harware and devices. Most of them are integrated into the `VMX` process of *ESXi* [3, 15]. The guest OS can communicate with the virtual hardware through port I/O (`PIO`) or memory-mapped I/O (`MMIO`). Table 1 shows some significant virtual hardware integrated in *VMware ESXi*. Most of them require that the root privilege be opened in the guest operating system.

Interface	Category	Privilege
SVGA2D	Virtual Graphic	ROOT
SVGA3D	Virtual Graphic	ROOT
e1000	Virtual Ethernet	ROOT
e1000e	Virtual Ethernet	ROOT
VMXNET3	Virtual Ethernet	ROOT
xHCI	Virtual USB	ROOT
uHCI	Virtual USB	ROOT
aHCI	Virtual SATA	ROOT
Lsilogic	Virtual SCSI	ROOT
Printer	Virtual COM Device	ROOT

Table 1: The virtual hardware has been demonstrated to affect the interaction of guest-to-host. The privilege field indicates the requirement to open the device in the guest OS.

3. RPC Channels: VMware developed some RPC protocols such as `backdoor` and VMware Virtual Machine Communication Interface (`VMCI`) to accelerate the communication between the guest OS and the host OS. It has been applied in VMware's virtual machines for decades, because it does not rely on hardware virtualization extensions. Thus, it also results in some virtual machine escape attack surfaces. Some RPC handlers exist in the `VMX` process. By exploiting the bugs in these handlers, adversaries can escape from the guest OS.

Common Mitigations. The host maintains a POSIX-like operating system, and some Linux-like security mitigations are integrated into the host.

1. ASLR: Address Space Layout Randomization (ASLR) [20] was introduced to mitigate the exploitation of memory corruption vulnerabilities. In *ESXi*, the addresses encompassing the program, stack, heap, and libraries of the user space binaries are randomized. In the *ESXi*'s `VMX` process, which contains most of the virtual hardware, some hardware such as the network card runs in Ring-3. Therefore, an attacker who wants to attack virtual hardware or other Ring-3 services in *ESXi* first has to leak code pointers (i.e., information leakage) to further hijack the control flow of *ESXi*.

2. NX/ DEP: This option is referred to as Data Execution Prevention (DEP) or No-Execute (NX). It works with the processor to help prevent buffer overflow attacks by blocking code execution from memory that is marked as non-executable [11]. In *ESXi*, when the process is trying to execute shellcodes on stack, heap, or data segments, it will crash.

3. Compact VMX: Compared with *VMware Workstation*, the type-2 hypervisor developed by VMware, *ESXi*'s

```
# Rules applicable for all VMs

-s genericSys grant
-s ioctlSys grant
-s vsiReadSys grant
...

-c unix_socket_create grant
-c unix_stream_socket_bind grant
-c unix_dgram_socket_bind grant
...

-p inet_socket_bind all grant
-p inet_socket_connect loopback grant
-p inet_socket_connect nonloopback grant
...

-d tpm2emuObj tpm2emuDom file_exec grant

-r /var/run rw
-r /var/lock rw
...
```

Figure 4: A sample rule for global VMs. It grants what system calls a `VMX` process is allowed to call, what network connections a `VMX` process is allowed to establish, and what directories a `VMX` process is allowed to read or write.

```
void __usercall vmxnet3_reg_cmd(vmxnet3_class *a1,
__int64 read_or_write, _DWORD *data, __int64 a4, __int64 a5)
{
   ...
   case 4: // VMXNET3_CMD_UPDATE_MAC_FILTERS
     if ( a1->field_1A20 ) {
*      dma_memory_create(a1->driver_shared_addr + 8, 0x2B0ui64, 1,
*          a1->state->field_B8, &page);
       vmxnet3_cmd_update_mac_filters(v6, &page, a5);
*      destruct_page_struct(&page);
       sub_14017CB30(v6);
     }
     break;
   ...
 }

 char __fastcall dma_memory_create(unsigned __int64 addr, unsigned
   __int64 size, int a3, int a4, page_struct *page)
 {
   unsigned __int64 v5;

   v5 = *(qword_140DAA810 + 12160);
   // check the addr
*  if ( addr > v5 || !size || size > v5 - addr + 1 )
*    return 0;
   set_page_struct(addr, size, a3, a4, page);
   return 1;
 }
```

Figure 5: `dma_memory_create()` is responsible for creating a `page` struct used to read/write memory between guest and host. Inside the function, it checks the `addr` passed by users. If it is invalid, the function will return directly. However, it does not check whether the allocation is successful; thus, an uninitialized stack variable could be used in `destruct_page_struct()`.

