Subverting the Xen hypervisor

Rafal Wojtczuk
rafal.wojtczuk@invisiblethingslab.com

August 7, 2008

***Abstract*— This paper outlines the recent work by the author to design and develop a backdoor for machines running the Xen hypervisor. An attacker can gain backdoor control over the host by overwriting Xen code and data structures; as not a single byte in dom0 domain is modified, the detection of such a backdoor is difficult if conducted from within dom0.**
It is shown that it is feasible to modify device drivers and core kernel code to conveniently conduct DMA to arbitrary physical address, which allows for control over the hypervisor. Two backdoors have been implemented: one resides in the hypervisor code, the other resides in a hidden domain with artificially elevated privileges.

***Index Terms*— computer security, Xen hypervisor, backdoor, DMA.**

I. Introduction

BLUEPILL [1] and Vitriol [2] are well-known projects that install a malicious hypervisor in run-time. Initially, there is no hypervisor present on the target machine; these projects are capable of inserting a malicious hypervisor on the fly, without reboot, and since then exercise control over the underlying OS.

Many analysts predict that in near future, many systems will run some sort of hypervisor by default. This presents both challenges and opportunities for attackers.

If the legal hypervisor is compromised and its code modified, this will give attacker similar capabilities to the ones available to Bluepill and Vitriol. It would be no longer necessary to hide the mere presence of hypervisor by masking some side effects of a malicious hypervisor, as the (legal) hypervisor is expected to be present and these side effects can be attributed to it. Also, significantly less code must be created, as a lot of necessary functionality is implemented in the legal hypervisor.

In some architectures, the legal hypervisor can protect itself against runtime modification, even if the attacker has all privileges in the underlying OS. For instance, in case of Xen, there is no supported method to modify Xen code in runtime by the administrator of the dom0 domain.

In this paper we will discuss how the Xen 3.x hypervisor can be modified in runtime with DMA transfers so that backdoor functionality can be inserted. The generic framework allowing to load compiled C code into Xen will be described. Two backdoors using this framework will be evaluated. We will also discuss possible detection methods.

II. A few important fact about Xen 3.x architecture

The Xen 3.x architecture is described in the documentation bundled with the Xen source code[3]. The reader is encouraged to get acquainted with these documents; in this section we will only mention the details directly referenced later in this paper. Also, throughout this paper, we cover only x86_32 and x86_64 architectures, and Linux as dom0's OS.

The Xen hypervisor is directly loaded by the bootloader; unlike e.g. VMware Workstation [4] it does not require a full OS to be active in order to start. It is the only piece of code running in ring 0. After its initialization, it starts the first VM named dom0. The kernel of this VM (as well as all other domains) runs in less privileged ring (1 in case of x86_32, 3 in case of x86_64), so it is under full control of the hypervisor. Dom0 operating system (usually Linux, although NetBSD and OpenSolaris are also supported) is special: unlike all other domains, it is allowed to request Xen to conduct administrative actions, like creating, starting and stopping other VMs.

Moreover, dom0 is allowed to access most of the hardware directly. It exports restricted set of resources (e.g. disk space) to other unprivileged domains. This design allows to use mostly unmodified device drivers written for dom0 OS, instead of porting all of them to Xen.

After dom0 has booted, it can request Xen to start other VMs. There are two types of VMs:

- Fully virtualized. Requires processor support for virtualization (Intel VTx or AMD AMD-V). Unmodified guest OS can run in the VM. All operations that are privileged or depend on the privileged state are intercepted by Xen and simulated to provide illusion of full control over the machine.
- Paravirtualized. The guest OS must be aware that it runs under Xen's control, and instead of performing privileged operations directly, it requests Xen to conduct them. Particularly, dom0 is a paravirtualized domain.

The Xen services can be requested by the guest domain by invoking hypercalls. Xen sets up the IDT entry for the software interrupt 0x82 so that guest's kernels can issue int 0x82 instruction, and then the control is passed to a Xen function (hypercall) selected by the value of the eax/rax register. A hypercall validates the arguments provided by the guest and if the required action does not violate security policies, it is executed. Example hypercalls are do_mmu_update and do_set_gdt, whose purpose should be self-explainable.

