Report

Poacher Turned Gamekeeper

Lessons Learned From Eight Years of Breaking Hypervisors

Bromium®

Table of Contents

AUTHOR

Rafal Wojtczuk

Summary

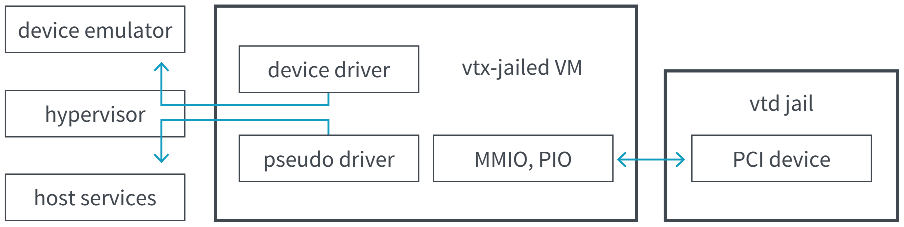
Hypervisors have become a key element of both cloud and client computing. It is without doubt that hypervisors are going to be commonplace in future devices, and play an important role in the security industry. In this report, we discuss in detail the various lessons learned while building and breaking various common hypervisors. In particular, we take a trip down memory lane and examine a few vulnerabilities found in popular hypervisors that have led to break-outs, trying to offer a generic mitigation when possible. To add some spice, we will talk about details of four not-yet-discussed vulnerabilities we recently discovered in VirtualBox, and examine Direct Memory Access (DMA) attacks against DeepSafe.

Scope

There is a plethora of various hypervisor solutions available nowadays. Some of them are designed from scratch with security in mind, and for example, use formal verification to provide assurance about their security. As those solutions are not the mainstream today, in this report we will focus on the popular commercial virtualization software commonly used nowadays. One of the solutions, DeepSafe is very different from the others, and will be covered in the later part of the report. We will start with a discussion about common Type 1 and Type 2 hypervisors; Xen, VirtualBox, VMWare, ESX, HyperV all belong to these two categories.

“SEVERAL ATTACK VECTORS CAN BE USED BY MALICIOUS CODE RUNNING IN A VTX VIRTUAL MACHINE TO BREAK OUT OF THE VM.”

TYPE 1 AND TYPE 2 ATTACK SURFACE



What are the attack vectors that a malicious code running in the VTx virtual machine (VM) can use to break out of the VM? They can be grouped in the following categories:

1. **Vulnerabilities in VTx itself.** Certainly, if the CPU does not isolate properly the code running in non-root mode, then all bets are off. There are certain CPU erratas related to VTx support (e.g., AAJ66 [1]). Most of them require “complex micro-architectural conditions”, that are not disclosed in the erratas. We are not aware of any successful exploit based of VTx CPU errata. Also note that CPU erratas have the potential to circumvent any security software, not only hypervisors.
2. **Vulnerabilities in the core hypervisor (code near vmexit handler).** Such vulnerabilities are definitely thinkable; however, the size of the hypervisor core is relatively small, therefore the potential for vulnerabilities is limited.
3. **Device emulators.** There are many cases of such vulnerabilities. If a device emulator runs in a privileged environment (e.g., Type 2 host), such a vulnerability is fatal.
4. **Other hypervisor-related services.** One common example is shared folders service. By their very nature, they usually run in a privileged environment and again compromise of them is fatal.
5. **Pass-through PCI devices.** A VM that can command a PCI device can mount certain additional attacks. Notably, a hypervisor must protect its memory via VTd from DMA attacks; other types of attacks are also known [2]. Note that it is possible to create a usable virtualization system without the need to grant any PCI devices to VMs. Also, in certain scenarios, PCI passthrough actually decreases attack surface—see the discussion below on service VMs.

"IF THE GOAL OF THE HYPERVISOR IS TO PROVIDE A RELIABLE ISOLATION OF MALICIOUS CODE, THEN CARE MUST BE TAKEN TO DESIGN AND IMPLEMENT THE HYPERVISOR IN A WAY THAT MINIMIZES EXPOSURE."

"THE ATTACK SURFACE SHOULD BE MEASURED NOT BY TOTAL LINES OF CODE OF THE WHOLE PRODUCT, BUT BY THE AMOUNT AND COMPLEXITY OF THE UNTRUSTED INPUT THAT MUST BE PROCESSED."