`VMX` program has fewer source codes, because VMware moved the data package operations into the VMkernel, enhancing the efficiency of *ESXi*. Meanwhile, it narrows the attack surfaces of `VMX`.

Sandboxing. Since vSphere 6.5, *VMware ESXi* has introduced a mandatory access control (MAC), similar to Ubuntu's AppArmor, to enforce the security policy of guest VMs. (VMware vSphere is a commercial name for the whole VMware Suite, and *ESXi* is the hypervisor server of vSphere.) The sandbox maintains some pre-defined policies that contain some white lists of allowed syscalls, sockets, and file permissions in the file system (`VMFS`). The main functionality of sandboxing is offered by `VMKernel`, dividing the target into several different restricted domains (`app`, `ioFilter`, `pluginFramework`, `globalVM`, `plugin`, `tpm2emu`). After the specific virtual machine starts, Its behaviors are limited. It ensures the safety and security of VMs by running them in an operational sandbox with strict controls regarding hypervisor capabilities available to them [6]. Even if the adversaries have escaped from the guest operating system, they are still restricted by the vSphere sandbox; thus, it qualifies the impact of virtual machine escape attacks.

Figure 4 shows a sample rule for global virtual machines. For instance, if an adversary exploits a vulnerability in virtual hardware that is integrated in the `VMX` process of *VMware Workstation*, the adversary can spawn a shell or reverse shell on the host OS, finishing the VM escape. However, in *ESXi*, even if an adversary gets an arbitrary shellcode execution primitive by exploiting vulnerabilities in the `VMX` process, it cannot invoke any sensitive syscalls such as `execve` or establish a remote shell through sockets. Therefore, it makes the VM escape more difficult. The sandbox of *ESXi* is a rule-based sandbox, and each virtual machine runs in its own sandbox. However, it already has predefined rules; the users of *ESXi* do not need to configure it by themselves. The rules dictate what the `VMX` process running the virtual machine is allowed to do or access, e.g., the `VMX` process has no access to some sensitive files such as `/etc/passwd`, and the `VMX` process cannot run scripts or initial network connections on the host. Meanwhile, the sandbox restricts what system calls a `VMX` process allowed to call; thus, some sensitive system calls such as `execve`, `execl` are restricted. The complete rules of the sandbox can be found in `/etc/vmware/secpolicy/domains/` on the host. By auditing the sandbox profiles, we can get some plausible attack surfaces of the sandboxing in *ESXi*.

3 VM Escape: Our Approach

Overview. There are two uninitialized usages in vmxnet3 virtual ethernet card and a logical issue in the sandbox policy. CVE-2018-6982 is used for code pointer leak, and CVE-2018-6981 for arbitrary pointer free. Our technique is to chain both for an arbitrary write and, ultimately control-flow hijacking. Next, we bypass the vSphere sandbox and achieve virtual machine escape.

3.1 Vulnerabilities

The vmxnet3 adapter is the recommended network adapter to use as default in *VMware ESXi* because it offers the best throughput of all the adapter options. As such, it's likely in use on most virtual machines. In this section, we introduce two memory corruption bugs leveraged in our exploitation chain.

CVE-2018-6981. This bug is caused by an uninitialized use of stack memory in the vmxnet3 virtual ethernet card. As Figure 5 shows, there is an interface (`vmxnet3_reg_cmd()`) that can execute commands in the `MMIO` memory of vmxnet3. In the command `VMXNET3_CMD_UPDATE_MAC_FILTERS`, the `dma_memeory_create()` function creates a `page` structure used to read/write memory between guest and host. The `destruct_page_struct()` is responsible for releasing the memory of the `page` structure.

According to Figure 5, the `page` structure is allocated and initialized in the function `set_page_struct()`. At the beginning of the function, the `dma_memeory_create()` function checks the validity of the physical address given by the guest. Unfortunately, if the guest provides an invalid physical address, the function will return immediately. However, after the `dma_memeory_create()`, the `VMXNET3_CMD_UPDATE_MAC_FILTERS` handler fails to check whether the allocation of the `page` structure is successful, resulting in an uninitialized use in the `destruct_page_struct()` function.