III. Getting control over Xen with DMA

A. Introduction

Let's assume that an attacker gained root privileges in dom0. It can be an effect of improperly secured access to a vulnerable network service running in dom0. Also, in the past there were known vulnerabilities in code running in dom0 that can be

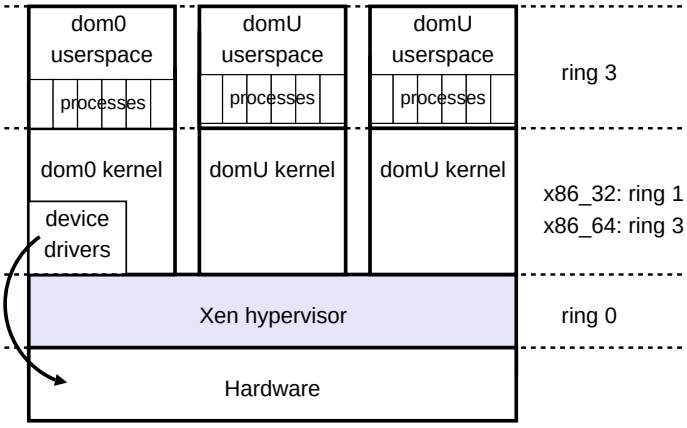


Fig. 1. The simplified Xen architecture

exploited from within other unprivileged domain; see CVE-2007-4993 [5] or CVE-2007-5497[6] for examples.

If the attacker wishes to plant a backdoor in the Xen code, he will need the ability to alter Xen memory. He can modify the bootloader so that after reboot, a backdoored Xen version will start; however this approach has disadvantages:

- The reboot of the hypervisor is a very noticeable event, that may reveal the intrusion.
- The ability to alter the boot process can be limited; possible means include booting from media write-protected by hardware, or more sophisticated techniques like Intel Trusted Execution Technology[7].

Therefore, the attacker needs to alter Xen memory in runtime. It is a well-known fact (particularly, explicitly mentioned in the Xen documentation) that without means to restrict DMA transfers (via IOMMU[8] or VT-d[9]; most chipsets do not support these today) any DMA capable device can overwrite arbitrary physical memory, including regions occupied by Xen code.

Let's note that the abilities of DMA-capable devices pose problems for other systems that separate device drivers from the rest of the privileged code, e.g. ones based on microkernel design (like Minix3[10]).

Luic Duflot in his paper [11] discusses similar ideas. He showed that AGP GART or USB subsystem can be configured to overwrite arbitrary physical memory. However, we will take a slightly different approach. Instead of using particular devices, which are easy to program[1], but can be disabled by the administrator (or not present at all), we will use popular hardware: a network card or a hard drive. Moreover, we will not dwell into all details on how a given device is programmed to schedule a DMA transfer; we will let the code of the device driver (and kernel support code) do most of the work. As we will see, in the HDD case, we will achieve portability over any make of device.

There is one particular issue to be considered: the device that we want to program for arbitrary DMA access can be used by the OS. We have to act so that we do not disrupt normal operation of the device. Particularly, unloading of the existing driver and initializing device from a scratch is not an option.

There is another paper by Luic Duflot [12] that discusses how to abuse System Management Mode of a x86 processor to gain full control over the physical memory. However, as all modern BIOSes protect the SMM code from modification by OS, this type of attack is unlikely to succeed[2], so we will not discuss it here.

B. A network card example

The network card has a nice property: as generally the network medium is unreliable, we can afford to temporarily drop all packets sent to the card by the OS; their loss will be attributed to the medium. Naturally, we must not affect higher layer structures, as the ip routes.

It may seem that NIC is not suitable for our task: it uses DMA to copy data between RAM and its internal buffers, and the latter are not accessible directly. Therefore, all we can achieve is to copy the contents of arbitrary memory location to (or from) the network medium, and the latter is not easily controllable. However, almost all NICs support "loopback mode". In this mode, the transmit operation does not copy a packet to the wire, but to internal buffers, and then the packet is received back. That said, we can achieve arbitrary physical memory access at address X. First, we stop the packet queue (with netif_tx_disable() kernel function). Then we set the loopback mode and promiscuous mode (so that there is no check on the MAC header residing in the received data). Then:

- reading: set a transmit ring entry so that the data pointer points to X, and the receive ring entry data pointer points to buffer we can read
- writing: set a transmit ring entry so that the data pointer points to our data, and the receive ring entry data pointer points to X

The good thing is that the implementation of the above procedure is simple. On Linux, we can write a kernel module that will get the address of the relevant "struct net_device" structure with dev_get_by_name() macro, and then we can operate on the card state. For instance, for writing, the line in tg3.c driver:

```
tg3_set_txd(tp, tp->tx_prod, map,
    tx_len, 0, 1);
```

will be changed in our module to

```
tg3_set_txd(tp, tp->tx_prod, arbitrary_phys_addr,
    tx_len, 0, 1);
```

Still, as the functionality to set the loopback mode is not exported by Linux drivers[3], for each card make we will have to create appropriate code. The author has successfully implemented tg3dma.c module, that works with all cards supported by the Linux tg3.c driver.

