Attacks on More Virtual Machine Emulators

Peter Ferrie, Senior Principal Researcher, Symantec Advanced Threat Research
peter_ferrie@symantec.com

Abstract As virtual machine emulators have become commonplace in the analysis of malicious code, malicious code has started to fight back. This paper describes known attacks against the most widely used virtual machine emulators (VMware and VirtualPC). This paper also demonstrates newly discovered attacks on other virtual machine emulators (Bochs, Hydra, QEMU, Sandbox, VirtualBox, and CWSandbox), and describes how to defend against them.

Index Terms Hardware-assisted, Hypervisor, Para-virtualization, Virtual Machine

I. INTRODUCTION

Virtual machine emulators have many uses. For anti-malware researchers, the most common use is to place unknown code inside a virtual environment, and watch how it behaves. Once the analysis is complete, the environment can be destroyed, essentially without risk to the real environment that hosts it. This practice provides a safe way to see if a sample might be malicious.

The simplest attack that malicious code can perform on a virtual machine emulator is to detect it. As more security researchers rely on virtual machine emulators, malicious code samples have appeared that are intentionally sensitive to the presence of virtual machine emulators. Those samples alter their behavior (including refusing to run) if a virtual machine emulator is detected. This behavior makes analysis more complicated, and possibly highly misleading. Some descriptions and samples of how virtual machine emulators are detected are presented in this paper.

A harsher attack that malicious code can perform against a virtual machine emulator is the denial-of-service; specifically, this type of attack causes the virtual machine emulator to exit. Some descriptions and samples of how that is done are presented in this paper.

Finally, the most interesting attack that malicious code can perform against a virtual machine emulator is to escape from its protected environment. No examples of this type of attack are presented in this paper.

It is important to note here that most virtual machine emulators are not designed to be completely transparent. They are meant to be "good enough" so that typical software can be fooled to run inside them. Their use in the analysis of malicious code was never a requirement. This situation is changing, though, with the creation of new virtual machine emulators, such as Hydra. However, even with full knowledge of what has been used to detect existing virtual machine emulators, it is clearly difficult to develop a virtual machine emulator that cannot be detected. Some descriptions and samples of how to detect Hydra are included in this paper.

The interest in detecting virtual machine emulators is also not limited to the authors of malicious code. If malicious code is released that makes use of its own virtual machine emulator, then it will become necessary for anti-malware researchers to find ways to detect the virtual machine emulator, too.

Sample detection code is presented in Appendix A. For simplicity and to prohibit trivial copying, only 16-bit real mode assembler code for .COM-format files is supplied.

Virtual machine emulators come in two forms: "hardware-bound" (also known as para-virtualization) and "pure software" (via CPU emulation). The "hardware-bound" category can be split into two subcategories: "hardware-assisted" and "reduced privilege guest" (or ring 1 guest).

Both forms of the hardware-bound virtual machine emulators rely on the real, underlying CPU to execute non-sensitive instructions at native speed. They achieve better performance, for this reason, when compared with pure software implementations. However, since they execute instructions on a real CPU, they must make some changes to the environment, in order to share the hardware resources between the guest operating system and the host operating system. Some of these changes are visible to applications within the guest operating system, if the applications know what those changes look like.

SECTION 1: HARDWARE

II. HARDWARE-BOUND VIRTUAL MACHINE EMULATORS

The difference between hardware-assisted virtual machine emulators and reduced privilege guest virtual machines emulators is the presence of virtual machine-specific instructions in the CPU. The hardware-assisted virtual machine emulators use CPU-specific instructions to place the system into a virtual mode. The guest runs at the same privilege level that it would do if it truly controlled the CPU in the absence of the virtual machine emulator. The important data structures and registers have shadow copies that the guest sees, but these shadow copies have no effect on the host.

Instead, the host controls the real data structures and registers. The result is that the virtualization is almost
completely transparent. The host can direct the CPU to notify it of specific events, such as an attempt to query the capabilities of the underlying CPU, or to access particular memory locations and important registers.

By contrast, the reduced privilege guest virtual machine emulators must virtualize the important data structures and registers themselves. The guest is run at a lower privilege level than it would do if it truly controlled the CPU. There is no way to prevent the CPU from notifying the host of all interesting events.

The idea of hardware-bound virtual machine emulators is not new - IBM has been using them for four decades on the System/360 hardware and its descendants.

In the days of DOS, reduced privilege guest virtual machine emulators could be implemented by hooking interrupt 1h, for example. The interrupt hook allows the real CPU to execute instructions at native speed, but the downside is that every instruction is also treated as though it were sensitive.

Another method of reduced-privilege guest virtual machine emulation is buffered code emulation\[^{15}\]. Buffered code emulation works by copying an instruction into a host-controlled buffer and executing it there, if it is not a sensitive or special instruction. Buffered code emulation has fairly good performance.

A major problem for both of these methods, when implemented in DOS, is that DOS has no notion of privileges. Thus, reduced privilege guest is actually a misnomer since it runs at the same privilege level as the host. As a result, code could "escape" from the environment by hooking an "Interrupt ReQuest Vector" (IRQ) and then waiting for that IRQ to be asserted (or, in the case of disk drive IRQs, issuing a command which causes the IRQ to be asserted on completion). There were also problems when the emulation was run in virtual-8086 mode, because the emulator couldn't switch into protected mode and retain control.

This is not a problem for more modern operating systems, though, such as Windows and Linux. In fact, VirtualPC\[^{15}\] uses buffered code emulation. It preloads up to 128 bytes, and executes them from there, if possible. Otherwise, it wraps special code around them, and then it passes them to the VMM.sys driver that performs the actual execution. The use of buffered code emulation allows VirtualPC to intercept instructions that cannot be intercepted by other hardware-bound virtual machine emulators.

Another application that uses buffered code emulation is Dynamo Rio\[^{1}\]. The difference between VirtualPC and Dynamo Rio in this case is that Dynamo Rio runs at an application level and as a Dynamic Link Library within the process space of the guest application, whereas VirtualPC runs at the system level. Dynamo Rio actively attempts to hide itself by intercepting and manipulating memory requests, module lists, etc. Since it is not a virtual machine emulator as defined by the terms described in the introduction, it was not considered further.

Some examples of reduced privilege guest virtual machine emulators are VMware\[^{vi}\], Xen\[^{ix}\], Parallels\[^{vii}\] and VirtualBox\[^{ix}\]. One other product called Virtuozzo\[^{vii}\] is known to the author, but a copy could not be acquired at the time of writing. According to documentation on their website, they virtualize the kernel itself, rather than the hardware. It is unclear what exactly they mean by this.

III. HARDWARE-ASSISTED VIRTUAL MACHINE EMULATORS

Xen 3.x, Virtual Server 2005\[^{xiv}\], and Parallels, can exist as hardware-assisted virtual machine emulators.