Technically, this bug can also be turned into an information leakage bug. However, to improve the stability of the exploitation, we decided to chain a dependent information leakage bug into the exploitation.

CVE-2018-6982. This bug is also caused by an uninitialized stack variable in the memory of *ESXi*, and we utilized it to independently retrieve memory address information from the host. There is another command handler in `vmxnet3_reg_cmd()` called `vmxnet3_cmd_get_coalesce()`.

Figure 6 depicts the core logic of it. The `get_args()` function is used to retrieve some data from a memory region of the `VMX` process. The `sanity_check()` function qulifies that the `v19` must satisfy $\texttt{v19} \leqslant 16$. Also, the `write_back_to_guest()` function will write 16 bytes of data into the guest context. Unfortunately, only 8 bytes of them (`v20`) are initialized.

```
1  bool __fastcall vmxnet3_cmd_get_coalesce(__int64 a1, char a2)
2  {
3    v17 = 0;
4    v26 = __readfsqword(0x28u);
5    qmemcpy(&v25, (const void *)(*(_QWORD *)(a1 + 208) + 272LL),
6      0x100uLL);
7    if ( !(unsigned __int8)get_args(a1, &v18)
8      || (_DWORD)v18 != 1 //v18 is controllable
9      || !HIDWORD(v18)
10     || !v19
11 ★   || HIDWORD(v18) != 16 // constrait; but v18 is controllable
12     || !(unsigned __int8)sanity_check(v19, 16LL) ) {
13     return 0;
14   }
15   if ( !a2 ) { //a2: always zero
16     v14 = *(_QWORD *)(a1 + 208);
17 ★   v20 = 0xFA000000003LL;
18
19     // first 8-byte of v20 is initialized, but 16-byte is read
20     // (HIDWORD(v18) == 16)
21 ★   write_back_to_guest(v19, &v20, HIDWORD(v18), 0, *(_DWORD *)
22 ★     (v14 + 184));
23     return 1;
24   }
25   ...
26 }
```

Figure 6: A code snippet of `vmxnet3_cmd_get_coalesce()`. The `get_args()` function reads a memory region of the `VMX` process. Ultimately, `v18`, `v19` are controllable. The second parameter `write_back_to_guest` indicates the source buffer, and the third one indicates the size.

3.2 Exploitation

A significant challenge of exploiting uninitialized use bugs is how to control the uninitialized variable. In this section, we illustrate the entire process of turning the uninitialized use bug to arbitrary code execution and how we overcome the challenges.

1) Arbitrary address free primitive. As Figure 7 delineated, first, we leak some addresses to break `ASLR` through the information leakage bug. Next, in `destruct_page_struct()` function (Figure 8), the function frees a field of the uninitialized stack memory. Hence, after filling a pointer into the uninitialized memory, an arbitrary-address-free primitive could be constructed.

2) Arbitrary address write primitive. As Figure 11 illustrates, the metadata of Backdoor-RPC channel exists in the data segment. Therefore, we use this feature to construct an arbitrary-address-write primitive. First, we opened several Backdoor-RPC channels; thus, some metadata structures of the channel in the data segment will be activated. Second, we fake a glibc fast-bin chunk on it to do the *House of Spirit Attack*.

Specifically, after leaking the address of the data segment, we calculated the addresses of the metadata for the `backdoor`, and put them into the uninitialized stack memory using the function `handle_port_io()`. For example, in Figure 9, when the size of the data is less than `0x8000`, it will put all of the data into the stack. Next, we use the arbitrary-address-free primitive to free the fake fast-bin chunk.

House of Spirit Attack. Because *ESXi* uses a variant of glibc to maintain Ring-3's heap, we decide to fake a fast-bin chunk on the global metadata of Backdoor-RPC channels, i.e., *House of Spirit Attack* of glibc [1,13,22]. However, glibc has several integrity checks to mitigate memory corruption attacks. To bypass it, we need to construct the fake chunk and pick the size properly.

