Breaking Through the Memory Barrier in Malicious PCI Expansion ROM

Malicious code in x86/x64 firmware could reside in many potential places. One of them is in the PCI expansion ROM. One of the downside of this method in the past is the very limited amount of memory during PCI expansion ROM execution. Therefore, limiting the possible task carried out by such malicious code due to the limited space for code and data. This article explains how a malicious PCI expansion ROM might exploit the not widely known BIOS memory management interface to break through the memory "barrier", thus creating a potentially more complex threat. The discussion in this article is limited to PCI expansion ROM conforming to PCI firmware revision 3.1 specification. This article would not delve into EFI expansion ROM despite the subject is very closely related. EFI expansion ROM will be presented in different article.

What is PCI Expansion ROM Anyway?

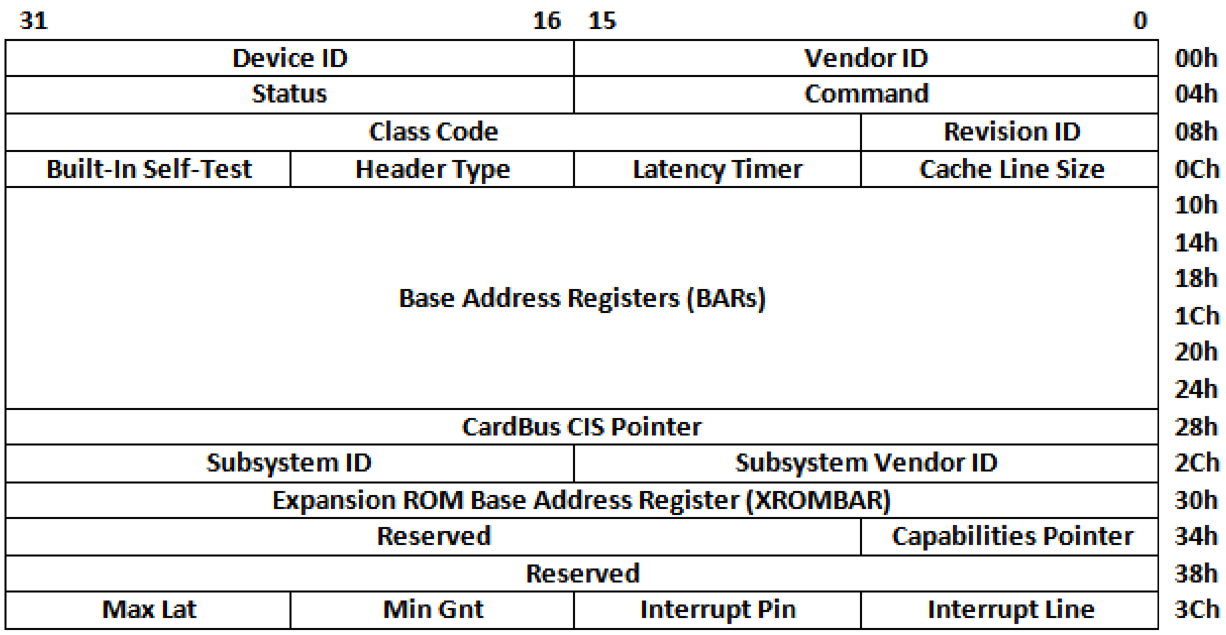
It is important for you to know the basics of PCI expansion ROM before I delve into how malicious PCI expansion ROM code might use the memory management routine in the BIOS. This part is dedicated to that end.

PCI Device Initialization

PCI Expansion ROM is a code, executed during BIOS initialization to initialize certain types of PCI/PCI Express devices. *From this point on PCI Express will be abbreviated as PCIe*. Device in this context refers to logical—as opposed to physical—PCI/PCIe device. PCI specification explains about this logical device detailed definition. PCIe devices are backward compatible to PCI in the logical level. Therefore, from programmer's point of view, PCIe devices are seen just like PCI devices. Of course there are several fundamental differences, but they are "extensions" to the basic PCI specification. *For the purpose of this article, you can regard PCI and PCIe as the same thing except when I specifically stated the matter to be PCI-specific or PCIe-specific.*

A single physical PCI chip; let's say an Ethernet controller chip could contain more than one logical PCI devices. Each logical PCI device has a corresponding set of PCI configuration registers in the *PCI configuration space*. In the x86/x64 architecture, the PCI configuration space is defined as 256 (100h) bytes of PCI configuration registers (per-logical PCI device) accessed through two 32-bit ports at I/O port CF8h-CFBh and CFCh-CFFh respectively. Port CF8h-CFBh is the "index" port, used to point to specific 32-bit PCI configuration register within the configuration space. Port CFCh-CFFh is the "data" port, used to write/read to/from PCI configuration register within the PCI configuration space.