Some virtualization solutions are focused on providing as much features and functionality as possible, and inherently that increases the attack surface. Note that in case when users do not run potentially malicious code in the VMs, it is the natural approach. On the other hand, if the goal of the hypervisor is to provide a reliable isolation of malicious code, then care must be taken to design and implement the hypervisor in a way that minimizes exposure.

It is obvious that hypervisors present a non-negligible attack surface, and there are numerous examples of vulnerabilities allowing the malicious code to break out of the VM. Should we stop using hypervisors as a method to isolate malware? The answer is no—because they seem to be the best available mechanism to implement isolation. The other method is OS-based sandboxing, with the Chrome sandbox being a primary example. However, this method relies on the security of an underlying OS—a vulnerability in the OS kernel can be used to break out of the OS-based sandbox [3]. At least in the case of Windows, its kernel is a vast attack surface—400 syscalls, 800 win32k.sys syscalls, resulting in 76 CVEs for kernelmode issues in Windows in the year 2013 alone.

It is very difficult to come up with a convincing quantitative comparison of a given hypervisor attack surface vs a given OS kernel (let's focus on Windows). The common way is to compare the size of the code base, because it is easy to come up with hard numbers. For example, xen-4.4.0 is approximately 1.7 million lines of code (LOC). It can be trimmed to 110K lines of usermode code and 60K lines of ring0 code, still retaining rich hypervisor functionality. Windows7 kernel is believed to have approximately 2 million lines of code; win32k.sys is probably larger. It shows that a hypervisor can be much smaller. However, it is extremely difficult to determine what percentage of the code base actually is responsible for processing untrusted and potentially malicious input, which makes all LOC comparisons essentially a futile game.

The attack surface should be measured not by total LOC of the whole product, but by the amount and complexity of the untrusted input that must be processed. Unfortunately, it is difficult to measure it in a way that makes it possible to compare the attack surfaces between different solutions. We need to

"THERE ARE CERTAIN PROPERTIES OF A HYPERVISOR THAT ARE OBJECTIVELY ATTRACTIVE FROM THE SECURITY PERSPECTIVE."

rely on subjective views, based by experience. Most knowledgeable parties agree that a well-written hypervisor has a much smaller attack surface than a Windows kernel, the difference being even more significant than what could be concluded by LOC comparison.

Putting subjective views aside, there are certain properties of a hypervisor that are objectively attractive from the security perspective. Particularly, the vmexit boundary is much stronger than syscall and process boundaries, which poses a challenge for the exploitation of hypervisor memory corruption vulnerabilities. A few facts:

1. In the case of broker-vulnerability-based sandbox escapes, on Windows the attacker knows the libraries bases—no ASLR protection.
2. In the case of kernel exploits, the attacker can craft useful data structures in usermode that can be misinterpreted by the kernel, because the address space is the same (unless SMAP—but no SMAP for Windows anytime soon).
3. Windows kernel hands out its memory layout for free to the attacker (better on Windows8.1) [4].
4. In the case of browser vulnerabilities, the attacker has a lot of control over memory layout, thanks to javascript/other scripting, that allows for a very high degree of precision when mounting an attack, which is crucial for the reliable exploitation.

No such problems exist in the case of hypervisor-enforced separation. Usually, just because of the need to bypass ALSR, the attacker needs two separate vulnerabilities, a pointer leak and write primitive (while in, for example the browser case, a single use-after-free vulnerability usually provides both). Most VM-escape exploits known to the author relied on ASLR being not functional (i.e., executable not position-independent, or libraries with a fixed base). One notable exception is the Cloudburst [5] exploit against VMWare, because the underlying integer overflow allowed for full read-write access of the host process memory—again, this seems to be a rare case.