After investigating, as Figure 10 illustrates, we determine some constraints in the `free()` function of glibc that need to be satisfied:

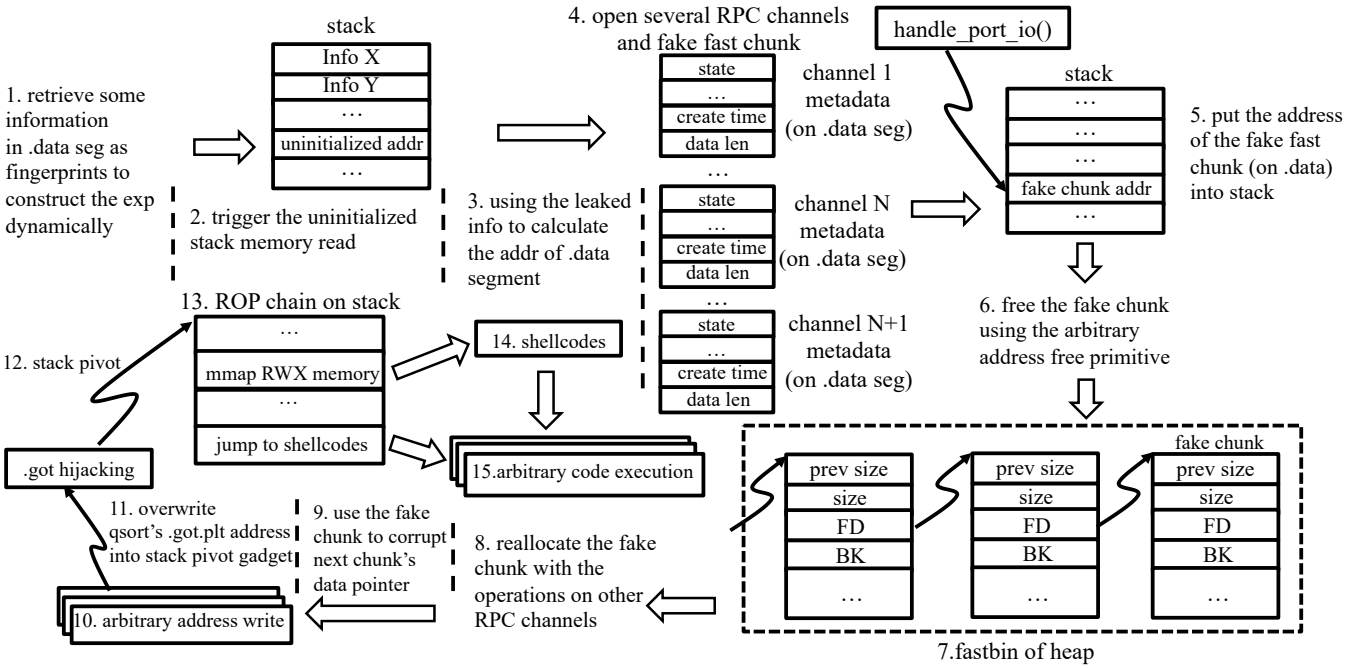

Figure 7: Exploiting the vulnerabilities and getting arbitrary shellcode execution privilege

```
void __fastcall destruct_page_struct(page_struct *a1)
{
  int v1; // eax
  page_struct *v2; // rbx
  unsigned int v3; // edi
  __int64 v4; // rbp
  __int64 v5; // rsi
  __int64 v6; // r12
  __int64 v7; // rax

  v1 = a1->ready;
  v2 = a1;
  if ( v1 == 1 )
  {...}
  else
  {
    v3 = 0;
    if ( v1 )
    {
      v4 = 0i64;
      v5 = 0i64;
      do
      {
        ...
      }
      while ( v3 < v2->ready );
    }
    free(v2->field_18); // free the pointer on stack
  }
}
```

Figure 8: A code snippet of `destruct_page_struct()`. We use it to free arbitrary addresses.

1. The `ISMMAP` bit of the fake chunk is `0`.
2. The fake chunk's address is aligned.
3. The size of the fake chunk is `32` bytes to `128` bytes and aligned.

```
void __usercall handle_port_io(__int64 a1, __int64 a2, __int64 a3)
{ ...
  char *v11; // rsi
  ...
  __int64 v35; // [rsp+A0h] [rbp-8038h]
  __int64 v36; // [rsp+80B0h] [rbp-28h]

  v3 = *(a1 + 4);
  v4 = *(a1 + 13);
  read_or_write = *(a1 + 48);
  ...
  if ( *(a1 + 60) && (v10 = *(a1 + 52) << 12, v10 > 0x8000) )
    v11 = malloc_heap_memory(v10); // copy the data into heap
  else
    v11 = &v35;
  if ( read_or_write & 1 )
  { if ( *(v8 + 60) )
    {   ...
        v15 = v11;
        do
        { ...
          memcpy(v15, v18, v17); // copy the data into stack
  ...
```

Figure 9: A code snippet of `handle_port_io()`. We use it to spray the stack.