Figure 1 shows, the general layout of the *header portion* of the PCI configuration registers in a PCI device. This particular configuration is called PCI configuration Type 0. It is used for PCI device which *is not* a PCI bridge. A PCI bridge connects two different PCI bus. PCI devices, such as Ethernet card or video card are not PCI Bridge. Therefore, I'm only focusing on PCI configuration Type 0.



31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
Built-In Self-Test	Header Type	Latency Timer	Cache Line Size	0Ch
Base Address Registers (BARs)				10h
				14h
				18h
				1Ch
				20h
				24h
CardBus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address Register (XROMBAR)				30h
Reserved			Capabilities Pointer	34h
Reserved				38h
Max Lat	Min Gnt	Interrupt Pin	Interrupt Line	3Ch

Figure 1 Base Address Registers (BARs) in the PCI Configuration Space

PCI devices' registers (outside of the PCI configuration register) and memory (RAM and/or ROM) could be located anywhere within the CPU memory address space or CPU I/O address space because PCI devices contain programmable registers, called Base Address Register (BAR). Figure 1 shows the location of the BARs in the header portion of the PCI configuration registers. BAR controls where the PCI devices' registers (outside of the PCI configuration register) and memory are mapped in the CPU memory or I/O address space. The reason for this programmability is to eliminate address conflicts between devices which plague the old ISA and EISA devices.

Figure 1 also shows a very specific form of BAR at offset 30h, the *Expansion ROM Base Address Register (XROMBAR). This particular register maps the PCI expansion ROM in a PCI device into the CPU memory or I/O address space.* It's important that you take a note of this particular register because it's related directly to PCI expansion ROM execution. I'll talk more about this register in later sections of this article.

Some PCI devices also map their registers—the non-PCI configuration registers—into the I/O address-space but that approach is *not* the preferred approach as of today except for the crucial *PCI configuration registers*. One of the reasons the PCI configuration registers are mapped to I/O space is to maintain backward compatibility, the backward compatibility is especially important for legacy operating system such as Windows XP and other older proprietary operating systems.

Figure 2 shows a simplified mapping of PCI/PCIe device registers and memory into the CPU I/O address space and memory address space respectively. Remember, Figure 2 is a simplified version of the real world device registers and memory mapping. In practice, you might find the mapping to be more complicated than that. Read the PCI/PCIe device datasheet along with other related datasheet, such as the CPU and chipset(s) to find out the exact mapping.

