Hacking the Preboot eXecution Environment

Using the BIOS network stack for other purposes

Julien Vanegue
jfv@cesar.org.br

CESAR
Recife Center for Advanced Studies and Systems, Brasil.

September 27, 2008

Abstract

It is well known that most modern BIOS come with a Preboot eXecution Environment (PXE) featuring network capabilities. Such a PXE stack allows computers to boot on the network using standard protocols such as DHCP and TFTP for obtaining an IP address and downloading an Operating System image to start with. While some security researchers have been exposing the insecurity of using kernel images downloaded in such way, it is less known that the PXE stack remains usable even after the operating system has started. This article describes how to come back to real-mode from a protected mode environment in order to use the PXE stack for receiving and sending any kind of packets, and come back to protected mode as if nothing had happened. The proposed proof of concept implementation allows to send and receive UDP packets. It is explained in great details and was tested on Intel Pentium 4 and AMD64 processors on a variety of five different BIOS.

1 Introduction

The PXE protocol version 2.1 [1] was defined by Intel Corp. in 1999. It has now become a widely used standard in any modern BIOS environment for allowing network capabilities since the very first moment of code execution on Intel processors based computers. It is common knowledge that PXE capabilities can be used for sending and receiving **DHCP** and **TFTP** packets over the LAN. However, the **PXE API** allows much more than this, as any UDP packet can be sent over the network. Even more, some particular PXE calls (such as UNDI PACKET TRANSMIT) canbe used to send any packet on the network, including **IP, ARP, RARP** packets. Additionally, broadcast and multicast packets can also be sent to the network. This opens the door to the highest performance, OS-independent access to the network, such as the development of network scanners, Intrusion Detection (or Prevention) Systems, and so on. Such features can be developed independently of the network card, as the PXE protocol is independent of the hardware layer.

It is worth noting that, while PXE does not provide any API for receiving packets others than UDP, any received packets has to be written in memory whatever protocol is used. Such behavior can be used to receive any kind of packets only using a PXE stack. Such packets are written in the data segment of the PXE stack before either being discarded or passed to the program running on top of the PXE stack. Additionally,the treatment of packets on top of PXE can be made more efficient using an integrated PXE packet filter (as made accessible from the **PXE_SET_PACKET_FILTERAPI**).

The purpose of this paper is neither about describing any particular tool derived from such concept, nor developing a rootkit downloadable from PXE, but rather showing the low-level implementation mechanisms involved in using the PXE stack outside its original scope, even when the operating system is already in execution. As such, our bricks of low-level code can be used in another wider project handling all the high-level data management. The article is organized as follow: we first explain our proof of concept code step by step. This code involves coming back to real mode from protected mode, calling the PXE stack, and coming back to protected mode. Before concluding, we mention some security related works about abusing the PXE stack.

2 The code explained

Our code is made of around 400 assembly instructions. The algorithm first comes back from protected mode to real mode. At such step, we must avoid **unreal mode** (aka 32 bits real-mode) since the BIOS and PXE functions of interest to us have to be called from a 16 bits real-mode environment. The PXE reference teaches us how to use the PXE stack API. Sending a packet consists in calling only two functions. Finally, we need to come back to the original protected mode environment we had previously left.

2.1 Switching back to 16 bits real-mode

Coming back to real-mode from protected mode is documented in the INTEL reference manual volume 3 [1]. While such procedure is well known, we find important to recall what unreal mode is. When coming back from protected mode, disabling the **PE** bit in the **cr0** register will not come back to 16 bits real mode. Indeed, the segment descriptor caches will remain used by the processor. Generally, protected mode segments are 32 bits. This means that even after disabling the PE bit, the processor will continue interpreting the code as 32 bits code. This is not good for us since BIOS functions are compiled in 16 bits. Executing a 16 bits code from a 32 bits segment will result in wrong instruction fetching and mostly likely crashes the machine. Instead, we need to setup a new Global Descriptor Table (GDT) with 16 bits segments and swap the GDTR register to use this new GDT. Once this is done, we can disable the PE bit and enter in 16 bits real-mode.