4. For the next chunk's size θ: $2 * SIZE\_SZ \leqslant \theta \leqslant av \rightarrow system\_mem$.
5. The first chunk in the fast-bin is not the fake chunk.

Then, we reallocate the fake chunk by leveraging other Backdoor-RPC channel operations, i.e., when a new channel is opened, the channel allocates a new buffer with a controllable length that pointed by the `data` field in the metadata of the channel. This is a flexible and reusable trick to manipulate the heap of *ESXi*. Finally, we overwrite the next data pointer

```c
void public_fRE(Void_t* mem)
{
    mstate ar_ptr;
    mchunkptr p;
    ...
    p = mem2chunk(mem);
    if (chunk_is_mmapped(p)) // check mmap bit
    {
        munmap_chunk(p);
        return;
    }
    ...
    ar_ptr = arena_for_chunk(p);
    ...
    _int_free(ar_ptr, mem);
}

void _int_free(mstate av, Void_t* mem)
{
    mchunkptr p;
    INTERNAL_SIZE_T size;
    mfastbinptr* fb;
    ...
    p = mem2chunk(mem);
    size = chunksize(p);
    ...

    // check current size
    if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
    {
        // check next chunk
        if (chunk_at_offset(p, size)->size <= 2 * SIZE_SZ
            || __builtin_expect(chunksize(chunk_at_offset(p, size))
                >= av->system_mem, 0))

        {
            errstr = "free(): invalid next size (fast)";
            goto errout;
        }
        ...
        fb = &(av->fastbins[fastbin_index(size)]);
        ...

        p->fd = *fb;
        *fb = p;
    }
}
```

Figure 10: To fake a fast-bin chunk successfully, we need to bypass some constraints in glibc.

of the next channel and an arbitrary-address-write primitive can be constructed.

3) Code execution. To execute codes on the context of the host, we use the arbitrary-address-write primitive to corrupt the stack. First of all, we corrupt the global offset table and overwrite the `qsort` function into a stack pivot gadget. Next, a ROP chain will be put into the stack. Meanwhile, the control flow will be forwarded to the ROP chain on the pivoted stack. In particular, the ROP chain will invoke `mmap` syscall and enable the privilege to execute arbitrary shellcodes.

3.3 Circumventing the vSphere Sandbox

The logical bug in the sandbox policy. By scrutinizing the policies of the sandbox, we determine that there are some loopholes inside it. The sandbox grants the `VMX` process to read and write the `/var/run` directory of `VMFS`, and an internet server (`inetd`) configuration database exists in the

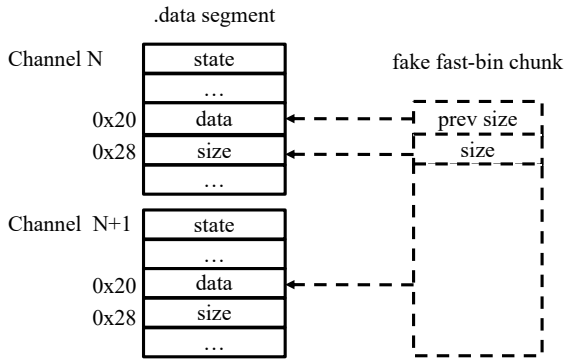

Figure 11: Arbitrary Address Write. By faking a fast-bin chunk on the metadata of Backdoor-RPC channels, we can reallocate the fake through Backdoor-RPC operations. After reallocating, we can overwrite the next channel's metadata to corrupt arbitrary addresses.

`/var/run/inetd.conf`.

In this way, we can bind a shell on a specific port by overwriting the `inetd.conf` file. Note that files existing in the `/var/run/*` are not persistent and copied from the backup firmware in the `/bootbank/*` directory after rebooting.

Forcing the process to restart. To activate the configuration and spawn a shell, we need to force the `inetd` process to restart. However, we cannot simply restart the entire OS, because the `inetd.conf` file is not persistent. Files in the `VMFS` are copied from the `bootbank` after the host OS restart. Fortunately, there is a watchdog can help us to restart some processes. As a result, we use the `kill()` system call to terminate the `inetd` process. After that, the watchdog restarts the process, and a bind shell spawns.