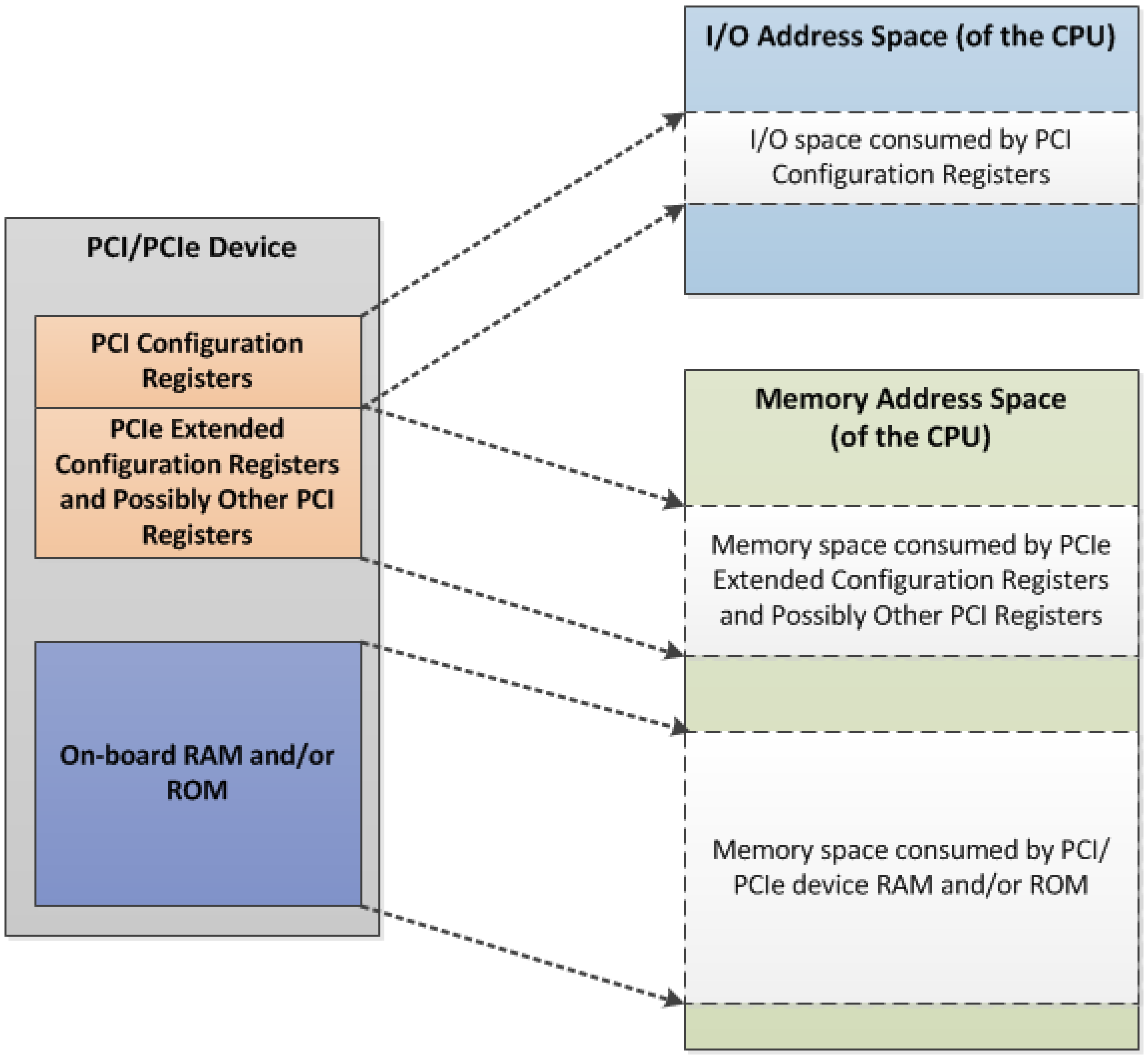

I/O Address Space (of the CPU)
I/O space consumed by PCI Configuration Registers
PCI/PCIe Device
PCI Configuration Registers
PCIe Extended Configuration Registers and Possibly Other PCI Registers
On-board RAM and/or ROM
Memory Address Space (of the CPU)
Memory space consumed by PCIe Extended Configuration Registers and Possibly Other PCI Registers
Memory space consumed by PCI/ PCIe device RAM and/or ROM

Figure 2 PCI/PCIe device registers and memory mapping to CPU/system address spaces

PCI device registers—non PCI configuration registers—and memory must occupy certain memory or I/O address space before they could be used. Therefore the device must be *initialized* to known—exact—addresses within the CPU I/O or memory address space. *The motherboard BIOS carries out this initialization task*. Figure 2 shows a sample of the PCI/PCIe device register and memory mapping into the system I/O and memory address space after the initialization is completed.

Certain types of PCI devices require much more elaborate, *device-specific initialization*. In these devices, mapping their devices' registers and memory to the CPU address space is *only one of* the initialization tasks which need to be done. *Motherboard BIOS cannot carry out these device-specific initialization tasks because it doesn't have prior knowledge of all possible devices and their respective initialization procedures*. Besides, doing the initialization in the motherboard BIOS is impossible due to motherboard BIOS size constraint. You would need the amount of space as much as an OS needs to do that, which is impractical. Therefore, the PCI expansion ROM emerges to fill the gap.

There are at least two types of PCI devices that require the additional device-specific initialization—hence needs PCI expansion ROM—i.e., devices used to boot an operating system and display devices. The first category includes storage controller chips, such as SCSI or RAID controller and network controller such as Ethernet controller (for boot from LAN), while the second category includes video card chips.

PCI Expansion ROM Location

Now, let me turn to explain where you can find the PCI expansion ROM. PCI expansion ROM is placed as binary file in a flash ROM chip. There are two types of PCI expansion ROM in terms of physical appearance, depending on the location of the PCI device:

1. The first one is PCI expansion ROM located in a PCI expansion card, in which case the binary file is placed in a flash ROM chip soldered or socketed into the PCI expansion card.
2. The second one is PCI expansion ROM integrated—embedded—into the mainboard BIOS.