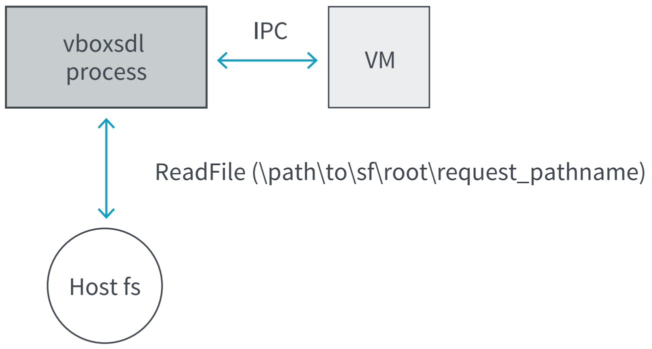
"OFTEN WE DO NOT HAVE TO CHOOSE BETWEEN OS-BASED AND HYPERVISOR-BASED ISOLATION—WE CAN COMBINE BOTH IF REASONABLE CONDITIONS ARE MET."

An important point is that often we do not have to choose between OS-based and hypervisor-based isolation—we can combine both, if the following, reasonable conditions are met:

1. The hypervisor can be attacked only after compromising the VM kernel (note some hypervisors, such as VMWare, expose host services to the unprivileged VM usermode).
2. Hypervisor-related drivers in the VM do not weaken the VM kernel security significantly.
3. There is nothing valuable in the VM so placing a OS-based sandbox within a VM is a pure gain—the attacker would need to pierce both protections.

Case Studies

Let's have a look at a few new and a few old vulnerabilities in hypervisors, trying to pinpoint the design issues, and offer a generic mitigation if possible. We will start with four new VirtualBox vulnerabilities, reported by the author to the vendor in March 2014 and fixed in July 15 Critical Patch Update [6]. Three of them are related to shared folder functionality:



Shared folders need to be enabled in the VM configuration; the path of the root of the shared folder tree (\path\to\sf\root in the diagram above) needs to be specified there. The VM sends a message containing the operation type (read, write, and others) and a pathname (request_pathname in the diagram above) to the device model process running on the host (vboxsdl.exe) over a communication channel (implemented over HGCM; the nature of the channel is irrelevant for our purposes). Vboxsdl performs the requested operation and returns the result to the VM.

VirtualBox shared folders host code is relatively complex:

1. Supports utf8 and unicode pathnames
2. Supports pathname casing correction (needed when VM has case-insensitive pathnames, and host does not)
3. Guest can specify path delimiter; host is supposed to normalize path changing each occurrence to \

Likely because of 1, there is no early null-termination check for the received pathname.

VirtualBox Issue S0434934

Memory corruption in vbsfbuildfullpath()

```
/* Correct path delimiters */
if (pClient->PathDelimiter != RTPATH_DELIMITER)
{
LogFlow((“Correct path delimiter in %ls\n”, src));
while (*src) // src comes from VM, not null-terminated
{
if (*src == pClient->PathDelimiter)
*src = RTPATH_DELIMITER;
src++;
}
```

As we can see, if the VM sends a string that is not null-terminated, vbsfbuildfullpath() overwrites each occurrence of (attacker-chosen) pClient->PathDelimiter character to RTPATH_DELIMITER (that is defined as '\'), until a null-terminator is found. If the VM sends a string that is not null-terminated, this will corrupt the adjacent memory. The exploitation for code execution is difficult due to the limited control the attacker has over the way the memory is corrupted, but not ruled out—particularly, overwriting a location that holds the size of a buffer might pave the way for a subsequent buffer overflow.

This is a textbook example of bad design. As both sides of the channel are VirtualBox code, it should be the guest side that does all the conversions. The host side should accept only a narrow range of inputs, say, only Unicode null-terminated ones—checking for input conformance requires much less code than actually doing the conversions.

VirtualBox Issue S0434968

Shared folders directory traversal

Obviously, just concatenating the pathname received from the VM to the shared folder root leads to directory traversal via „..\..\..\..\..\request_pathname", so the service needs to sanitize the pathname to prevent this. The VirtualBox path sanitize algorithm is:

1. Split the path into components (/ or \ is the path separator)
2. Start with depth_credit=0
3. For each component do: Switch (component)
4. For the case "." : do nothing
5. For the case "..": depth_credit-- //fail verification if depth_credits becomes negative
6. Default: depth_credit++;

The goal is to check whether the number of ".." components in any path prefix is not greater than the number of "normal" path components. „dirname\.." is ok, „dirname\..\.." is not. If the above algorithm returns success, the request pathname is appended to the shared folder root and passed directly to the host file API.