[1]Luic uses PIO only to configure the devices; we can do more, e.g. access the relevant iomem

[2]Naturally, assuming there are no bugs in the above protection

[3]But it is often implemented for the needs of the selftest functions, accessible by ethtool

C. A hard drive controller example

In order to set up a DMA transfer, a device driver needs to obtain the physical address of a buffer passed by upper layers. In case of HDD drivers, it turns out that the function dma_map_sg() is used. On some architectures this function is in fact a simple macro; in case of Xen 3.x dom0 Linux kernels, it is a real function implemented and exported by the kernel.

The basic idea is: we will hook the dma_map_sg() function, so that it will report an attacker-chosen physical address X (instead of a real physical address of a buffer). Then, $write(somefd, userbuffer, len)$ syscall will result in DMA transfer to disk not from the $userbuffer$, buf from address X.

The challenge is that we cannot blindly redirect all transfers to X. Other processes/kernel threads may issue disk access, and if we redirect it to X, memory or filesystem corruption will occur. So, we have to know whether a given dma_map_sg() invocation is in progress in order to get the physical address of our $userbuffer$.

The dma_map_sg() prototype is:

```
int dma_map_sg(struct device *dev,
        struct scatterlist *sg, int nents,
        enum dma_data_direction direction);

struct scatterlist {
  struct page *page;
  unsigned int offset;
  dma_addr_t dma_address;
  unsigned int length;
};
```

dma_map_sg() is supposed to set the $dma\_address$ field with the physical address of the $page$ field.

We could expect that if $write(somefd, userbuffer, len)$ syscall was invoked, then the $page$ field will correspond to the page occupied by $userbuffer$. Unfortunately, it is not so; instead, the $page$ field will point to the page structure used by the filesystem cache. It is not easy to determine which filesystem cache page will be used for our transfer. Therefore, we need to bypass filesystem cache, by setting the O_DIRECT flag in open() syscall. Then, the hook for dma_map_sg() will look like this:

```
call_original_dma_map_sg();
for (i = 0; i < nents; i++)
  if (sg[i].page == page_of_userbuffer)
    sg[i].dma_address = our_phys_addr;
```

The author has successfully implemented directhdd.c kernel module that allows for arbitrary memory access with the help of any HDD controller. Again, the required amount of code was small (c.a. 200 lines of C).

IV. Xen loadable module framework

A. Introduction

As shown in the previous section, it is relatively easy to overwrite the physical memory occupied by Xen in a controllable manner. Particularly, we can read a given page and then apply modifications to it. In this section we will describe the framework allowing to load compiled C code into Xen, similarly to Linux loadable kernel modules.

B. Locating important code and data structures

The bootloader loads Xen at physical address 0x00100000. In case of Xen 3.2.0 and later on x86_64 architecture, the Xen boot code relocates the hypervisor to the top of physical RAM. Hypervisor's pages are mapped linearly at high virtual addresses: on x86_32 (with PAE) starting at 0xff100000, on x86_64 starting at 0xffffffff80200000. Thus, if we know the virtual address of a variable/function, we can easily find its physical address.

The $hypercall\_table$ array stores all hypercall addresses; int 0x82 handler invokes a hypercall by $call * hypercall\_table(, eax, 4)$ (and similarly in case of x86_64). Similarly, all exceptions are handled by $handle\_exception$ function; it calls appropriate function by $call * exception\_table(, eax, 4)$ (and similarly in case of x86_64). If we have access to the $xen - syms$ binary, we can easily retrieve the addresses of these tables from this file. Otherwise, we can locate the $hypercall\_args\_table$ array by pattern matching (because this array has fixed content, starting with "\x01 \x04 \x02 \x02 \x04 \x01 \x02 \x01 "), and as this array is directly preceded by $hypercall\_table$ and $exception\_table$, we can discover their addresses as well.

We will also need a few more addresses of Xen functions; if the $xen - syms$ binary is not available, they can be located as follows:

- $xmalloc$ and $printk$. The alloc_xenoprof_struct() function contains the following fragment:

  ```
  static int alloc_xenoprof_struct(...)
  {
    ...
    d->xenoprof = xmalloc(struct xenoprof);
    if (d->xenoprof == NULL) {
      printk("alloc_xenoprof_struct():  "
          "memory allocation failed \n");
      return -ENOMEM;
    }
    ...
  }
  ```

  We can find the error message in the Xen binary; then by finding the only reference to it we will discover $printk$ address; and by finding the reference to the basic block containing the above $printk$ invocation we can find the address of $xmalloc$ function.
- $copy\_from\_user$. A few hypercalls, e.g. $do\_physdev\_op\_compat$, contain $copy\_from\_guest$ macro at its beginning. This macro contains calls to $copy\_from\_user$ and $copy\_from\_user\_hvm$; it is easy to distinguish these two.