In the first case, the *physical* PCI chip to be initialized is soldered—could be socketed as well –into the PCI expansion card, while in the second case the PCI chip to be initialized is soldered into the motherboard. Figure 3 shows both of these arrangements.

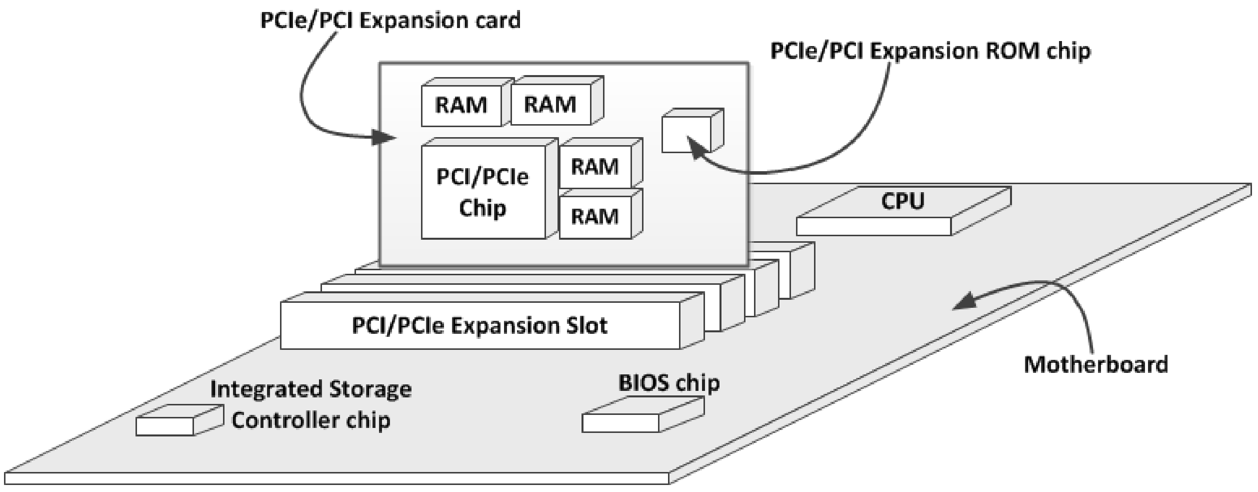


Figure 3 PCI expansion ROM location illustration

Figure 3 shows an *incomplete* system with missing motherboard RAM and many other motherboard components. That's because the emphasis is in showing you the possible locations of the PCI expansion ROM chip and the BIOS chip. The exact location of these chips depends on the particular motherboard and PCI/PCIe expansion card. Therefore, you should check the motherboard manual and the PCI/PCIe expansion card manual, or carry out physical inspection to make sure about the location of these chips. If you're still unsure, you can look for flash ROM chip on the respective boards. Look for memory chip made by Winbond, SST, Atmel or other flash ROM manufacturer, then cross check the chip number on the web to make sure. BIOS chip usually has clear marking. However, PCI expansion ROM chip usually doesn't have marking and in the case of PCIe expansion card, sometimes soldered into the back of the card. Therefore, the latter case is a bit of a guessing game.

PCI Expansion ROM embedded in the motherboard BIOS vs. PCI Expansion ROM in PCI Expansion Card

PCI expansion ROM embedded in the motherboard BIOS and PCI expansion ROM residing in PCI expansion card differ logically from programmer perspective. The logical difference lies in PCI expansion ROM execution steps and the role of the XROMBAR in the corresponding PCI device. The steps to execute PCI expansion ROM are:

1. Motherboard BIOS copies the PCI Expansion ROM—from where the PCI expansion ROM is stored—to RAM in the expansion ROM area (within C_0000h – DFFFFh range).
2. Motherboard BIOS jumps into offset 03h in the copied PCI expansion ROM to execute it.
3. PCI expansion ROM "returns" the execution back to the motherboard BIOS.

The first step above is *different* for PCI expansion ROM stored in flash ROM on the PCI expansion card and PCI expansion ROM embedded in the motherboard BIOS. In the first case, the motherboard BIOS copies the PCI expansion ROM from the flash ROM in the PCI expansion card to RAM by using the address stored in the XROMBAR of the PCI chip in the corresponding PCI device as the PCI expansion ROM address.