The above algorithm has a fatal flaw: it assumes that "\" is a path separator. It is true on a Windows host, but not on a Posix (say, Linux) host. As a result, assuming the VM is a Linux system with a writable shared folder mounted on /mnt/vboxsf, and the host runs Linux, the following sequence of shared folders operations will allow the VM to access an arbitrary file on the host:

1. Mkdir /mnt/vboxsf/a\a\a\a\a\a\a\a\a\a
2. Access /mnt/vboxsf/a\a\a\a\a\a\a\a\a\a/../../../../../../../../arbitrary/file/on/the/host

Pathnames verification is a tricky subject, but there are well-known methods to deal with possible directory traversal attacks. The best way is to use OS-provided mechanisms—chroot() on Posix, \\?\ prefix on the Windows host. If we resort to hand-crafted verification, it should be simple—we can require the guest to take care of "dirname/.." removal, and then the host can deny any request with ".." path component in it.

VirtualBox Issue S0434952

Data leak in HGCM, exploitable via shared folders operations

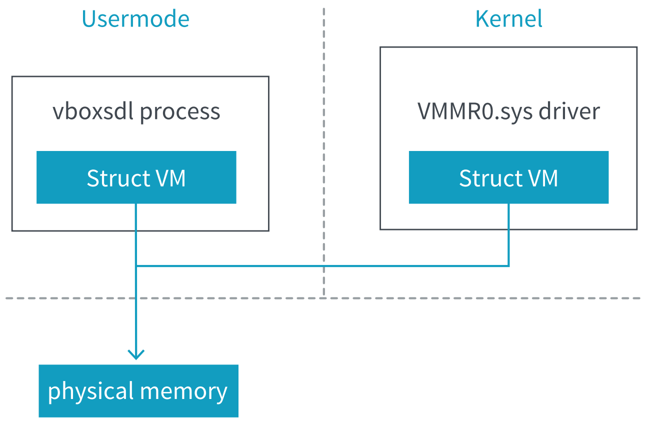
If the VM sends a SHFL_FN_READ message requesting Nreq bytes to be read, the response from the host contains: a response buffer B and the amount of bytes read Nresp. Due to the particular way the underlying HGCM transport mechanism works (and some negligence), B is a character buffer always Nreq bytes long, even if Nresp is actually smaller. So, when the VM requests to read 1024 bytes from a zero-length file, the host returns a 1024 bytes long uninitialized buffer (plus information that 0 bytes have been read), thus leaking contents of uninitialized malloced buffer from the host.

VirtualBox Issue S0434947

Frontend to kernel escalation on the host

This vulnerability cannot be exploited by a rogue VM. Instead, it allows an unprivileged user on the Windows host to achieve kernelmode arbitrary code execution.

The problem is that the crucial data structure “struct VM” is mapped read-write both in the kernel component (VMMR0.sys driver) and in the usermode frontend process (vboxsdl.exe):



It was confirmed that “struct VM” contains kernelmode code pointers, which are frequently dereferenced by the VMMR0 driver. A malicious usermode frontend process can overwrite these pointers and gain kernelmode execution.

The above scenario was tested on a Windows host—an unprivileged user can attach vboxsdl.exe with a debugger and perform the required malicious actions. On a Linux host, the frontend process runs as root, so it cannot be attached with a debugger (and /dev/vboxdrv device default permissions do not allow an unprivileged process to create a VM).

The remaining vulnerabilities referenced below are not new.

CVE-2007-5497

Integer overflow in libext2fs

In order to construct a PV domain on Xen, one must specify the kernel image to use. Xen's pygrub process runs in privileged domain dom0 and uses libext2fs to retrieve the kernel image from the VM's filesystem. The latter is controlled by malicious code running in the VM. So, the attacker that has administrative privileges in the VM can corrupt VM filesystem in a way that trigger any vulnerability (e.g., CVE-2007-5497) in libraries that parse the filesystem structures, and reboot the VM. In order to restart the VM, pygrub will be invoked, and it will parse the malicious filesystem—this might result in code execution in dom0.