Many other important data structures are referenced from the current_vcpu structure; this structure resides at the bottom of

the ring0 stack, so it can be found in runtime (the $current$ macro).

C. Code execution at ring 0

In order to execute code at ring 0, we can use the following procedure:

- Using DMA, overwrite the body of $do\_ni\_nypercall$ (hypercall no 11; it consists of "return -ENOSYS" instruction and is normally unused) with "call 4(%esp)+ret" or "call %rdi+ret". We need to do it just once.
- Invoke hypercall 11, giving as its first parameter the address of arbitrary function (that can reside in guest space).

In the framework, there is a function $run0(void * fun)$ implementing the above.

D. Module loading and unloading

The module to be loaded is a normal ELF relocatable object; one may need correct Xen includes to compile it. The xenload utility performs the following actions:

- link the module with the xenlib.o file containing addresses of functions and structures mentioned in the previous subsection (so that the module can reference them)
- get the size S of the resultant static ELF
- using run0 function, allocate S bytes in the Xen heap by calling the Xen xmalloc function; let it return address A
- relink the module, this time using the linker script that specifies base A;
- copy the linked executable to address X (by calling memcpy function via run0); as the linker script specifies "impure" section layout, we can just copy the executable contents and the sections will be positioned correctly
- parse the "varname=value" pairs passed in the xenload command line; initialize appropriate locations in the loaded module; fill .bss section with zeroes
- retrieve the address of the "init_module" function from the linked module
- call the "init_module" function via run0.

The xenunload utility just calls the "cleanup_module" function in the module via the run0 primitive; for simplicity, the allocated memory is not freed.

A sample module source code is included in appendix B.

V. Debug register backdoor

A. Goals and assumptions

Armed with the tools described in the previous section, we are ready to start designing a Xen-based backdoor. It is supposed to grant remote access to dom0. The main property is the lack of modification of the dom0 code and data structures, so that a memory scanner running in dom0 cannot detect the backdoor presence.

There are two aspects of backdoor stealthiness:

- How difficult it is to detect the presence of the backdoor while it is not being used
- How difficult it is to detect the presence of the backdoor while it is being used

In this section, we focus on the first issue. When activated, the backdoor will provide arbitrary shell commands execution capability; there is no attempt to hide these commands. We will briefly discuss the second issue later.

The basic design of the debug register backdoor (or, dr backdoor)[4] is:

- When dom0 is active, Xen configures debug registers DR3 and DR7 so that a call to the netif_rx() function (called by network interface driver to pass a received packet to the network stack) will result in the debug exception being raised
- The Xen debug exception handler is replaced by the backdoor. When the handler determines that is was called because of exception at address netif_rx(), it will inspect the packet payload. If the packet contains appropriate magic bytes, shell will be executed in dom0 with the arguments taken from the packet payload.
- Anytime when dom0 queries or sets debugging registers, it cannot find out that they are used for the backdoor purposes.

There is one important feature of this design. The fault to Xen space and packet inspection by the backdoor happens very low in the networking stack, just after the device driver has retrieved the packet. Particularly, Linux firewall code has not been run yet, therefore the packet will be inspected even if all traffic is dropped by the firewall. Similarly, the backdoor can force dropping of the packet (by returning control not to netif_rx, but to kfree_skb), so the network monitoring tool running in dom0 userspace will not even see the packet.

We will continue with more detailed description of a few issues.

B. Debug registers handling

For each VM, Xen keeps its state, to be able to restore it during interVM context switch. Particularly, the $arch.guest\_context.debugreg$ array contains the values of the debug registers. Normally, when a guest requests Xen to set a debugging register, this array is updated along with setting the register.

Dr backdoor hooks[5] do_set_debugreg() hypercall, invoked when the guest requests Xen to set a debugging register. When dr backdoor sees (in the hooked hypercall) that the guest attempts to set DR3 register (used by the backdoor), it will allow the guest to do it (and become deactivated). Similarly, when the backdoor sees that the guest sets the "active" mask for DR3 in DR7 to 0, it will regain control over DR3 (and relevant DR7 bits) thus becoming active again.

Dr backdoor hooks $arch.schedule\_tail$ function, thus it enables DR3 for its purposes[6] anytime the dom0 is selected to run by the Xen scheduler.

The do_get_debugreg() hypercall needs not to be hooked; it returns the value from the $arch.guest\_context.debugreg$

[4]Implementing a backdoor with debug registers in case of a usual OS is discussed in a few papers, e.g. phrack article p65-8[13]

[5]It is not enough to change this hypercall's address in hypercall_table, because this function is also called directly by Xen in GPF handler.

[6]checking $arch.guest\_context.debugreg$ first to make sure that the guest does not use DR3

array, and the content of it is always coherent with what the guest expect.