The motherboard BIOS initializes all of the BARs and XROMBARs in all PCI devices prior to starting to execute **any** PCI expansion ROM(s). The initialization effectively sets the location of the PCI Expansion ROM in the CPU I/O or memory address space. Therefore, the motherboard BIOS knows the address of the PCI expansion ROM in the CPU I/O or memory address space before copying it to the RAM. The mechanism to copy the PCI expansion ROM in this particular hardware configuration is the same across all BIOS, regardless of the BIOS code base, whether it's AMI, Award-Phoenix, Insyde, or others.

PCI expansion ROM embedded in the motherboard BIOS is treated differently. Each BIOS code base, either from AMI, Award-Phoenix, Insyde or others has its own vendor-specific module that locates the PCI expansion ROM in the motherboard BIOS and then copy the PCI expansion ROM to RAM before the PCI expansion ROM executes. In this case, the format of the PCI expansion ROM in the motherboard BIOS also vendor-dependent, each BIOS vendor have their own specific approach. However, most vendors store the PCI expansion ROM in compressed format inside the motherboard BIOS to save space.

PCI and PCIe Expansion ROM Structure

Various specifications, including the PCI specification dictates the basic structure of both PCI and PCIe expansion ROM until circa 2005. In 2005, the disparate specifications were consolidated into the PCI Firmware Specification Revision 3.0. This consolidated specification of course is backward compatible to the previous PCI specification revisions. The most recent version of the PCI Firmware Specification is revision 3.1, published in December 2010.

Present day PCIe devices mostly still use PCI expansion ROM instead of EFI expansion ROM. However, some PCIe devices use a mixed PCI expansion ROM and EFI expansion ROM and a very small fraction of PCIe devices use EFI expansion ROM only. You will see very shortly how this is accomplished.

Let me start with the very basic requirement for a PCI expansion ROM. PCI expansion ROM could contain one or more “image(s”). Each “image” is a self-contained code and data that contains all of the necessary routines and data required to initialize the corresponding PCI/PCI chip. The purpose of having multiple “images” in one PCI expansion ROM is to support several CPU architectures, which have different instruction encoding. Therefore, despite the PCI expansion ROM contains several images, *only one of them will be executed* at runtime because only the *first* image that match the CPU architecture at that point would be executed by the platform firmware—the BIOS in the case of x86/x64.

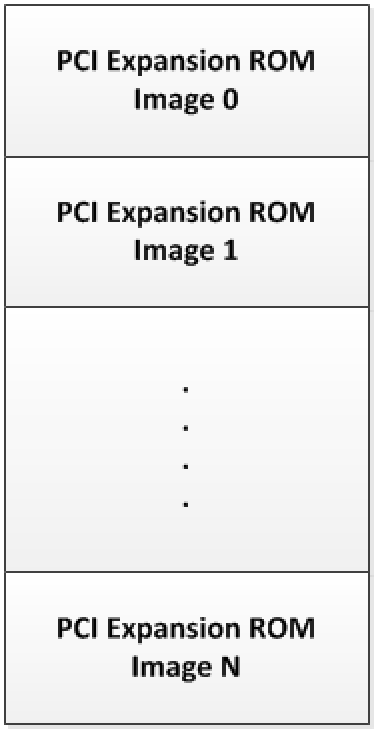

PCI Expansion ROM Image 0
PCI Expansion ROM Image 1
PCI Expansion ROM Image N

Figure 4 Structure of PCI expansion ROM with Multiple Images

Figure 4 shows the structure of PCI expansion ROM with multiple images. Each image in the PCI expansion ROM must start in 512-bytes boundary and must contain PCI expansion ROM header. As you will see later, a special field named *Last Image Indicator* (offset 15h) in the *PCI data structure* of the PCI expansion ROM indicates whether the image is the last image in the image list or not.

Each image in the PCI expansion ROM starts with the *PCI expansion ROM header*. Table 1 shows the format of the PCI expansion ROM header. The *offset* in Table 1 is calculates from the *start of the image* in the PCI expansion ROM.