From a malicious code author's perspective, the most interesting thing about hardware-assisted virtual machine emulators (hypervisors) is that they can be used to virtualize the currently running operating system at any point in time. Thus, the host can boot to completion, and launch any number of applications as usual, with one them being the virtual machine emulator. That emulator then sets up some CPU-specific control structures and uses the VMLAUNCH (Intel) or VMRUN (AMD) instruction to place the operating system into a virtualized state. At that point, there are effectively two copies of the operating system in existence, but only one (the host) is suspended while the other (the guest) runs freely in the new state. Whenever an interesting event (an interrupt, interrupt, or exception) occurs, the host operating system (the virtual machine emulator) regains control, handles the event, and then resumes execution of the guest operating system.

Thus, any machine that supports the existence of a hypervisor can have a hypervisor start running at any time. Neither the operating system, nor the user, will be aware of it. Further, the hypervisor is actually more privileged than the operating system itself, since it sees the interesting events first and can hide them even from the host operating system. A hypervisor is, in effect, an "enhanced privilege host". Additionally, once a hypervisor is active, no other hypervisor installed later can gain full control of the system. The first hypervisor is in ultimate control.

In theory, once the guest is active, the virtual machine emulator cannot be detected since it can intercept all sensitive instructions, including the CPUID instruction. The instructions that would leak information now see a shadow copy of the sensitive information which appears to correspond to a real CPU. The suggested methods to hide the presence of the hypervisor are: clear the CPUID flag that corresponds to the hardware-assisted "Virtual Machine eXtensions" (VMX) capabilities or emulate the VMX instructions, which would allow for nested virtual machines. The former method is apparently used by BluePill; the latter method is used by Xen.

The method used by Xen is especially interesting since it
means that even a hypervisor can be fooled into thinking that it is running on the real hardware. Normally, one might think that if a hypervisor starts running correctly, then it is in full control of the system. In fact that is not the case.

This promise of "undetectibility" has alarmed many people. Early Intel documentation regarding these Virtual Machine Extensions went as far as to say that it was impossible to detect. More recent documentation has softened the language to say that it is difficult to detect. It is indeed difficult to detect, but not impossible.

The most obvious attack against hypervisors is to check a local time source, such as the "Time Stamp Counter" (TSC). This fact was understood by both Intel and AMD. The result is the "TSCDelta" field in the "Virtual Machine Control Block" (VMCB) which can be used to skew the guest's TSC by an appropriate value to hide the delay caused by faults to the hypervisor.

Therefore, all of the currently documented methods for detecting hypervisors rely on external timing. Specifically, they rely on the fact that executing certain instructions many times will take far longer within a hypervisor environment than without. While that is true, without any baseline comparison (time required for the same machine to run the same number of iterations of the same instructions, prior to the hypervisor being installed), it is impossible to know that a hypervisor is present. Any other time source must be considered suspect. For example, the protocol for interacting with time servers is documented and easily intercepted by the hypervisor.

An alternative exists for Intel-based hypervisors, which relies on a different kind of timing. The method was discovered earlier this year, but no details were given at that time. The method is described below.

The "Translation Lookaside Buffers" (TLBs) can be filled with known data, by accessing a series of present pages. Then if a hypervisor is present, a hypervisor event can be forced to occur by using a hypervisor-sensitive instruction.

In particular, we need a hypervisor-sensitive instruction that is not otherwise destructive to the TLBs. There is only one instruction that meets the criteria: CPUID. CPUID is the only instruction that is intercepted by a hypervisor, is not privileged, and most importantly, does not affect memory in any way.

If the TLBs are explicitly flushed, then the time to access a new page can be determined by reading the time stamp counter before and after the access. This duration can be averaged over the number of TLBs to be filled. Once the TLBs are filled, the time to access a cached page can be determined by reading the time stamp counter before and after the access of each page in the TLBs. This duration can also be averaged over the number of TLBs that were filled.

Next, the CPUID instruction is executed, which will cause a hypervisor intercept to occur, and at least some of the TLBs will be flushed as a side-effect. If a hypervisor event occurred, then each of the pages that should be in the TLBs can be accessed again, and the access time can be measured. If the access time matches that of a new page instead of a cached page, then the hypervisor's presence is revealed.

The TLB method does not work on AMD-based hypervisors because they can direct the hardware to not flush the TLBs when a hypervisor event occurs. However, other methods are available for AMD-based hypervisors, which can also be used to detect Intel-based hypervisors. One similar method is to fill a different cache, such as the L2 via the PREFETCH instruction. At that point, the method is the same: measure the time to fetch something from memory before and after executing CPUID. The L2 cache will be flushed on both kinds of CPU when a hypervisor event occurs.

Other possible methods that should work on both CPUs include the use of particular "Model Specific Registers" (MSRs). The likely candidates are the "Last Branch Record", "Last Exception Record", and "Fixed-Function Performance Counter Register 0".

IV. PURE SOFTWARE VIRTUAL MACHINE EMULATION

Pure software virtual machine emulators work by performing equivalent operations in software for any given CPU instruction. The main advantage that pure software virtual machine emulators have over hardware-bound virtual machines is that the pure software CPU does not have to match the underlying CPU. This allows a guest environment to be moved freely between machines of different architectures. Some examples of pure software virtual machine emulators are Hydra, Bochs, and QEMU.

Another method of virtual machine emulation is most often used by anti-virus software. It emulates both the CPU and a portion of an operating system, such as Windows or Linux. Two examples of this are Atlantis and Sandbox. Both of these are intended to allow a malicious file to "run", while capturing information about its behavior in a completely safe manner. Atlantis supports DOS, Windows, and Linux. Sandbox supports Windows only.

Some virtual machine emulators, such as Hydra, Bochs, and Atlantis, support different CPUs internally, in order to more reliably emulate an environment when the required CPU is not known. A problem for any emulator is that different generations of CPUs can display slightly different behaviors for identical instructions. For Intel 80x86 CPUs, for example, the AAA instruction sets the flags in one of three different ways, depending on whether the CPU is an 80486 or Pentium, a Pentium 2 or Pentium 3, or a Pentium 4 or later. Therefore, if a pure software virtual machine emulator is written for one specific CPU, the software that is emulated might not behave...
correctly. This is, of course, also a problem for hardware-bound virtual machine emulators, but more so in their case because they cannot do anything about it.

V. VIRTUAL MALICIOUS CODE

Predictably, the increasing interest in virtualization has led some researchers to propose malicious uses for virtual machines. One reduced privilege guest virtual machine rootkit, called SubVirt, has been described in detail elsewhere, and is described briefly here. SubVirt works by installing a second operating system. This operating system becomes the new host operating system, which carries an operating system-specific virtual machine emulator. SubVirt supports both the Windows and Linux operating systems. For the Windows platform, SubVirt carries VirtualPC; for the Linux platform, SubVirt carries VMware. Once the new host operating system loads and runs the virtual machine emulator, the virtual machine emulator places the old host operating system into a virtual machine and carries on as before. In the absence of software that is able to recognize the presence of a virtual machine emulator, software within the system will not easily determine that the system has been compromised.