C. Implementing code execution in dom0

When dr backdoor decides that it has seen a command packet, it is supposed to create a new shell process in dom0. Theoretically, as the hypervisor has full control over the dom0 address space, it could "manually" tweak all dom0 kernel data structures so that a new process would be created. However, creating a new process is a complicated action, and implementing it in the backdoor would be difficult and unreliable.

A different approach was chosen. Before loading dr_backdoor, we will allocate (via vmalloc_exec) a "scratch" page in the dom0 address space. Most of the time this page will contain only zeroes. When dr backdoor sees the command packet, it will copy trampoline code to the scratch page, and instead of returning control to the netif_rx function, it will set dom0's EIP to the scratch page. After executing the shell, the trampoline code in the scratch page will self-destruct by returning into $memset$, so that no trace of backdoor code is left in dom0 address space.

Thus we shifted the burden of executing the shell from the hypervisor to the dom0 kernel. Still, it is not an easy task, particularly because the trampoline code runs in the interrupt context. The trampoline code must do the following:

- defer its execution until interrupt is completed by calling execute_in_process_context (a function exported by the kernel)
- call one of call_usermodehelper family functions to actually fork a shell. Most of these functions are macros; call_usermodehelper_keys is the one exported by the kernel.

D. Deficiencies of dr backdoor

The idle dr backdoor cannot be detected by scanning dom0 memory. However, there are different methods.
Probably, the handling of debug registers in not entirely transparent. There may be subtle differences to the clean systems in corner cases (e.g if dom0 sets both DR0 and DR3 to the netif_rx(), will the DR6 content be correct in the dom0 debug exception handler ?). They are possible to fix, but there is a more severe problem. A simple timing analysis will reveal that the first instruction of netif_rx() takes much more time to execute[7], due to overhead imposed by the debug exception handler. The baseline for timing analysis is easily obtainable.

Additionally, the Linux kernel assigns debug registers during the context switch in a "lazy" way. As a result, if a process sets debug registers and then exits, they will not be reset. All processes and kernel code will run with nonzero DR7, until a debug exception is generated - only then it will be zeroed. Dr backdoor will perceive DR3 as used and it will not re-enable itself.

[7]Tests showed overhead around two orders of magnitude

VI. Foreign backdoor

A. Introduction

Dr backdoor can be detected with timing analysis. It is an effect of its design: it hooks some function in dom0, and the associated overhead can be measured. In order to overcome this problem, we must use other means of dom0 state inspection than hooking.

Xen provides API for a domain (which must be privileged) to inspect other domain's memory. We will start a separate domain (let's name it "lurker") and by setting its "is_privileged" flag in Xen structures, we will empower it to inspect the memory of all other domains. Then, we will need a method to ship a "magic" pattern to some location in the domain memory. This pattern need to stay in the memory long enough so that VM context switch can happen and our lurker domain has a chance to run and spot this pattern. Then the lurker domain can force arbitrary code execution in dom0 (possibly with the help of the hypervisor).

The author implemented the code capable of detecting that a sshd process in dom0 domain has received a protocol identification string with a "magic" content. Then the lurker domain changes the sshd process stack (and saved registers in the kernel stack) so that it executes arbitrary shellcode.
This approach has a significant drawback. It is a good practice to firewall dom0 so that its network services can be reached only from trusted hosts. It would prevent the above scenario from happening, as we won't be able to contact the sshd daemon in the dom0 at all. However, if there is any other domain whose sshd can be reached, we can apply this approach as well; we will get, say, a remote command shell in an unprivileged domain, but then we can ask the hypervisor (using the run0 primitive, that can be used from an unprivileged domain as well) to conduct arbitrary action on our behalf, including spawning shell commands in dom0.

B. Some Xen-specific implementation details

Foreign backdoor is implemented as a initrd image for a particular kernel version. When started, its init process:

- patches the kernel to disable the code that checks whether the kernel runs in the initial domain. This code is clearly redundant; the hypervisor enforces appropriate security checks on its own.
- using the run0 primitive, it sets the "is_privileged" field in its domain
- in the loop, starts scanning the dom0 processes list in order to enumerate sshd processes
- for each sshd process, it inspects its state

In order to conveniently monitor memory in dom0, we need a method to map a page by its virtual address. Libxenctrl library provides API to

- retrieve the foreign domain's cr3 register
- map a frame by its physical address
- using the above two primitives, walk the pagetables in order to resolve a virtual address

One important detail is that if we want to inspect userland memory (e.g. in sshd process), and this process is not scheduled to run at the moment, then we don't want the current

cr3 value, but the process' Page Global Directory, stored in $task->mm->pgd$.
The above functionality to handle virtual addresses is the purpose of the xenaccess[14] library; however, xenaccess does not work with some kernel/hypervisor version pairs, therefore foreign backdoor uses the libxenctrl library only.