Table 1 Generic PCI Expansion ROM Header Format

Offset	Length	Value	PCI Expansion ROM Header Field
00h	1	55h	ROM signature, byte 1
01h	1	AAh	ROM signature, byte 2
02h-17h	16h	xx	Reserved (processor architecture unique data)
18h-19h	2	Xx	Pointer to PCI Data Structure

PCI Firmware Specification has a specific rule for the format of the processor architecture unique data for x86/x64 architecture. Table 2 shows the PCI expansion ROM header format.

Table 2 x86/x64-specific PCI Expansion ROM Header Format

Offset	Length	Value	X86/x64 PCI Expansion ROM Header Field
00h	1	55h	ROM signature byte 1
01h	1	AAh	ROM signature byte 2
02h	1	Xx	Current image size in unit of 512 bytes
03h	3	Xx	Entry point for INIT function. Power-On Self-Test (POST)does a FAR CALL to this location
06h-17h	12h	Xx	Reserved (application unique data)
18h-19h	2	Xx	Pointer to CPI Data Structure.

The offset in Table 2 is calculated from the start of the PCI expansion ROM image. Previously, I explained that multi-image PCI expansion ROM contains images for different CPU architectures. A specific field, named *Code Type* (offset 14h) in the PCI Data Structure indicates the type of the CPU architecture supported by the specific PCI expansion ROM image.

The PCI specification dictates that the *PCI data structure must reside within the first 64KB of the PCI expansion ROM image*. Now, let me proceed to show you the format of the PCI Data Structure that I have referenced several times above. Table 3 shows the format of the PCI Data Structure.

Table 3 PCI Data Structure

Offset	Length	PCI Data Structure Field
0	4	Signature, the “PCIR” string
4	2	Vendor Identification
6	2	Device Identification
8	2	Device List Pointer
A	2	PCI Data Structure Length
C	1	PCI Data Structure Revision
D	3	Class Code
10	2	Image Length
12	2	Revision Level of the Vendor’s ROM
14	1	Code Type
15	1	Last Image Indicator
16	2	Maximum Run-time Image Length
18	2	Pointer to Configuration Utility Code Header
1A	2	Pointer to DMTF CLP Entry Point

Table 3 shows the *code type* field at offset 14h and *last image indicator* at offset 15h. Table 4 shows the meaning of each of this entry.