Two hardware-assisted virtual machine rootkits have also been described elsewhere, by their authors. One is BluePill, and the other is Vitriol. Both of them work by making use of the virtual machine extensions that exist in newer AMD and Intel CPUs respectively.

It seems that none of these applications is available to other anti-malware researchers.

VI. DETECTING VMWARE

VMware is a proprietary, closed-source, reduced privilege guest virtual machine emulator. It supports guest-to-host and host-to-guest communication. Since it relies on the underlying hardware for execution of instructions, it must relocate sensitive data structures, such as the “Interrupt Descriptor Table” (IDT) and the “Global Descriptor Table” (GDT). VMware also makes use of the “Local Descriptor Table” (LDT) which is not otherwise used by Windows. Thus, a simple detection method for VMware is to check for a non-zero LDT base on Windows. The more common method for detecting VMware is to check the value of the IDT, using the “RedPill” method. For the “RedPill” method, if the value of the IDT base exceeds a certain value, a virtual machine emulator is assumed to be present. However, as the LDT paper shows, this method is unreliable on machines with multiple CPUs. The “Scooby Doo” method uses the same basic idea as the RedPill method but it compares the IDT base value to specific hard-coded values in order to identify VMware specifically. While the Scooby Doo method is less likely to trigger false positives, compared to the RedPill method, there is still the chance that some false positives will occur.

In addition to the Descriptor Table methods, VMware offers a method of guest-to-host and host-to-guest communication which can also be used to detect the presence of VMware. The most common form of this detection is the following:

\[
\begin{align*}
\text{mov} & \text{ eax, 564d5868h } ;'VMXh' \\
\text{mov} & \text{ ecx, 0ah } ;'\text{get VMware version} \\
\text{mov} & \text{ dx, 5658h } ;'\text{VX'} \\
\text{in} & \text{ eax, dx} \\
\text{cmp} & \text{ ebx, 564d5868h } ;'\text{VMXh'} \\
\text{je} & \text{ detected}
\end{align*}
\]

When run in ring3 of a protected-mode operating system, such as Windows or Linux, execution of the IN instruction causes an exception to be generated, unless the I/O privilege level is altered. This is because the IN instruction is a privileged instruction. The reason that the IDT is relocated is to hook this exception privately. The exception can now be normally trapped by an application. However, if VMware is running, no exception is generated. Instead, the EBX register is altered to contain 'VMXh' (the ECX register is also altered to contain the VMware product ID, which is not relevant in this case).

This detection method was attempted recently in the W32/Polip virus. The virus author attempted to obfuscate it and ended up by introducing a bug, so VMware was not detected even when it was running.

Of course, other values in the ECX register can be specified for different effects. Since the execution of the IN instruction should never change register values other than the EAX register in a real machine, disabling the "get VMware version" method alone will not be sufficient to hide VMware.

There are many other ways to detect the presence of VMware, depending on the guest operating system that is in use. For example, the Windows registry is full of VMware-specific keys, but all of these can be removed. Other methods depend on the presence of particular hardware, such as hard disks whose device names are constant, and network cards whose MAC addresses fall within a predictable range. The problem with these dependencies is that, depending on the intended use of the virtual system, none of these hardware elements might be present, and some of them require special privileges to access.

Going beyond detection, in December 2005, it was disclosed that a component of VMware allowed an attacker to escape from the environment. Specifically, the "VMnet" contained an unchecked copy operation while processing specially crafted 'EPRT' and 'PORT' FTP requests. The result was heap buffer corruption within the host environment, with the potential to execute arbitrary code there.

A more serious vulnerability potentially exists in hardware-bound virtual machine emulators, if the guest can interact with third-party devices on the system. For example, if a buffer-overflow vulnerability exists in a network driver in the host environment, it might be possible for an application within the
guest environment to send a specially crafted network packet that reaches the host network driver intact, and thus exploit that vulnerability.

VII. DETECTING VIRTUALPC

VirtualPC is a proprietary, closed-source, reduced privilege guest virtual machine emulator. It supports guest-to-host and host-to-host communication. A version exists for the Macintosh platform, as well as for the Windows platform. Only the Windows version is considered here.

Just like VMware, VirtualPC must relocate sensitive data structures, such as the IDT and the GDT. Just like VMware, VirtualPC makes use of the LDT. Thus, RedPill, LDT, and Scoopy Doo, all work to detect VirtualPC.

Whereas VMware uses a special port to perform guest-to-host and host-to-guest communication, VirtualPC relies on the execution of illegal opcodes to raise exceptions that the kernel will catch. This method is very similar to the illegal opcode execution that Windows NT and later operating systems use in their DOS box to communicate with the operating system. By reverse-engineering the VirtualPC executable file, the author found that the opcodes are the following:

- `0F 3F x1 x2`
- `0F C7 C8 y1 y2`

In ordinary circumstances, execution of these opcodes causes an exception to be generated. The `0F 3F` opcode causes an exception because it is an otherwise undefined opcode. The `0F C7 C8` opcode causes an exception because it is an illegal encoding of an existing opcode. This exception can be trapped by an application. However, if VirtualPC is running, no exception is generated, depending on the values of `x1`, `x2`, `y1`, and `y2`.

The full list of allowed values for `x1` and `x2` is not known. However, the BIOS code in VirtualPC uses the values `0A 00`, `11 00`, `11 01`, and `11 02`. The file-sharing module that can be installed uses value 02 followed by 01-13, and 07 0b. These appear to be examples of guest-to-host communication. An example of host-to-host communication is given in the following: if `x1` is 03 and `x2` is 00, then the current host time (in hour:minute:second notation) is placed into the DX, CX, and AX, registers respectively (see VIRTUALPC TIME demo). Other values for `x1` and `x2`, such as 02 00, return other values in the CPU registers. The values 10 01-03 and 10 06 alter the Z flag. The IsRunningInsideVirtualMachine() API uses the value 07 0B.

The allowed values for `y1` are 00-04. The allowed values of `y2` depend on the value of `y1`. If `y1` is 00 or 03, then `y2` can be 00-03. If `y1` is 01, then `y2` can be 00-02. If `y1` is 02, then `y2` can be 00-04. If `y1` is 00, then `y2` can only be 00. The BIOS code in VirtualPC uses the values 00 00 and 00 01. The Virtual Machine Additions driver uses the value 00 01. The IsRunningInsideVirtualMachine() API uses the value 01 00.