Once we have the ability to manipulate the memory of sshd process, we need to perform the following tasks:

- retrieve client's identification string (before the terminating newline is sent)
- if the magic string has been sent, insert and execute shellcode

The challenge is to do it without any information on the sshd binary version. As we will see, these tasks are not related to hypervisor topic; yet they are technically interesting, therefore they are described in the following subsection.

C. sshd transmutation

The code responsible for client identification string receiving looks the same in all opensshd versions (in function sshd_exchange_identification()):

```
/* the "buf" array is on the stack */
for (i = 0; i < sizeof(buf) - 1; i++) {
        if (atomicio(read, sock_in, &buf[i], 1) != 1) {
                /* log error */
                cleanup_exit(255);
        }
        /* handle \r */
        if (buf[i] == '\n') {
                buf[i] = 0;
                break;
        }
}
```

So, the identification string is read by one byte with the read syscall, and stored on the stack. Therefore, in order to observe that the whole magic string has been sent, the following check (specific for x86_32) that relies on the fact that the process sleeps in the $read$ syscall, will suffice:

- walk the processes list; recognize the sshd processes by name (the $comm$ field)
- for each sshd process, inspect its kernel stack (pointed to by $task->thread.esp0$); locate the system_call frame by searching for the word on the stack equal $syscall\_call + 7$ [8]. Then we know the location of saved registers.
- if the orig_eax is not the number of "read" syscall, exit
- if the saved ebx does not look like a file descriptor number, or the saved ecx does not point to the stack, or saved edx is not 1, exit
- let N be the length of the magic string; copy N bytes from userspace from address saved_ecx-N to a temporary buffer; check whether this buffer contains the magic string

The diagram in appendix A shows the relation between all the referenced data structures.

The last question remains: how to insert shellcode. A few straightforward approaches will not work against a recent Linux distribution (like Fedora):

- copy shellcode to the process stack or heap, set the saved eip to point to this location: neither the stack nor heap is executable
- copy shellcode to one of the memory regions occupied by a mapped executable: as the binaries are mapped as shared, we would damage all sshd processes
- setup return-into-libc payload on the stack: besides knowing the libraries version, we would have to determine their actual base (they are randomized)
- copy the shellcode to the VDSO region: this has space limitations, and also the change would be visible in all processes (VDSO is mapped as shared)

One solution would be to change the process's page tables directly so that the stack would be executable. Currently, a different approach was taken; it can be useful in other scenarios as well. The main idea is to execute sys_mprotect on one of the stack pages. We don't want to look for this function in libc, so we will call it via int $0x80 in VDSO (function $\_\_kernel\_vsyscall$). In order to do this, we need to set eax to $NR\_mprotect$. But we cannot achieve this by altering the kernel stack; eax is always overwritten with the syscall return value. Therefore, we will transit through sys_sigreturn (function $\_\_kernel\_sigreturn$ in VDSO) in order to get full control over all registers[9]. Full procedure:

- change the saved eip so that it points to sys_sigreturn code in VDSO[10]
- setup the sigreturn frame. The registers in the sigreturn frame should be set so that sys_mprotect will be executed (particularly eip=$\_\_kernel\_vsyscall$, eax=NR_mprotect) after sys_sigreturn, setting a part of the stack as executable
- copy shellcode to the stack region marked as executable; the top stack word should be set to point to the shellcode, so that after the "ret" from sys_mprotect, the control will be transferred to the shellcode.

D. Detection and possible improvements

The lurker domain can be well hidden. As showed in appendix B, it is easy to hook domctl hypercall so that this domain will not show in the "xm list" output. Probably xenstore tree should be modified properly as well.

If the lurker domain runs the scan rarely (say, once a second), it will consume negligible (hard to notice) amount of CPU time. Still, some memory must be allocated for this

[8]The system_call function (from arch/i386/kernel/entry.S) does roughly:
ENTRY(system_call)
push all registers on the stack (arguments to the system call)
syscall_call:
call *sys_call_table(,%eax,4)
Surprisingly, the syscall_call address is exported in kallsyms.

[9]It would be enough to transit through "popl %eax+ret" sequence residing in an executable region; but in case of ASLR, it is nontrivial to even scan for such pattern

[10]we know VDSO address: saved eip points after the int $0x80 in VDSO

domain; it can be just a few megabytes, yet it could be noticeable.