3.4 Reliability

Compatibility. To improve the compatibility of our exploitation, we use some fingerprints retrieved from the host OS to manipulate the exploitation dynamically. As Figure 7 indicates, we first utilize the information leakage bug to retrieve some memory of `.data` segment, using it to determine the version of the running *ESXi*.

Next, we can dynamically construct some crucial payloads (e.g., shellcodes, address of fake chunk, etc.) in terms of the current version.

Persistency. As depicted in §3.3, after the host OS restarted, the entire `VMFS` is overwritten again, i.e., files in `VMFS` are overwritten by the backup firmware stored in `/bootbank`. In light of this, after getting the root privilege, we can overwrite the files under the `/bootbank` directory to achieve persistent exploitation.

4 Implementation

The exploit has been implemented in a kernel module with 450 LoC. The shellcodes embedded in the kernel module will spawn a bind shell on a specific port and a reverse shell connecting to our remote server. To launch the escape, we merely need to insert the module into the kernel of the guest OS.

Notably, all of the bugs depicted above have been reported to the vendor and patched appropriately. Now the exploit-chain only affect the instances of *VMware ESXi* whose version is earlier than 6.7.0 with build number 10764712.

5 Evaluation

To examine the stability and compatibility of our exploit. We evaluate it on *ESXi* 6.7 with different versions.

Experimental Setup. We run the exploit on *ESXi* with different versions. The guest OS is Ubuntu 16.04.3 LTS, with 2 cores, and 4GB memory. The host machine has an i9-7980XE processor with a 64GB physical memory. For each guest OS, we run the exploit 30 times.

Results. Table 2 shows the results. The exploit chain has been demonstrated to affect ESXi 6.7, and ESXi 6.5. Also, the exploit can effectively adapt to the targeted environment. In particular, the maximum success rate of our proposed exploit chain is 93.33%.

Version	Build number	V?	S?	Success	Adaptable
ESXi 6.7	10764712	Y	Y	93.3%	Y
ESXi 6.7	10302608	Y	Y	86.7%	Y
ESXi 6.7	10176752	Y	Y	90.0%	Y
ESXi 6.7	9484548	Y	Y	86.7%	Y
ESXi 6.7	9214924	Y	Y	93.3%	Y
ESXi 6.7	8941472	Y	Y	90.0%	Y
ESXi 6.5	<10719125	Y	Y	N/A	N
ESXi 6.0	Any	N	N	N/A	N/A

V: Vulnerable or not, S: Sandboxed or not

Table 2: The results of the evaluation. The adaptable field indicates whether the exploit can adapt to the environment automatically. Actually, the exploitation chain can also work on *ESXi* 6.5, but we have not adapted those versions. Furthermore, the reason the success rate is not 100% is that the stack could be polluted by the *ESXi* itself, resulting in the arbitrary address free primitive fails.

6 Discussion

Evaluation results show that the success rate of our exploit is not 100%. In this section, we try to determine the reasons. By scrutinizing the memory status, we found two significant factors that qualified the success rate of the exploit.

1) Manipulating the uninitialized variables. In light of the fact that the arbitrary-address-free primitive requires valid addresses, we have to make sure the targeted address is legitimate before triggering the logic of free. However, the normal actions and executions in *ESXi* may pollute the stack, resulting in the “free" operation not going as expected.

2) The stability of heap. After releasing the targeted fake memory chunk, we need to reallocate it immediately. However, if *ESXi* allocates it before we do, the process will crash suddenly.

7 Related work

Virtual Machine Escape. Chaitin Security Research Lab has demonstrated an exploit of *VMware Workstation* with a single vulnerability in Backdoor-RPC [4]. However, in their tests, the stability is qualified by the Low Fragment Heap of the Windows 10 operating system, and the maximum success rate is 80%. Keen Security Lab has also demonstrated an exploit for *VMware Workstation* by using an information leakage bug in Backdoor-RPC and a memory corruption bug in xHCI [9]. For type-1 hypervisors, CVE-2015-7835 [2] has been demonstrated to achieve a virtual machine escape of the *Xen* hypervisor. Furthermore, Jordan Rabet has proposed an in-depth analysis of a successful virtual machine escape of *Hyper-V* [12].