Another method for detecting VirtualPC relies on the fact that VirtualPC does not limit the length of an instruction. Intel and AMD CPUs have a maximum instruction length of 15 bytes. This is achievable only in 16-bit mode, using the 81 opcode. The instruction would look something like the following:

```plaintext
lock
add dword ptr cs:[eax+ebx+01234567], 89abcdef
```

In addition to the "ADD" instruction, this encoding of the 81 opcode also supports "OR", "ADC", "SBB", "AND", "SUB", or "XOR". The 81 opcode also supports the "CMP" instruction, but it is not permitted in this context because of the "LOCK" prefix.

Any instruction longer than 15 bytes - which is achievable only by the addition of redundant prefixes - will cause a General Protection Fault. However, VirtualPC does not issue this exception, seemingly no matter how long the instruction (see VIRTUALPC ILEN demo).

As noted above, VirtualPC's use of buffered code emulation allows it to intercept instructions that cannot be intercepted by other hardware-bound virtual machine emulators, particularly the hardware-based ones. In theory, the RedPill method could be defeated by intercepting the SIDT instruction, as described in the SubVirt paper. However, this is currently not implemented. The CPUID instruction is one instruction that VirtualPC does intercept. On a real CPU, the returned vendor identification string is either "GenuineIntel" or "AuthenticAMD". In VirtualPC, though, it is "ConnectixCPU", a reference to the company which developed the earlier versions of VirtualPC.

As with VMware, there are many other ways to detect the presence of VirtualPC, including the use of hardware devices with constant names. One detection method is even described by a Microsoft VirtualPC developer. That method queries the name of the manufacturer of the motherboard, which is "Microsoft Corporation" in VirtualPC. Since there can be only one motherboard, the code can be shortened significantly (see VIRTUALPC BOARD demo). However, the problem with this method is that it requires that the Windows Management Instrumentation service is running.

VIII. DETECTING PARALLELS

Parallels is a proprietary, closed-source, reduced privilege guest virtual machine emulator. It supports guest-to-host and host-to-guest communication. It resembles VirtualPC in many ways. Just like VirtualPC, a version exists for the Macintosh platform, as well as for the Windows platform. Only the Windows version is considered here.

Just like VMware and VirtualPC, Parallels must relocate sensitive data structures, such as the IDT and the GDT. Just
like VMWare and VirtualPC, Parallels also makes use of the LDT. Thus, RedPill and LDT work to detect Parallels.

Parallels has two methods of guest-to-host and host-to-guest communication. One of them relies on the execution of an opcode to raise an exception. In this case, the opcode is the BOUND instruction. The difference between the method used by Parallels, and the method used by other virtual machine emulators, is that Parallels uses authentication to determine whether or not the exception is trapped by the kernel.

The method of authentication is to pass in the CPU registers (EAX, ECX, EDX, EBX) values that are specific to the currently executing session. When Parallels first loads the kernel driver, the driver halts the CPU and waits for an interrupt to occur. At that time, the RDTSC instruction is read sixteen times in a row, and the lowest byte is stored in an array that corresponds to those registers. To communicate with the kernel, the guest sets the EBP registers to the string "0x90", and the EDI register contains the index of the function to execute in a function pointer array, and then executes the BOUND instruction with values that are guaranteed to raise the BOUND exception. The main Parallels executable file also uses this method.

```
pushad
mov    esi, [ebp+xxxx]
mov    eax, [esi+4]; load auth value
mov    ebx, [esi+4]; load auth value
mov    ecx, [esi+8]; load auth value
mov    edx, [esi+10h]; load auth value
mov    edi, [esi+10h]; load auth value
mov    esi, [ebp+xxxx]; load real esi
 xor    ebp, ebp
 push    ebp ; upper bound value
 push    ebp ; lower bound value
 mov     ebp, '0x90'
 bound   ebp, [esp]; raise exception
 add     esp, 8 ; discard bound values
popad
```

The second method of guest-to-host and host-to-guest communication occurs through the use of the INT 1B vector. In that case, the registers are initialized in the following way: the ESI register contains the string "magi", the EDI register contains the string "c!nu", and the EBX register contains the string "mber". It spells "magic!number". The EDX register is set to point to any variables on the stack that must be passed, and the EAX register is set to the function number to call. One of the Parallels driver files also uses this method.

```
mov    esi, 'magi'
mov    edi, 'c!nu'
mov    ebx, 'mber'
push   [ebp+xxxx]
push   [ebp+xxxx]
push   [ebp+xxxx]
push   [ebp+xxxx]
push   edx, esp
mov    eax, 0
int    1bh
```

The reason for the two different methods is that the BOUND method is available from user mode, so it must be protected from abuse by non-privileged applications. The INT 1B method is available only from kernel mode, so a user with sufficient privileges to install a kernel-mode driver should presumably have sufficient privileges to communicate with Parallels itself.

In addition, the author found not another way to detect Parallels, but a way to crash it. By entering v86 mode (a Windows DOS box was used) and issuing a SIDT instruction with the Trap flag set, Parallels encounters a fatal error and closes.\(^1\)

IX. DETECTING VIRTUALBOX

VirtualBox is an Open Source, reduced privilege guest virtual machine emulator. It uses a recompiler to perform a dynamic translation of some code to improve performance. This recompiler is based on QEMU, and for that reason it is detected in some of the same ways that the author found. Some of the methods are described in the following:

- CPUID instruction returns wrong value for Easter egg on AMD CPU (see BOCHS and QEMU CPUID_AMD2 demo)

  This code works by executing the CPUID instruction to check for an AMD CPU. If one is found, then the CPUID instruction is executed again to query the Easter egg. For a real AMD K7 processor, the returned value is "IT'S HAMMER TIME". For QEMU, nothing is returned. This detection method is available due to what appears to be an oversight.

- CMPXCHG8B instruction does not always write to memory (see QEMU CMPXCHG8B demo)

  This code works by executing registering a Page Fault handler then executing a CMPXCHG8B instruction on a read-only page. For a real CPU, the CMPXCHG8B instruction always writes to memory, no matter what is the result. For a read-only page, a Page Fault will be raised. For QEMU, no Page Fault occurs. This detection method is available due to what appears to be an oversight.

- Double Fault exception is not supported (see QEMU EXC_DBL demo)

  This code begins by setting the limit of the IDT less than what is required to describe the General Protection Fault handler. Then a General Protection Fault is raised. For a real CPU, being unable to raise the General Protection Fault causes the Double Fault exception to be raised. For QEMU, the General Protection Fault is raised repeatedly. This detection

\(^1\) The vendor was notified, but did not respond after sixty days.
method is available due to a limitation in the exception handling code.

SECTION 2: SOFTWARE

Pure software virtual machine emulators are also vulnerable to detection. In their case, detection is possible mostly because of software bugs or incomplete support for the CPU which is being emulated.