Instead of running the scan in the separate domain, we can run it within the hypervisor, by hooking some periodically invoked function (e.g. in scheduler). Then there would be no need to hide a lurker domain properly[11]. Also, in order to avoid the necessity to map user pages, it would be better to look at the kernel data structures only. The IP fragments queue is well-suited for this purpose. In case of Linux, an IP fragment is stored in the queue for 30 seconds, which gives us plenty of time to conduct the scan. If the magic string is detected in an IP fragment, arbitrary commands can be forced to be executed, as shown in the dr backdoor. Also, as the source of an IP fragment can be spoofed, this can help bypass firewall rules in some cases. Unfortunately, there are some implementation problems:

- the IP fragments hashtable (ipq_hash) is not exported in kallsyms in most cases; if System.map is not available, some work must be done to determine its location
- there are differences in IP fragments hashtable implementations between linux-2.6.24 (and later kernels) and previous kernels.

The implementation of IP fragments backdoor is left as an exercise for the reader.

VII. Other considerations

As mentioned previously, the current implementation of the backdoors does not feature hiding the shell commands spawned by a backdoor. It is a nontrivial task; assuming we can inspect dom0 kernel data structures, it is very difficult to hide a process. Consequently, although it is possible to hijack dom0 syscalls by altering int 0x80 dispatching, it will only help to hide backdoor activity from the usermode, not from the kernel, so it is of little use and has not been implemented.

Instead of spawning a shell in a separate, well-visible process, backdoors could force execution of arbitrary operations (say, system calls) in $event/x$ kernel thread; this would be more difficult to spot. Still, any operation that changes dom0 data structures or filesystems in an anomalous way can be detected with enough effort. However, if all that attacker needs is the content of memory (say, cryptographic keys in some process), then this information can be retrieved really stealthily.

In order to provide convenient ability to execute code in ring 0, we have overwritten the $do\_ni\_hypercall$ with "call first_argument" assembly instructions. Anyone can call this hypercall and thus discover the backdoor presence. The $seal$ module, shipped with the framework, replaces $do\_ni\_hypercall$ body with a function that checks the arguments for the magic values before actually calling a function provided by the caller.

Currently, the framework and the direct_hdd module work on both x86_32 and x86_64 architectures; the backdoors are for x86_32 only, but it should be straightforward to port them to x86_64.

We have discussed taking control over hypervisor's code by DMA. If there is a programming error in the hypervisor code (e.g. a buffer overflow in a hypercall), it could allow to overwrite hypervisor's code and install the backdoors as well. Moreover, if the said error was reachable by an unprivileged domain, it could allow for direct elevation to ring0 from domU.

Other hypervisors that are designed to be the only all-powerful entities in the system (and thus are able to control administrative operations), e.g. Hyper-V[15], are attractive targets for placing a backdoor as well.

This paper addresses the defense methods to the presented attacks very superficially; they will be discussed in more depth in other publication.

[11]Note that if we use an already existing, legal domain for lurking purposes instead of spawning a dedicated lurker domain, then the Xen code can be left unmodified, which can be significant for some integrity scanner capable of reading memory occupied by Xen code

References

[1] Joanna Rutkowska, *Subverting VistaTM Kernel for Fun and Profi t*, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf

[2] Dino Dai Zovi, *Hardware Virtualization Rootkits*, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf

[3] *Xen 3.2*, http://bits.xensource.com/oss-xen/release/3.2.0/xen-3.2.0.tar.gz

[4] *VMware Workstation*, http://www.vmware.com/products/ws/

[5] *CVE-2007-4993*, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4993

[6] *Multiple integer overflows in e2fsprogs (Xen related)*, http://www.mcafee.com/us/local_content/misc/threat_center/e2fsprogs.pdf

[7] Intel, *Intel Trusted Execution Technology*, http://www.intel.com/technology/security/index.htm

[8] AMD, *AMD IOMMU specifi cation 1.2* http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf

[9] Intel, *Intel Virtualization Technology for Directed I/O (Intel VT-d)*, http://www.intel.com/technology/magazine/45nm/vtd-0507.htm?iid=techmag_0507+rhc_vtd

[10] *The MINIX 3 Operating System*, http://www.minix3.org/

[11] Luic Duflot, *Programmed I/O accesses: a threat to Virtual Machines ?*, http://www.ssi.gouv.fr/fr/sciences/fichiers/lti/pacsec2007-duflot-papier.pdf