Lets now dig into the details of our pxe-reload function. We assume that an alternative GDT has been setup:

```
.balign 8
real_gdt:
.long 0
.long 0                    /* First descriptor must always be 0 */
.long 0x0000FFFF           /* Code segment limit suitable for real-mode */
.long 0x00009B03           /* Code segment, 16 bits, base 0x30000 */
.long 0x0000FFFF           /* Data segment limit suitable for real-mode */
.long 0x000F9300           /* Data segment, 16 bits, base 0 */

real_gdt_desc:
.word 0x18                 /* Number of bytes in GDT */
.long unreal_gdt           /* Pointer on GDT */
```

Table 1: Creating the GDT for switching back to real-mode

First we need to save the original context of protected mode:

```
pxe-reload:
pushf
push %eax
push %ebx
push %ecx
push %edx
push %esi
push %edi
push %ebp
```

Table 2: Saving protected mode context

Once this is done, we are ready to restore the real-mode settings for interrupt handling.

2.1.1 Handling Interrupt 1A

It is generally not possible to call the BIOS without restoring the interrupt table. Unlike the Interrupt Descriptor Table (**IDT**) used in protected-mode, real-mode uses an Interrupt Vector Table (**IVT**) whose format is slightly different. Each entry of the **IVT** is 32 bits long. It contains a 16 bits real-mode segment selector and a 16 bits offset. The address of the interrupt handler is then given by computing **(seg ¡¡ 4) + off** (thus real-mode can only address the first 1MB of RAM over 20 bits and each memory segment is 64KB).

Again, we assume that proper data for the IVT has been setup:

```
IVT: .skip 128                  /* Space for 32 entries of real-mode IVT */
rm_idt:
.word 0x80                      /* Size of IVT */
.long IVT                       /* Address of IVT */
```

Table 3: Allocate memory for the real-mode Interrupt Vector Table

In our case, we only care about restoring a usable handler for interrupt 0x1A. Such interrupt is responsible for all PXE events. We declare a dummy interrupt handler that we will use for all interrupts except *int 0x1a*:

```
int handler: iret
```

Table 4: Universal interrupt handler while being in real-mode. Very subtle

The code for setting up such IVT is following:

```
ivt_prepare:
movl $32, %ecx                  /* We need to copy 32 entries */
lea int_handler, %eax           /* Our default handler stands in EAX */
andl $0x0000FFFF, %eax          /* We isolate the segment offset for it */
movl $0x30000000, %edx          /* Assume IVT segment base at 0x3000 */
movw %ax, %dx                   /* Compute real-mode int handler address */
lea IVT, %ebx                   /* Put IVT address in EBX */
movl (SAVED_1AOFF), %edi        /* Handler for int 1A stands in EDI */

ivt init:
cmp $0x0006, %ecx               /* Are we setting handler for int 0x1A ? */
jne ivt_voidhandler             /* If not, simply write the default handler */
movl %edi, (%ebx)               /* Else, write the int 1A address in the current IVT slot */
jmp ivt_init_next               /* Switch to the next entry */

ivt_voidhandler:
movl %edx, (%ebx)               /* If not 1A, put default handler in current slot */
ivt_init_next: add $4, %ebx     /* Switch to the next IVT entry */
loop ivt_init                   /* Loop until we have filled 32 entries */
```

Table 5: Initialize the real-mode IVT with empty handlers, except for interrupt 1A

There is no restriction on where the IVT is standing in memory as long as it is inthe first 1MB of memory (as explained earlier). We are now ready to swap GDT.

2.1.2 Setting up 16 bits protected mode GDT

Changing GDT to come back to 16 bits real mode is quite cumbersome. Not only we need to change the content of the GDTR and IDTR registers to make them point on the new GDT and IDT (after saving the original values), but we also need to flush the current segments descriptors cache to make sure the new GDT is actually used.

This is realized using a long ret as explained below:

```
mov %esp, (SAVED SP)          /* Save pmode stack pointer */
mov %ss, (SAVED SS)           /* Save pmode stack segment */

switch_to_rmode: cli          /* Disable interrupts */
sidt saved_idt                /* Save pmode IDT */
lidt rm_idt                   /* Switch to rmode IDT */
sgdt saved_gdt_desc           /* Save pmode GDT */
lgdt real gdt desc            /* Set GDT for unreal mode */
jmp next16                    /* Empty prefetch queue */
next16: call flush16          /* Retreive address of current instr */
flush16: pop %eax             /* Put it in EAX */
add $0x10, %eax               /* Make EAX points after lret */
and $0x0000FFFF, %eax         /* Only keep the segment offset in EAX */
.code16 .byte 0x66            /* Force a 16 bits push */
pushw $0x8                    /* Push segment selector for new CS */
.byte 0x66 pushw %ax          /* Push segment offset where to come back */
.byte 0x66 lret               /* Flush the CS descriptor cache ! */

xorl %eax, %eax               /* EAX = 0 */
add $0x10, %ax                /* Set AX to the value of the data segment selector */
movw %ax, %ds                 /* Change all data segment registers */
movw %ax, %es
movw %ax, %ss
movw %ax, %fs
movw %ax, %gs
```

Table 6: Switch to 16b protected mode

We are now executing in 16 bits protected mode. This will allow us to leave protected mode directly in 16 bits in order to come back to a 16 bits real mode andnot a 32 bits real mode (unreal mode). This step is mandatory. If not performed, we would end up calling the PXE routines in 32 bits, which would not work as expected.