X. DETECTING BOCHS

Bochs is an Open Source, pure software virtual machine emulator. It does not support guest-to-host or host-to-guest communication since it is intended to behave like a stand-alone machine. It is vulnerable to a number of detection methods. The simplest of these involves the device support. For example, Bochs cannot handle floppy disks of non-standard sizes. Attempting to format a 3.5" floppy disk with more than 18 sectors per track, or with sectors other than 512 bytes in size, will cause a kernel panic. As with VMware and VirtualPC, Bochs has constant names for its hardware devices, but again, the presence of these devices cannot be relied upon. Thus, we are left with the CPU as the target for detection. The author discovered a number of methods to detect Bochs. Here are some of them:

- **INVD** and **WBINVD** instructions always flush TLBs (see **BOCHS WBINVD** demo)

  The code works by entering paging mode, and then accessing a page. This causes the CPU to place the page's physical address into one of the Translation Lookaside Buffers. When an **INVD** or **WBINVD** instruction is executed inside Bochs, the Translation Lookaside Buffers are flushed. Hence, if the same page is marked "not present" then accessed again, a Page Fault occurs. By registering a Page Fault handler prior to executing the **INVD** or **WBINVD** instruction, Bochs can be detected. This detection method is available due to what appears to be an oversight.

- **CMPS** instruction flags are not retained while **REP** continues in single-step mode (see **BOCHS CMPS** demo)

- **SCAS** instruction flags are not retained while **REP** continues in single-step mode (see **BOCHS SCAS** demo)

  These two codes begin by setting the carry flag. Then, in the case of the **CMPS** instruction, two ranges of bytes that are known to be identical are compared (the source and destination registers are set to the same value). In the case of the **SCAS** instruction, a single byte, whose value is known to match the destination, is compared to the destination. The source register is set to the value in memory that is pointed to by the destination register. In a real machine, the carry flag remains set until the **REP** has completed. However, in Bochs, the flag is updated immediately. By registering a trap handler prior to executing the **CMPS** or **SCAS** instruction, the carry flag can be seen to have been cleared, and thus Bochs can be detected. This detection method is available due to what appears to be an oversight.

- **CPUID** instruction returns wrong value for processor name on **AMD** CPU (see **BOCHS CPUID_AMD1** demo)

  This code works by executing the **CPUID** instruction to check for an **AMD** CPU. If one is found, then the **CPUID** instruction is executed again to query maximum input value for the extended **CPUID** information. If the processor brand string is supported, then the **CPUID** instruction is executed again to query the processor brand string. For a real **AMD K7** processor (the only one that Bochs supports), the returned string is "AMD Athlon(tm) P[rocessor]". For Bochs, it is "AMD Athlon(tm) P[rocessor]" (note the lowercase 'p'). This detection method is available due to what appears to be an oversight.

- **CPUID** instruction returns wrong value for Easter egg on **AMD** CPU (see **BOCHS and QEMU CPUID_AMD2** demo)

  This code works by executing CPUID to check for an **AMD** CPU. If one is found, then the **CPUID** instruction is executed again to query the Easter egg. For a real **AMD K7** processor (the only one that Bochs supports), the returned value is "IT'S HAMMER TIME". For Bochs, nothing is returned. This detection method is available due to what appears to be an oversight.

- **ARPL** instruction destroys upper 16 bits of 32-bit register in 32-bit mode (see **BOCHS ARPL** demo)

  This code executes the **ARPL** instruction using the undocumented 32-bit register mode. Officially, the instruction accepts 16-bit registers. For some reason, Bochs ORs the top 16 bits with 0f3f0000h, but the author found no real CPU where that behavior occurs. This detection method is available due to what appears to be an oversight.

- 16-bit segment wraparound is not supported (see **BOCHS and HYDRA SEGLOAD** demo)

---

This list is the longest in this paper because Bochs was the first application to be examined, and received the most scrutiny. It does not reflect the quality of the software.
This code executes a segment:register load, at an offset where the register part is at a lower address than is the segment part. By registering a trap handler prior to executing the load instruction, an exception will occur in Bochs that should not occur at all. Thus Bochs can be detected. This detection method is available due to what appears to be an oversight.

- Non-ring0 SYSENTER CS MSR causes kernel panic

This is similar to the v86 SIDT problem in Parallels, in that it is not a method to detect Bochs, but a way to crash it. By simply writing to the SYSENTER CS MSR (174h) a value with any of the low two bits set, Bochs will encounter a kernel panic and close. A real CPU will accept this value since no checks are done until the SYSENTER instruction is actually executed. This detection method is available due to what appears to be an oversight.

XI. DETECTING HYDRA

Hydra is a proprietary, closed-source, pure software virtual machine emulator. It supports guest-to-host communication, even though it is intended to behave like a stand-alone machine. It does not intentionally support host-to-guest communication. The guest-to-host communication channel exists for the use of plug-ins that can alter the environment and control the execution flow. However a plug-in is not supposed to communicate with the guest. Hydra also uses a special port for guest-to-host communication, much like VMware does. The key differences between VMware and Hydra are that in Hydra, the port to use is specific to the plug-in; and a plug-in can still cause an exception to be generated, thus better hiding the interaction. Since no host-to-guest communication occurs, no Hydra-specific information is returned by the port access. In any case, the author discovered a number of methods to detect Hydra. Some of the methods are described in the following:

- REP MOVS instruction integer overflow (see HYDRA MOV demo)

- REP STOS instruction integer overflow (see HYDRA STOS demo)

This code works by causing a loop counter to overflow, when converting from a dword count to a byte count. Thus no bytes are copied (in the case of the MOVS instruction) or stored (in the case of the STOS instruction). This leads the emulator to believe that an error occurred, so a General Protection Fault is raised. In the absence of a General Protection Fault handler, a Double Fault occurs. In the absence of a Double Fault handler, a Triple Fault occurs, leading to the emulator exiting completely. This detection method is available due to a limitation in the string acceleration code.

- 16-bit segment wraparound is not supported (see BOCHS and HYDRA SEGWRAP demo)

This code executes a segment:register load, at an offset where the register part is at a lower address than is the segment part. By registering a trap handler prior to executing the load instruction, an exception will occur in Hydra that should not occur at all, and thus Hydra can be detected. This detection method is available due to what appears to be an oversight.

XII. DETECTING QEMU

QEMU is an Open Source, pure software virtual machine emulator. It does not support guest-to-host or host-to-guest communication since it is intended to behave like a stand-alone machine. It supports dynamic translation of code to improve the performance on the supported CPUs. The use of dynamic translation is always risky in the presence of self-modifying code, especially when non-intuitive CPU behavior occurs, such as a self-overwriting REP sequence. The author discovered a number of methods to detect QEMU. Some of the methods are described in the following:

- CPUID instruction returns wrong value for processor name on AMD CPU (see QEMU CPUID_AMD demo)

This code works by executing the CPUID instruction to check for an AMD CPU. If one is found, then the CPUID instruction is executed again to query maximum input value for the extended CPUID information. If the processor brand string is supported, then the CPUID instruction is executed again to query the processor brand string. For a real AMD K7 processor, the returned string is "AMD [processor name] Processor". For QEMU, it is "QEMU Virtual CPU version x.x.x.x".