The problem is elegantly solved by not using pygrub and switching to pvgrub. The latter is a "stub", a static kernel that one passes to Xen as a VM kernel. Pvgrub does all the filesystem parsing, and chainboots the real kernel. Thus, the untrusted filesystem is never parsed in the privileged dom0. Lesson—again, offload to the VM as much as possible.

CVE-2011-1751

Use-after-free in qemu/KVM

This vulnerability and its exploitation were described in detail during Black Hat USA 2011. Briefly, a malicious administrative user in a KVM VM can request a PCI unplug action (via writing to the emulated chipset registers) on a device that was not hotplugged in. This results in use-after-free condition in the device model process (running on the host) that leads to code execution on the host.

It is instructive to notice that a feature that was really needed by very few users could be leveraged to compromise all other users. By default, this feature should be disabled.

Generally, it would be good to reduce the attack surface by not allowing a VM to exercise code paths in the device model related to the PCI config space access. The key observation is that this functionality is needed only for VM boot.

Thus, if we plan to use a given VM as a container for potentially malicious code, we can prepare it in steps:

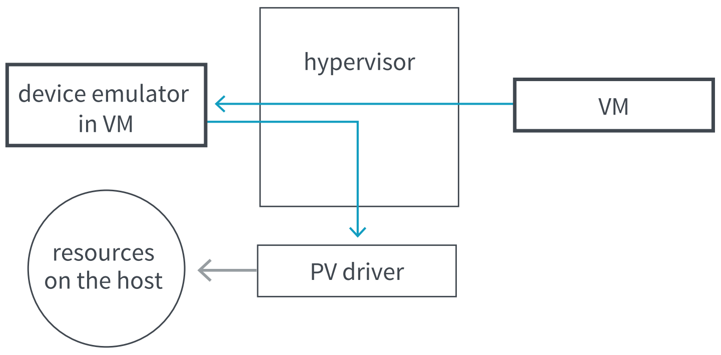
1. Start the VM with all PCI config space access granted (and other resources needed for boot time only); do not expose the VM to malicious input yet
2. Save the VM
3. Change the VM configuration so that it denies PCI config space access
4. Restore the VM with this new restrictive configuration
5. Expose the VM to the malicious input

Then even if the VM gets compromised at step 5, it runs in a more restricted environment that might mitigate attacks. The term "delusional boot" [7] was coined to describe this technique.

CVE-2012-0029

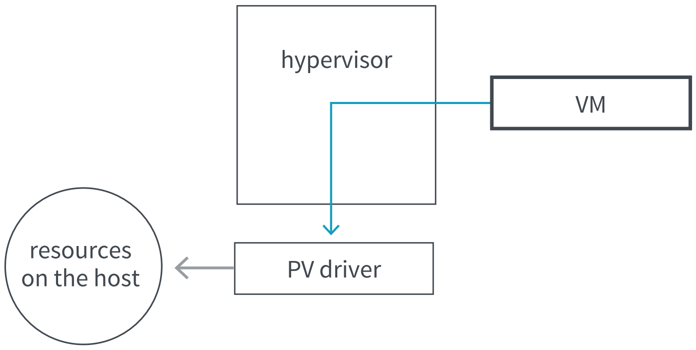
Heap-based buffer overflow in the process_tx_desc function in the e1000 emulation

As mentioned initially, device emulation presents a significant attack surface, and this CVE is a good example why. In case of Type 2 hypervisors, the relevant device model process usually runs on the host, and its compromise is fatal. It is thinkable to use OS facilities to restrict privileges of this process—however, if we trust the OS to do a sufficiently good job in isolating, we would not need hypervisors for isolation. So a better idea is to sandbox device emulation in a dedicated "stub VM":



The assumption is that the PV driver interface is much thinner that the interface exposed by the stub VM. Therefore, even if a malicious VM is able to achieve code execution in the context of device emulator, the damage is small—the attacker would need another exploit for PV driver.

In fact, if we do not require an unmodified VM, we can install proper PV drivers in the VM: Thus getting rid of large part of device emulation entirely.