2.1.3 Leaving protected mode

After we have switched to a 16-bits protected mode, we can now switch to 16 bits realmode only by disabling the PE bits and setting the segment registers to real-mode values. In such proof-of-concept, we have chosen to put the base address of all our segments to **0x3000** except %gs which has to hold the base address of one of the BIOS data segment (**0xF000**).

This code ends up the change from 32 bits protected mode to 16 bits real mode. Now the serious things can start as we will be able to call the PXE stack to send network packets.

2.2 Remapping the PXE stack

It is highly probable that the PXE stack has been unloaded from memory after the operating system has booted. It means that we need to restore it at its original address before being able to call anything. The PXE stack is considered as an extension of the BIOS and mapped right below it. The BIOS is generally mapped from A000:0000 to FFFF:FFFF (1MB limit). On all tested configurations, the PXE stack stands between 8C00:0000 and A000:0000. This means that 80K of memory must be restored before calling any routine of the PXE stack. In practice, this amount can be reduced since there are many holes in the PXE stack address range which are simply filled with NUL bytes. For simplicity, we keep copying back a 80K slice taken from the early stage of booting of the machine, when PXE is still mapped and enabled. When debugging a pre-boot code with gdb, such image can be obtained using the dump memory pxe.img 0x8C0000 0xA00000 command (remember that gdb treats all addresses as linear and does not handle the **seg:off** syntax for real-mode addresses).

Another solution is to compile a Etherboot [5] image for replacing the original PXEstack. However, this will means that the PXE stack initialization function has to be executed again before any packet can be sent or received over the network. A fresh **etherboot** PXE stack has to go throught its initialisation stage consisting of calling various PXE services such as *PXENV_START_UNDI*, *PXENV_UNDI_STARTUP*, *PX-*

```
mov %cr0, %eax                  /* We cant modify cr0 directly */
and $0xFE, %al                  /* Disable PE bit */
mov %eax, %cr0                  /* Put the new value */
jmp cont1                       /* Empty the prefetch queue */
cont1: call aftr                /* Get current address */

aftr: pop %eax                  /* Put it in EAX */
add $0xA, %ax                   /* Make EAX points after LRET */
push $0x3000                    /* Assume rmode segment base 0x3000 */
push %ax                        /* Prepare stack for lret */
lret                            /* Reload CS with a real mode value */

xor %ax, %ax                    /* EAX = 0 */
mov %ax, %ds                    /* Put back real-mode segment registers */
addw $0x3000, %ax               /* Assume segment base 0x3000 */
mov %ax, %es
mov %ax, %ss
mov %ax, %fs
mov %dx, %sp
xor %ax, %ax
add $0xf000, %ax                /* Only gs needs to point on a BIOS segment */
mov %ax, %gs
sti                             /* Enable interrupts */
```

Table 7: Switch to real-mode with code segment base 0x3000

ENV_UNDI_INITIALIZE, *PXENV_UNDI_OPEN* and optionally *PXENV_UDP_OPEN* if what you desire to do is to send and receive UDP packets.

2.3 Calling PXE

After restoring the complexe enabled PXE stack at its original place, we just have to call the PXE services to perform the desired actions. Most PXE network services take in parameter a special structure describing the packet to be sent to the network. The structure looks as follow:

```
/* Packet data format defined in the PXE standard */
.code16
pxe udp packet:
pxe udp packet.status:      .word 0          /* Status : will be filled by PXE on return */
pxe udp packet.sip:         .long 0          /* Server IP */
pxe udp packet.gip:         .long 0          /* Gateway IP */
pxe udp packet.lport:       .word 7777       /* Local port in NBO */
pxe udp packet.rport:       .word 7777       /* Remote port in NBO */
pxe udp packet.buffersize:  .word 20         /* Size of packet */
pxe udp packet.buffer:      .word 0,0        /* seg:off for packet data */
pxe udp force:              .word 0          /* Useful for PXE FORCE INT */
```

Table 8: Format of a PXE packet

The address of pxe udp packet should be put into a register, for example **EDI**. Sending an UDP packet now looks amazingly simple: Note that the call to **PXENV_UNDI_FORCE_INT** is not mandatory to send a packet, but it forces the frame to be sent right away without waiting for the next PXE interrupt to happen. This avoids lags due to interrupt latency and so on. Additionally, we have isolated the BIOS call into a small stub pointed by symbol **pxenv**.