Exploitation of Uninitialized Use. The leading part of the exploitation of uninitialized use is to control the uninitialized variables and turn them into other types vulnerabilities. Halvar Flake proposed an approach to determine all the paths that could be overlapped in the same stack frame [7]. Also, Kangjie Lu et al. proposed an automated approach using targeted stack spraying to facilitate the uninitialized uses of the kernel [10].

Mitigations for VM Escape. BitVisor [14] tried to enforce the security of I/O devices by minimizing the code size of hypervisors by allowing most of the I/O accesses from the guest OS to pass through the hypervisor. NoHype [16] presented a strategy to eliminate the hypervisor attack surface by enabling the guest VMs to run natively on the underlying hardware while maintaining the ability to run multiple VMs concurrently. Hypersafe [19] proposed an approach to apply the Control-Flow-Integrity to mitigate virtual machine escape attacks. Cloudvisor [23] introduced an approach that enforces the separation of resource management from security protection in the virtualization layer.

8 Conclusion

VMware ESXi is one of the most state-of-the-art enterprise class hypervisors. However, there has been no systematic security analysis or successful virtual machine escape until this research. We give a systematic overview of the architecture, attack surfaces, and exploitation approaches of *ESXi*. Furthermore, we proposed a flexible and reuseable strategy that leverages the backdoor RPC to manipulate the memory layouts. Our exploitation chain contains three vulnerabilities. Evaluation results show that it is reliable (90% success rate on average).

Acknowledgment

We would like to thank Xiaoshuai Zhang and Hong Hu for helpful suggestions in the paper writing, and anonymous reviewers for their helpful comments. We also would like to thank VMware for quick response.

References

[1] blackngel. MALLOC DES-MALEFICARUM, 2009. http://phrack.org/issues/66/10.html.

[2] Jeremie Boutoille. Xen exploitation part 2: XSA-148, from guest to host, 2016. https://bit.ly/2MaCGB4.

[3] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4):12, 2012.

[4] Amat Cama and Kun Yang. The Weak Bug - Exploiting a Heap Overflow in VMware, 2017. https://bit.ly/2uPYK6A.

[5] Charu Chaubal. The Architecture of VMware ESXi. *VMware White Paper*, 1(7), 2008.

[6] Adam Eckerle, Mike Foley, Eric Gray, Matthew Meyer, Kyle Ruddy, and Emad Younis. What's New in VMware vSphere® 6.5, 2016. https://bit.ly/2mwhSGV.

[7] Halvar Flake. Attacks on Uninitialized Local Variables, 2006. https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf.

[8] Jason Geffner. VENOM: Virtualized ENVIRONMENT NEGLECTED OPERATIONS MANIPULATION, 2015. https://venom.crowdstrike.com/.

[9] Marco Grassi, Azure Yang, and Jackyxty. A bunch of Red Pills: VMware Escapes, 2018. https://keenlab.tencent.com/en/2018/04/23/A-bunch-of-Red-Pills-VMware-Escapes/.

[10] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[11] Microsoft. DEP/NX Protection, 2018. https://docs.microsoft.com/en-us/windows/desktop/win7appqual/dep-nx-protection.

[12] Jordan Rabet. Hardening hyper-v through offensive security research, 2018. https://ubm.io/2WhwVW5.

[13] Shellphish. how2heap, 2019. https://github.com/shellphish/how2heap.

[14] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. Bitvisor: A thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 121–130, New York, NY, USA, 2009. ACM.

[15] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2001.

[16] Jakub Szefer, Eric Keller, Ruby B Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 401–412. ACM, 2011.

[17] Mehdi Talbi and Paul Fariello. QEMU Case Study, 2017. http://www.phrack.org/papers/vm-escape-qemu-case-study.html.

[18] VMware. Open-VM-Tools, 2019. https://github.com/vmware/open-vm-tools.

[19] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395. IEEE, 2010.

[20] Wikipedia. Address space layout randomization, 2019. https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[21] Wikipedia. Virtual Machine Escape, 2019. https://bit.ly/2WoFazv.

[22] Tianyi Xie, Yuanyuan Zhang, Juanru Li, Hui Liu, and Dawu Gu. New exploit methods against ptmalloc of glibc. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 646–653. IEEE, 2016.

[23] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.