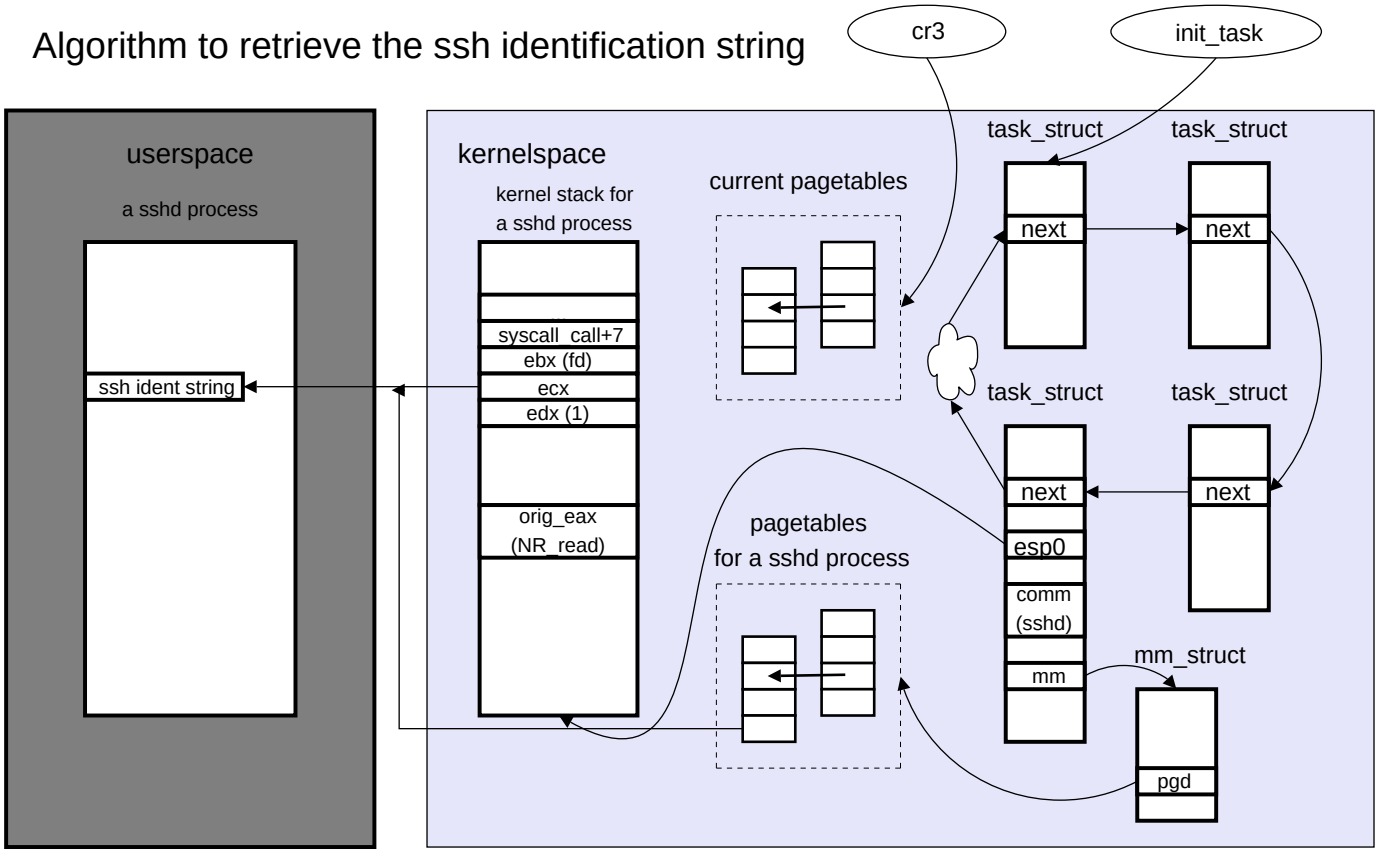
[12] Luic Duflot, *Security Issues Related to Pentium System Management Mode*, http://cansecwest.com/slides06/csw06-duflot.ppt

[13] halfdead@phear.org, *Mistifying the debugger, ultimate stealthness*, http://www.phrack.com/issues.html?issue=65&id=8

[14] *XenAccess Library*, http://code.google.com/p/xenaccess/

[15] Microsoft, *Hyper-V*, http://en.wikipedia.org/wiki/Hyper-V

VIII. Appendix A



IX. Appendix B

A verbatim copy of the source code of a module that hides a presence of a selected domain from dom0.

```
#include <xenlib.h>
#include <stdint.h>
#include <public/xen.h>
#include <public/domctl.h>
long (*orig_domctl) (void *);

/* set by xenload command line */
long hidden_domain = 0xaabbccdd;

long my_domctl(struct xen_domctl *p)
{
        long ret = orig_domctl(p);
        if (ret || p->cmd != XEN_DOMCTL_getdomaininfo
            || p->domain != hidden_domain)
                return ret;
/* p contains info on the hidden domain;
overwrite it with the info on the next domain */
        p->domain++;
        return orig_domctl(p);
}

int init_module()
{
        orig_domctl = hypercall_table[__HYPERVISOR_domctl];
```

```
        hypercall_table[__HYPERVISOR_domctl] = my_domctl;
        return 0;
}

void cleanup_module()
{
        hypercall_table[__HYPERVISOR_domctl] = orig_domctl;
}
```