CVE-2007-0069

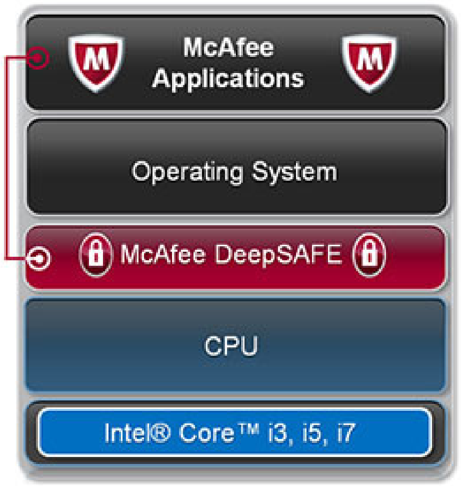
Windows kernel TCP/IP/IGMPv3 and MLDv2 vulnerability, remote code execution

This vulnerability is not related to virtualization, but it is interesting to note how hypervisors can mitigate such problems. Some software components normally run in a privileged environment and are exposed to attacks from the outside (not from the VM), with networking stack and wireless card drivers being primary examples. If we move this code to a dedicated "service VM" then the impact of an exploit against this component is greatly reduced. This requires granting the service VM full control over the relevant hardware.

In fact, it is thinkable to turn the whole type 2 host into a giant service VM [8]. If we implement the hypervisor core in a way that protects normal VMs from a compromised host, this provides significant advantage (particularly, it solves all the problems with device emulation code). However, this approach needs careful implementation—particularly, the hypervisor needs to be protected against hardware-based attacks originating in the host.

Deepsafe and DMA Attacks

Deepsafe hypervisor [9] is very different from Type 1 and Type 2 ones
The Deepsafe hypervisor installs itself in a way very similar to Bluepill [10]. Early during Windows OS boot time, it wraps the whole OS in a VM, and resumes the OS in the VM. There is only one VM in the system—the one holding the OS. The benefit is that even if malicious code finds a way into the OS and even executes a kernel exploit (at this stage, all security methods implemented in OS are bypassed), then still Deepsafe stays intact and has a chance to detect malicious activity in the OS. Currently Deepsafe focuses on detecting typical rootkit activity, e.g., changes to SSDT, IDT, kernel body and syscall-related MSRs.



The Deepsafe hypervisor needs to protect its body from a malicious OS. In order to prevent the CPU (running in a VM context) from accessing the hypervisor memory, the EPT mechanism is used, which is sound. Still, as there is no device virtualization, the OS needs full access to hardware. Particularly, it can instruct PCI devices to overwrite any memory via DMA. In order to prevent this attack, the VT-d [11] mechanism must be used—by configuring VT-d, the hypervisor can disallow certain memory ranges from being accessed via DMA.

It is somewhat unexpected, but Deepsafe (the latest available version 1.6.0 was tested) does not configure the VT-d engine, and thus is vulnerable to DMA-based attacks. They are not something new—such attacks were demonstrated in Black Hat USA 2008 [12] by the author against Xen. There are also well known discussions [13] about the necessity of it.

One possible explanation for the lack of VT-d protection in Deepsafe is that the vendor thought they are infeasible. Indeed, if we wanted to program given PCI device registers to initiate custom DMA directly, this would be a tedious device-specific task, and it would require a method to stop the OS from interfering with this process, at the same time maintaining system stability. However, there is a simple way to achieve the same—we can use the existing disk drivers for it (a similar approach was used in [12]).

The procedure to conduct arbitrary DMA on Windows, in a stable manner, is:

1. Achieve kernel privileges
2. Allocate a page at virtual address V
3. Change PTE of V so that it points to physical address P
4. CreateFile(... FILE_FLAG_NO_BUFFERING ...)
5. ReadFile/WriteFile(..., V,...) will do DMA to P

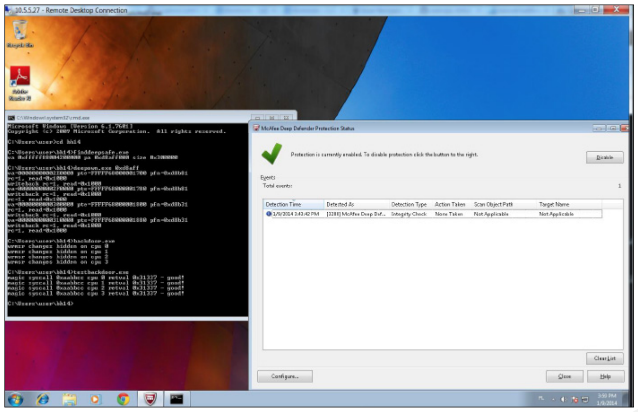
Step 4 instructs the OS to do all file operations on the given file directly via DMA to user a buffer. There are restrictions—the transfer size must be aligned with the sector size, and Bitlocker is not activated. Step 3 fools the OS into thinking that the user buffer is at [arbitrary] physical address P. The OS will dutifully program the disk controller so that it will do DMA to the physical address of the attacker's choice.