4 The REP instruction is handled specially by x86 CPUs, such that it completes even if the sequence is replaced in memory. For example,

```
mov al, 90h
mov cx, 7
mov di, offset $
rep stosb
jmp $
```

Here, the NOP instruction in the AL register is used to overwrite the REP STOSB and the following JMP instruction. Incorrect emulation (or single-stepping through the code, as with a debugger) will cause the REP to exit prematurely, resulting in the JMP instruction being executed.

3 All of the the problems described here have since been fixed.
• **CPUID** instruction returns wrong value for Easter egg on AMD CPU (see BOCHS and QEMU CPUID_AMD2 demo)

This code works by executing the CPUID instruction to check for an AMD CPU. If one is found, then the CPUID instruction is executed again to query the Easter egg. For a real AMD K7 processor, the returned value is "IT'S HAMMER TIME". For QEMU, nothing is returned. This detection method is available due to what appears to be an oversight.

• **CMPXCHG8B** instruction does not always write to memory (see QEMU CMPXCHG8B demo)

This code works by executing registering a Page Fault handler then executing a CMPXCHG8B instruction on a read-only page. For a real CPU, the CMPXCHG8B instruction always writes to memory, no matter what is the result. For a read-only page, a Page Fault will be raised. For QEMU, no Page Fault occurs. This detection method is available due to what appears to be an oversight.

• Double Fault exception is not supported (see QEMU EXC_DBL demo)

This code begins by setting the limit of the IDT less than what is required to describe the General Protection Fault handler. Then a General Protection Fault is raised. For a real CPU, being unable to raise the General Protection Fault causes the Double Fault exception to be raised. For QEMU, the General Protection Fault is raised repeatedly. This detection method is available due to a limitation in the exception handling code.

XIII. DETECTING ATLANTIS AND SANDBOX

Since both Atlantis and Sandbox emulate only a subset of all of the possible Windows APIs, and of those, some of the APIs do not behave in the same way as on a real machine. Thus, they are vulnerable to detection through the use of any unimplemented API or any API that is not emulated correctly. An example is the Beep() API, which has limitations on the frequency of the sound to produce when executed on Windows NT and later versions of Windows. Atlantis does not check that parameter since it emulates Windows 9x. Thus, it returns no error, no matter what value is specified. Any program that assumes it is running on Windows NT or later will know immediately if Atlantis is hosting the environment, by calling that API with an illegal value. Another example is through the use of an exploit. There are several current documented denial-of-service vulnerabilities in different versions of Windows for the Windows Meta File (WMF) format. If such a malformed WMF file is played successfully, then an operating system emulator is running. A detailed list of methods to detect Sandbox follows.

XIV. DETECTING SANDBOX

Sandbox is a proprietary, closed-source, pure software virtual machine and operating system emulator. Though it is a retail product, copies of it are freely available on many P2P sites. For some reason, Sandbox places the IDT in a very high memory location, and the LDT has a non-zero value. For those reasons, RedPill and LDT work to detect Sandbox.

The CPU supported by Sandbox seems to be a partial implementation of an Intel Pentium 2, however some Pentium 2 instructions such as FXSAVE are not supported, nor are some Pentium 1 instructions such as RDMSR or CMPXCHG8B. These instructions will cause exceptions in Sandbox, which can be used to detect its presence.

Strangely, despite the supported processor, the ID flag is not set in the EFLAGS register. Despite this, the CPUID instruction causes no exceptions. However, index 0 returns a bad Basic Processor Information value and Vendor Identification String.

The author discovered a number of methods to detect Sandboxes. Here are some of them:

• EFLAGS.bit 1 is clear by default and can be toggled

On a real CPU, this bit is always set and read-only.

• GetVersionExA() returns inconsistent information

This API returns the platform identification value that corresponds to Windows 2000, but the IDT is readable from ring 3, and certain interrupts point to 0c0xxxxxx space, which reflects Sandbox’s Windows 9x origins.

• the first KERNEL32 export is named “Aaaaaa” and matches the Windows 9x/Me NvDCall code

• IDT and GDT limits contain incorrectly aligned values

On a real system, the IDT and GDT limits are one less than the size of the table (i.e. a limit of 256 has a value of 255). On Sandbox, the values are exactly the size of the table.

• GDT base is in low memory

• vulnerable to self-overwriting REP, as described in the QEMU footnote

• CMPXCHG does not always write to memory
This is identical to the detection of QEMU, but using a slightly different instruction.

- int 2a instead of GetTickCount

Sandbox generates an exception when this interrupt is issued.

Since Sandbox does not support emulation of real mode, no source code is included to illustrate detection methods.

XV. DETECTING CWSANDBOX

As a special request, CWSandbox was analyzed by the author. CWSandbox is a proprietary, closed-source, application-level sandbox. As with Dynamo Rio, CWSandbox hooks some operating system APIs, but otherwise allows an application to run on the real hardware. The documentation states "...a lot of effort has been put into hiding the presence of the CWSandbox and the injected CWMonitor.DLL from the malware", however those efforts are ineffective. For example, the author found several global objects, such as a mutex called "cws_[pid] mutex" (where "[pid]" is the process ID of the targeted application), two events called "cws_[pid] event_data" and "cws_[pid] event_result", and a file mapping called "cws_[pid]_mapping". The API hooking consists of "ff 25"-style trampolines for 290 APIs and 10 methods (see Appendix B for the full list). Escape from the environment is simply a matter of calling FreeLibrary(GetModuleHandleA("cwmonitor")) to unload the DLL.

XVI. MISCELLANEOUS DETECTIONS

Following the publication of the original version of this paper, the author conducted further research on the low-level behavior of the CPU. Two very interesting things were noted. The first is operating-system specific. It detects hybrid models such as Atlantis and Sandbox.

The second is hardware-specific, and is actually a set of four different behaviors. The first hardware-specific behavior - fault while fetching - detected only Hydra. The reason for that is because the hardware performs a fetch and full decode in parallel, before testing if an opcode is invalid. However, for performance reasons, Hydra performs the test first, to avoid full decode.

The second hardware-specific behavior is the undocumented opcodes in the range 0f 19-1e. They are identical to 0f 1f (multi-byte NOP), but both Bochs and Sandbox raise an exception when those instructions are executed.

The third hardware-specific behavior is the undocumented opcode maps for the opcodes 0f 20-23, using MODR/M values below 0e0. Sandbox raises an exception when these values are used.