Such stub looks like this:

```
/* Now sending UDP packet using the BIOS PXE stack API */
.byte 0x66
movl pxe udp packet, %edi
xorl %ebx, %ebx
movw $PXENV_UDP_WRITE, %bx
call pxenv
.byte 0x66
xorl %ebx, %ebx
movw $PXENV_UNDI_FORCE_INT, %bx
add $20, %di
call pxenv
sub $20, %di

/* Here we call the PXE BIOS routine in charge of sending packets */
pxenv:
.code16
push %es                /* Push segment base (in our case: 0x3000) for return address */
push %di                /* Push address of parameter (generally packet descriptor) */
push %bx                /* Push PXE service code */
call after2             /* Get our address */

after2:
popw %ax                /* Put it in EAX */
add $0x13, %ax          /* We will come back on the add $6, sp */
xorl %ecx, %ecx         /* ECX = 0 */
movw $0x3000, %cx       /* Again, assume that our code is mapped at 0x3000 */
pushw %cx
pushw %ax

.off: pushw $0xEEEE     /* Will be relocated */
.seg: pushw $0xEEEE     /* Will be relocated */

lret                    /* Call BIOS PXE handler ! */
add $6, %sp             /* Remove parameters from stack */
ret                     /* Back to the real-mode caller */

.set PXESEG, .seg+1     /* We set PXESEG to point one byte after label seg */
.set PXEOFF, .off+1     /* We set PXEOFF to point one byte after label off */
```

Table 9: Sending a packet using PXE

3 Related Work

There is only few work dealing with the PXE stack for other purpose than booting an operating system image. Derek Soeder and Ryan Permeh created BootRoot [6], a boot-level rootkit downloaded by PXE and used to subvert the Windows kernel. On the same line, NGS presented on firmware rootkits [3], which included a quick briefing on using and modifying etherboot [5] in order to download an alternative OS image throught PXE. Such technique involves hooking interrupt 19h to get control at boot (after the PXE boot has been performed). However no implementation was provided with their article. It is worth mentioning that, unlike us, both of those works assume that PXE is enabled on the machine, that no operating system has been started yet, and that the OS image is downloaded using PXE. Our code assumes none of those preconditions.

Another related work is about high-performance network tools. Those are gener-ally loaded at the operating system level, for example using a loadable kernel module. PortBunny [7] was developed on this model to bring a time-sensitive network scanner capable of answering in a determined (real) time. This port scanner is Linux-specific and it is non-obvious how such tool can be maintained over the range of operating systemand kernel versions. For this reason, we believe that a BIOS-level port-scanner would make a more portable and maintainable tool.

4 Conclusion

We have opened the door to using the PXE stack for other purpose than booting an operating system image. The PXE API is very complete and allows almost any kind of packets to be sent or received from the network directly using the hardware without having to rely on any OS-level driver or userland OS API. Moreover, this is done independently of the network card, as the PXE UNDI driver abstracts such specificities to us. We believe such BIOS-level features will receive more attention in the future for developing light-weight cost-effective network applications independently of any operating system.

5 Acknowledgments

The author is grateful to Diogenes Pereira for his technical support during the development of this project and thankful to Paulo Urbano and Fabio Maia from the CESAR embedded team for making this publication possible.

References

[1] Preboot Execution Environment (PXE) specifications V2.1 - Intel Corp.

[2] Intel Pentium 4 reference manual, volume 3 - Intel Corp.

[3] Firmware rootkits (presentation) - Blackhat DC conference 2007 - John Heasman- NGS

[4] Implementing and Detecting a PCI Rootkit - Blackhat DC conference 2007 - JohnHeasman - NGS

[5] Etherboot - An opensource PXE stack - The Etherboot team

[6] Bootroot - Blackhat USA conference 2005 - Derek Soeder and Ryan Permeh - EEye

[7] PortBunny - a kernel level time-sensitive port scanner - Recurity Labs - http://www.recurity.de/portbunny/