As a result, a compromised OS can overwrite Deepsafe's body, and take full control over it. One possible way to use this capability is to disable the hypervisor entirely, by returning the OS into root mode. While possible, it looks like a lot of work, and may interact with other Deepsafe components running in the host.

From the attacker's perspective, a much better idea is to inject a rootkit into the Deepsafe hypervisor. It is the perfect location—secured from the rest of the OS, and having much more control over it. For proof-of-concept purposes, a very simple rootkit was implemented, that is installed in the code path handling vmexit for the reasons EXIT_REASON_MSR_READ and EXIT_REASON_MSR_WRITE. Normally, Deepsafe configures VTx so that reads from LSTAR MSR do not cause vmexit, and writes to it cause vmexit, resulting in reporting an alert about possible rootkit-like behavior. We change the hypervisor body so that:

1. Writes to LSTAR MSR are allowed and dutifully executed, without any alert being generated
2. Reads from LSTAR MSR cause vmexit (change to MSR bitmap is needed for this) and are emulated in a way that return to the OS the original value installed by the OS

Because of the above, the attacker in the OS is able to silently alter the LSTAR MSR as he wishes. It allows controlling all the syscall handling by the OS, which results in efficient rootkit capabilities. Moreover, Patchguard running in the OS is blinded—it still sees the original LSTAR MSR value when running its checks, because the hypervisor lies to it about this MSR value. To sum up, not only have we disabled the Deepsafe detection, we also have used its power to disable in-OS rootkit checks.


10.5.5.27 - Remote Desktop Connection
Recycle Bin
Adobe Reader XI
C:\Windows\system32\cmd.exe
McAfee Deep Defender Protection Status

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\user>cd bh14

C:\Users\user\bh14>finddeepsafe.exe
va 0xfffff80004200000 pa 0xd8aff000 size 0x300000

C:\Users\user\bh14>deepown.exe 0xd8aff
va=0000000000230000 pte=FFFFF68000001780 pfn=0xd8b01
rc=1. read=0x1000
writeback rc=1. read=0x1000
va=0000000000270000 pte=FFFFF68000001780 pfn=0xd8b01
writeback rc=1. read=0x1000
rc=1. read=0x1000
va=0000000000300000 pte=FFFFF68000001800 pfn=0xd8b31
rc=1. read=0x1000
writeback rc=1. read=0x1000
va=0000000000310000 pte=FFFFF68000001880 pfn=0xd8b31
writeback rc=1. read=0x1000
rc=1. read=0x1000

C:\Users\user\bh14>backdoor.exe
wrmsr changes hidden on cpu 0
wrmsr changes hidden on cpu 1
wrmsr changes hidden on cpu 2
wrmsr changes hidden on cpu 3

C:\Users\user\bh14>testbackdoor.exe
magic syscall 0xaabbcc cpu 0 retval 0x31337 - good!
magic syscall 0xaabbcc cpu 1 retval 0x31337 - good!
magic syscall 0xaabbcc cpu 2 retval 0x31337 - good!
magic syscall 0xaabbcc cpu 3 retval 0x31337 - good!

C:\Users\user\bh14>
```

Protection is currently enabled. To disable protection click the button to the right.

Events
Total events: 1

Detection Time	Detected As	Detection Type	Action Taken	Scan Object Path	Target Name
1/9/2014 3:43:42 PM	[3288] McAfee Deep Def...	Integrity Check	None Taken	Not Applicable	Not Applicable

There are a few interesting technical details regarding the above hypervisor overwrite. First, malware running in the OS needs to know where in the physical address space the Deepsafe hypervisor is located. Dumping all of the physical address space via DMA and doing pattern search in it is possible, but troublesome. A more elegant approach was found—it turns out that when an EPT fault occurs because the OS tried to read from a physical address belonging to the hypervisor, then Deepsafe does not bother to emulate the instruction, it just skips it. Thus, the following function will return MAGICVALUE if memory at rcx belongs to Deepsafe, and something else (real memory content) if not.