The fourth hardware-specific behavior is the undocumented opcode maps for the opcodes 0f 18 2x-3x, and 0f 1f. Both Bochs and Sandbox raise an exception when those instructions are executed.

XVII. CONCLUSION

So what can we do? The answer to this question depends on the application that is being used. However, for the reduced privilege guest virtual machines emulators, the ultimate answer is "nothing". The problem for them is that their design does not allow them to intercept non-sensitive instructions that cause information leakage, such as the SIDT instruction. As a result, they cannot hide their presence from the RedPill, LTD, and Scooby Doo, attacks.

The Liston/Skoudis paper has a title that suggests that they can reduce the ability of software to detect virtual machine emulators. However, it is actually more concerned with ways to detect virtual machine emulators. The recommendations in that paper for reducing the ability of software to detect virtual machine emulators are exclusively for VMware, and insufficient, as noted earlier.

VirtualPC could be improved to intercept the SIDT instruction. This would go a long way towards hiding its presence, but it would also need to implement a check for the maximum instruction length.

The interception of the CPUID instruction in both VirtualPC and QEMU to replace the processor identification string should be removed, too.

The use of session key authentication to control guest-to-host and host-to-guest communication in Parallels is a good idea that other applications could use.

Bochs, Hydra, QEMU, Sandbox, and VirtualBox, all suffer from bugs and limitations that allow their detection. These are problems that are relatively easily fixed. Given that, only pure software virtual machine emulators can approach complete transparency. It should be possible, at least in theory, to reach the point where detection is unreliable because it can also be attributed to anomalous behavior of a real CPU (for example, the f0 0f bug). We might call that "virtual reality".

On the other hand, if a majority of future machines run a virtual machine emulator, then malicious code that chooses to not run in its presence will eventually be unintentionally choosing to not run at all.

Once that point is reached, the attacks will move from detection to exploitation. The ultimate attack against a hypervisor would be to run arbitrary code inside it. Along those lines, in February a privilege escalation exploit was published for the hypervisor in Microsoft's Xbox 360 platform. The exploit code took advantage of improper
parameter validation to execute arbitrary code with the privileges of the hypervisor itself.

One thing is clear – the future looks complicated.

APPENDIX A

VIRTUALPC TIME DEMO:

.model  tiny
.code
org 100h
demo: mov ax, 2506h
       mov dx, offset int06
       int 21h
db 0fh,3fh,3,0
       jmp $ ;detected
int06: int 20h
.end demo

VIRTUALPC ILEN DEMO:

.model  tiny
.code
org 100h
demo: mov ax, 250dh
       mov dx, offset int0d
       int 21h
       db 0eh dup (2eh)
       jmp $ ;detected
int0d: int 20h
.end demo

VIRTUALPC BOARD DEMO:

For Each board in GetObject("winmgmts:\\.oot\cimv2").ExecQuery("Select * from Win32_BaseBoard") If board.Manufacturer = "Microsoft Corporation" then while 1 : wend 'detected Next

BOCHS WBINV DEMO:

.model  tiny
.486p
.code
org 100h
demo: mov edx, ds
       mov cx, 100h
       movzx eax, cx
       add ah, dh
       mov es, ax
       shl eax, 4
       mov cr3, eax
       shl edx, 4
       mov bx, offset gdt
       add [bx + 2], edx
       mov [bx + 0ah], dx
       add [bx+offset idtr-offset gdt+2],edx
       bswap edx
       mov [bx + 0ch], dh
       mov [bx + 0fh], dl

BOCHS CMPS DEMO:

.model  tiny
.code
org 100h
demo: mov ax, 2501h
       mov dx, offset int01
       int 21h
       mov cx, 10h
       mov si, cx
       mov di, cx
       push cx
       popf
       repe cmpsb
int01: jnb $ ;detected
       int 20h
.end demo

BOCHS SCAS DEMO:
```
.model tiny
.code
org 100h
demo:    mov    ax, 2501h
          mov    dx, offset int01
          int    21h
          mov    cx, 101h
          mov    di, cx
          push   cx
          popf
          repe  scasb
          int01:  jnb    $; detected
          int    20h
end     demo

BOCHS CPUID_AMD1 DEMO:

.model tiny
.586
.code
org 100h
demo:    xor    eax, eax
          cpuid
          cmp    ecx, 444d4163h
          jne    exit
          mov    eax, 80000000h
          cpuid
          cmp    eax, 2
          jb     exit
          mov    eax, 80000002h
          cpuid
          shr    edx, 1eh
          jb     $; detected
exit:    ret
end     demo

BOCHS and QEMU CPUID_AMD2 DEMO:

.model tiny
.586
.code
org 100h
demo:    xor    eax, eax
          cpuid
          cmp    ecx, 444d4163h
          jne    exit
          mov    eax, 80000000h
          cpuid
          shr    edx, 1eh
          jb     $; detected
exit:    ret
end     demo

BOCHS ARPL DEMO:

.model tiny
.486p
.code
org 100h
demo:    mov    eax, ds
          shl    eax, 4
          mov    bx, offset gdt
          add    [bx + 2], eax
          mov    [bx + 0ah], ax
          bswap  eax
          mov    [bx + 0ch], ah
          mov    [bx + 0fh], al
          cli
          lgdt    [bx]
          mov    eax, cr0
          inc     ax
          mov    cr0, eax
          cdq
          push   cs
          push   dx
          push    8
          push   offset pmode
          retf
          pmode   db  66h
          arpl    dx, ax
          test    edx, edx
          js      $; detected
          dec     ax
          mov    cr0, eax
          retf
gdt     dw     offset gdt_e - offset gdt - 1
          dw     offset gdt
don    dd  0
          dd     0ffffh
          dd     9b00h
          dd     0ffffh
          dd     0cf9300h
          gdt_e:
end     demo

BOCHS and HYDRA SEGLOAD DEMO:

.model tiny
.486p
.code
org 100h
demo:    mov    ax, 250dh
          mov    dx, offset int0d
          int    21h
          lds    ax, ds:[0fffeh]
          ret
          int0d:  jmp    $; detected
end     demo

HYDRA MOV'S DEMO:

.model tiny
.486p
.code
org 100h
demo:    mov    edx, ds
          mov    cx, 1000h
          movzx   eax, cx
          add     ah, dh
          mov     es, ax
          shl     eax, 4
          mov     cr3, eax
          shl     edx, 4
          mov     bx, offset gdt
          add     [bx + 2], edx
          mov     [bx + 0ah], dx
          add     [bx+offset idtr-offset gdt+2],edx
          bswap   edx
          mov     [bx + 0ch], dh
          mov     [bx + 0fh], dl
          add     eax, 1007h
          xor     di, di
          stosd
          push    7
```
HYDRA STOS DEMO:

QEMU CPUID_AMD DEMO:
QEMU CMPXCHG8B DEMO:

.model tiny
.model .586p
.code
org 100h
demo: mov edx, ds
mov cx, 1000h
movzx eax, cx
add ah, dh
mov es, ax
shl eax, 4
mov cr3, eax
shl edx, 4
mov bx, offset gdt
add [bx + 2], edx
mov [bx + 0ah], ax
add [bx+offset idtr-offset gdt+2],eax
bswap eax
mov [bx + 0ch], ah
mov [bx + 0fh], al
add [bx+offset idtr-offset gdt+2],dx
sidt fword ptr [offset idt_end]
ldt [bx + offset idtr - offset gdt]
lgdt [bx]
mov ax, cr0
inc ax
mov cr0, eax
int 3
int03: int 0ffh
int08: dec ax
mov cr0, eax
sand fword ptr [offset idt_end]
mov ah, 4ch
int 21h
create_tbl:
    stosd
    add eax, 1000h
    loop create_tbl
mov fs, cx
cli
sidt fword ptr [offset idt_end]
ldt [bx + offset idtr - offset gdt]
lgdt [bx]
mov eax, cr0
mov ecx, eax
or eax, 80010001h
mov cr0, eax
int 3
int03: mov byte ptr es:[1004h], 5
mov al, fs:[1000h]
inc ax
jmp $ ; detected
int08: mov cr0, ecx
ldt fword ptr [offset idt_end]
mov ah, 4ch
int 21h
mov es:[1000h]
op 0
mov 9b00h
idtr dw offset idt_end - offset idt - 1
idt dd 6 dup (0)
dd offset int03
dd 86000008h
dd 0
dd 8 dup (0)
dd offset int08
dd 86000008h
dd 0
idt_end:
en d demo

QEMU EXC_DBL DEMO:

.model tiny
.model .486p
.code
org 100h
demo: mov eax, ds
shl eax, 4
mov bx, offset gdt
add [bx + 2], eax
mov [bx + 0ah], ax
add [bx+offset idtr-offset gdt+2],eax
bswap eax
mov [bx + 0ch], ah
mov [bx + 0fh], al
add [bx+offset idtr-offset gdt+2],dx
sidt fword ptr [offset idt_end]
ldt [bx + offset idtr - offset gdt]
lgdt [bx]
mov eax, cr0
inc ax
mov cr0, eax
int 3
int03: int 0ffh
int08: dec ax
mov cr0, eax
sand fword ptr [offset idt_end]
mov ah, 4ch
int 21h
create_tbl:
    stosd
    add eax, 1000h
    loop create_tbl
mov fs, cx
cli
sidt fword ptr [offset idt_end]
ldt [bx + offset idtr - offset gdt]
lgdt [bx]
mov eax, cr0
mov ecx, eax
or eax, 80010001h
mov cr0, eax
int 3
int03: mov byte ptr es:[1004h], 5
mov al, fs:[1000h]
inc ax
jmp $ ; detected
int08: mov cr0, ecx
ldt fword ptr [offset idt_end]
mov ah, 4ch
int 21h
mov es:[1000h]
op 0
mov 9b00h
idtr dw offset idt_end - offset idt - 1
idt dd 6 dup (0)
dd offset int03
dd 86000008h
dd 0
dd 8 dup (0)
dd offset int08
dd 86000008h
dd 0
idt_end:
en d demo

APPENDIX B

APIs hooked by CWSandbox:

- KERNEL32.LoadLibraryExW
- ICMP.IcmpSendEcho
- ICMP.IcmpSendEcho2
- MPR.WNetAddConnectionA
- MPR.WNetAddConnectionW
- MPR.WNetAddConnection2A
- MPR.WNetAddConnection2W
- MPR.WNetAddConnection3A
- MPR.WNetAddConnection3W
- MPR.WNetCancelConnectionA
- MPR.WNetCancelConnection2A
- MPR.WNetCancelConnection2W
- MPR.WNetCancelConnection3A
- MPR.WNetCancelConnection3W
- MPR.WNetCancelConnection4A
- MPR.WNetCancelConnection4W
- MPR.WNetCancelConnection5A
- MPR.WNetCancelConnection5W
- MPR.WNetOpenEnumA
- MPR.WNetOpenEnumW
- NETAPI32.NetScheduleJobAdd
Methods hooked by CWSandbox:

Methods hooked by CWSandbox:

- IPStore.QueryInterface()
- IPStore.EnumTypes()
- IPStore.EnumSubtypes()
- IPStore.DeleteItem()
- IPStore.ReadItem()
- IPStore.WriteItem()
- IPStore.OpenItem()
- IPStore.EnumerateItems().Clone()
Bugcheck
Detecting hardware assisted hypervisors
Peter Ferrie
"Detecting hardware-assisted hypervisors without external timing"
Kevin Lawton et al
http://bochs.sourceforge.net
Fabric Bellard
http://fabrice.bellard.free.fr/qemu
Peter Ferrie
http://pferrie.tripod.com/#atlantis
Norman
http://www.norman.com
Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch
http://www.eecs.umich.edu/virtual/papers/king06.pdf
Joanna Rutkowska
"Subverting Vista Kernel For Fun and Profit"
Dino A. Dai Zovi
"Hardware Virtualization Rootkits"
http://www.whiteacid.org/misc/bh2006/036_Zovi.pdf
Danny Quist and Val Smith
"Detecting the Presence of Virtual Machines Using the Local Data Table"
http://www.offensivecomputing.net/files/active/0/vm.pdf
Joanna Rutkowska
"Red Pill"
http://invisiblethings.org/papers/redpill.html
Tobias Klein
"Scooby Doo - VMware Fingerprint Suite"
http://www.trapkit.de/research/vmm/scoopydoo/index.html
Tobias Klein
"jerry - (another) VMware Fingerprinter"
http://www.trapkit.de/research/vmm/jerry/index.html
Peter Ferrie
"Tumours and Polips"
http://pferrie.tripod.com/vb/polip.pdf
Ken Kato et al
"VMware Backdoor I/O Port"
http://chitchat.at.infoseek.co.jp/vmware/backdoor.html
Tim Shelton
Ben Armstrong
"Detecting Microsoft virtual machines"
Peter Ferrie
"Inside the Windows Meta File Format"
http://pferrie.tripod.com/vb/wmf.pdf
ReWolf
"Int 2Ah – KiGetTickCount"
CWSandbox
http://www.sunbelt-software.com/Developer/Sunbelt-CWSandbox/
Peter Ferrie
“Attacks on Virtual Machines”
Peter Ferrie
“Locked and Loaded”
Peter Ferrie
“x86 Fetch-Decode Anomalies”
Tom Liston and Ed Skoudis
"On the Cutting Edge: Thwarting Virtual Machine Detection"
http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf
Robert R. Collins
"The Intel Pentium F00F Bug Description and Workarounds"
http://www.x86.org/errata/dec97/f00fbug.htm