```
Mov rax, MAGICVALUE
Mov rax, [rcx]
Ret
```

Deepsafe allocates a contiguous physical memory region of size 0x300000, so it is easy and fast to find it via scanning all the memory.

Another question is how to trojan Deepsafe effectively. The implemented proof-of-concept code is simple, it works with a given Deepsafe version only, and just places a hook at the first instruction of a vmexit handler; it stores the hook body in the unused end of the last page used by the text section. A more generic approach is possible, based on the fact that it is feasible to find the VMCS hypervisor control structures by pattern matching [14] in the memory area reserved by Deepsafe. VMCS includes the vmexit handler location and CR3 value used by the hypervisor, so that we can change both reliably and without the need to hook the beginning of the original vmexit handler in a nondestructive manner.

More Concerns About Deepsafe

Currently Deepsafe detects only a very specific set of rootkit-like functionality, mostly duplicating Patchguard functionality (obviously, in a more reliable manner). However, there are legal interfaces exposed by the Windows kernel, like filter drivers, which can be abused to gain rootkit-like functionality. As these interfaces are used by non-malicious software (say AV), the mere use of them is not a clear indication of an attack.

Other concerns are, just briefly mentioned for brevity:

1. A compromised host kernel can overwrite crucial usermode memory (e.g., disabling usermode Deepsafe components)
2. How secure is mfeib.sys launch, on reboot/S3 resume?
3. No trusted UI domain
4. A host can mess with PCI config, SMM, BIOS, and PCI devices firmware

For more information

For more information, contact your Bromium sales representative or Bromium channel partner. Visit us at www.bromium.com.

References

[1] Intel® Core™ i7-900 Specification Update, http://download.intel.com/design/processor/specupdt/320836.pdf

[2] Rafal Wojtczuk, Joanna Rutkowska, *Following the White Rabbit: Software attacks against Intel® VT-d technology*, http://invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf

[3] *MWR Labs Pwn2Own 2013 Write-up—Kernel Exploit*, https://labs.mwrinfosecurity.com/blog/2013/09/06/mwr-labs-pwn2own-2013-write-up---kernel-exploit/

[4] Alex Ionescu, *KASLR Bypass Mitigations in Windows 8.1*, http://www.alex-ionescu.com/?p=82

[5] Kostya Kortchinsky, *"CLOUDBURST: A VMware Guest to Host Escape Story*, BHUSA09

[6] Oracle Critical Patch Update Advisory—July 2014, http://www.oracle.com/technetwork/topics/security/cpujul2014-1972956.html

[7] Anh Nguyen, Himanshu Raj, Shravan Rayanchu, Stefan Saroiu, Alec Wolman, *Delusional Boot: Securing Cloud Hypervisors without Massive Re-engineering*, http://research.microsoft.com/pubs/197590/inception.pdf

[8] Ian Pratt, *μXen*, http://www-archive.xenproject.org/xensummit/xs12na_talks/T6.html

[9] McAfee DeepSAFE, www.mcafee.com/us/solutions/mcafee-deepsafe.aspx

[10] Joanna Rutkowska, Bluepill, http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf

[11] Intel® Virtualization Technology for Directed I/O (VT-d), https://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices

[12] Rafal Wojtczuk, *Subverting the Xen Hypervisor*, http://www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf

[13] Joanna Rutkowska, *Thoughts on DeepSafe*, http://theinvisiblethings.blogspot.co.uk/2012/01/thoughts-on-deepsafe.html

[14] Gianluca Guida, private conversations

Br Bromium®

Bromium, Inc.
20813 Stevens Creek Blvd
Cupertino, CA 95014
info@bromium.com
+1.408.213.5668

Bromium UK Ltd.
Lockton House
2nd Floor, Clarendon Road
Cambridge CB2 8FH
+44.1223.314914

For more information go to www.bromium.com
or contact sales@bromium.com