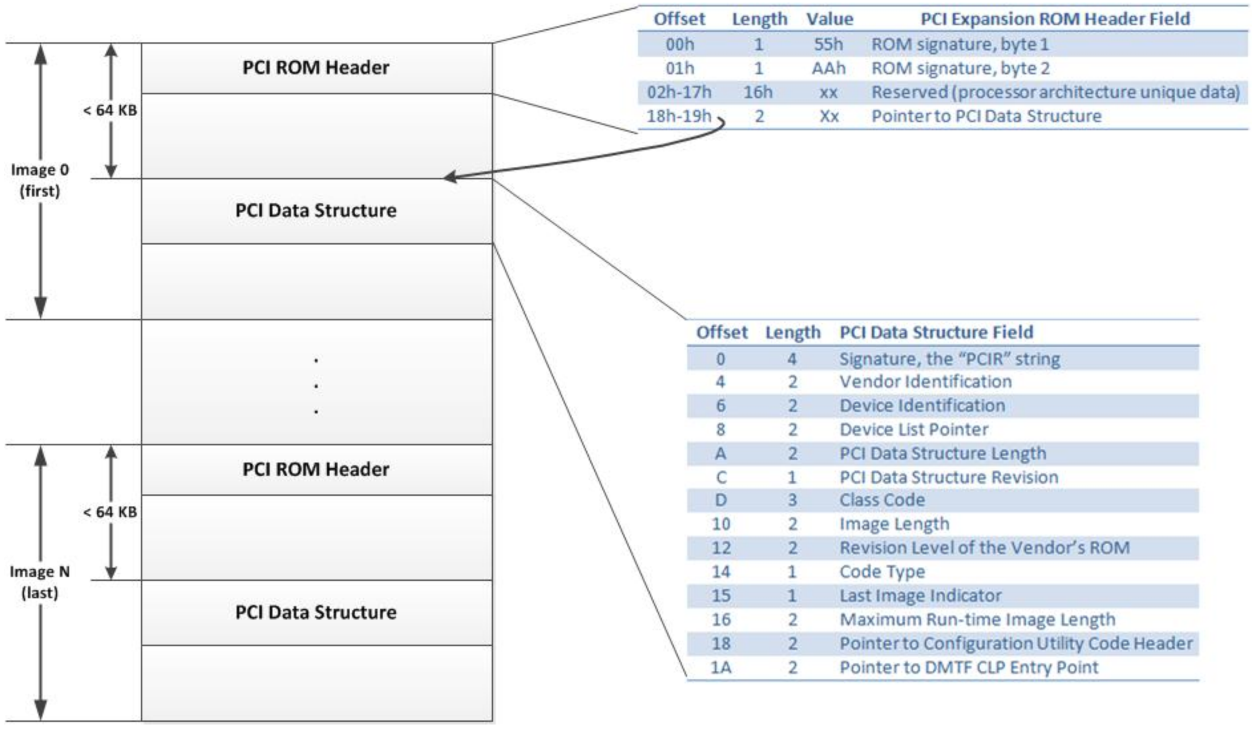
Table 4 PCI Data Structure Explanation

PCI Data Structure Field	Explanation
Signature	These four bytes provide a unique signature for the PCI Data Structure.

	The string “PCIR” is the signature with “P” being at offset 0, “C” at offset 1, etc.
Vendor Identification	The Vendor Identification field is a 16-bit field with the same definition as the Vendor Identification field in the PCI Configuration Space for this device.
Device Identification	The Device Identification field is a 16-bit field with the same definition as the Device Identification field in the PCI Configuration Space for this device.
Device List Pointer	The Device List Pointer is a two-byte pointer in little-endian format that points to the list of Device IDs supported by this ROM. The beginning reference point (“offset zero”) for this pointer is the beginning of the PCI Data structure (the first byte of the Signature field). *This field is only present in Revision 3.0 (and greater) PCI Data structures*.
PCI Data Structure Length	The PCI Data Structure Length is a 16-bit field that defines the length of the data structure from the start of the data structure (the first byte of the Signature field). This field is in little-endian format and is in units of bytes.
PCI Data Structure Revision	The PCI Data Structure Revision field is an eight-bit field that identifies the data structure revision level. The revision level is 3 for the current specification (revision 3, 3.1 and above).
Class Code	The Class Code field is a 24-bit field with the same fields and definition as the class code field in the PCI Configuration Space for this device.
Image Length	The Image Length field is a two-byte field that represents the length of the image. This field is in little-endian format, and the value is in units of 512 bytes.
Revision Level	The Revision Level field is a two-byte field that contains the revision level of the Vendor’s code in the ROM image
Code Type	The Code Type field is a one-byte field that identifies the type of code contained in this section of the ROM. The code may be executable binary for a specific processor and system architecture or interpretive code. The following code types are assigned: **Type** **Description** 0 Intel x86, PC-AT compatible 1 Open Firmware standard for PCI 2 Hewlett-Packard PA RISC 3 Extensible Firmware Interface (EFI) 4-FF Reserved
Last Image Indicator	Bit 7 in this field tells whether or not this is the last image in the ROM. A value of 1 indicates “last image;” a value of 0 indicates that another image follows. Bits 0-6 are reserved.
Maximum Run-Time Image Length	The Image Length field is a two-byte field that represents the maximum length of the image after the initialization code has been executed. This field is in little-endian format, and the value is in units of 512 bytes. This field will be used to determine if the run-time image size is small enough to fit in the memory remaining in the system. *This field is only present in Revision 3.0 and later of the PCI Data structure.*
Pointer to Configuration Utility Code Header	This pointer is a two-byte pointer in little-endian format that points to the Expansion ROM’s Configuration Utility Code Header table at the beginning

	of the configuration code block. The beginning reference point ("offset zero") for this pointer is the beginning of the Expansion ROM image. This field is only present in Revision 3.0 (and greater) PCI Data structures. A value of 0000 will be present in this field if the Expansion ROM does not support a Configuration Utility Code Header.
Pointer to DMTF CLP Entry Point	This pointer is a two-byte pointer in little-endian format that points to the execution entry point for the DMTP CLP code supported by this ROM. The beginning reference point ("offset zero") for this pointer is the beginning of the Expansion ROM image. This field is only present in Revision 3.0 (and greater) PCI Data structures. A value of 0000 will be present in this field if the Expansion ROM does not support a DMTF CLP code entry point.

Now, you have gone through three quite complex data structure and you're thinking how all of them interrelated. Let me put together what we have learned so far. Figure 5 shows the relationship between the data structures in a PCI expansion ROM explained above.



Offset	Length	Value	PCI Expansion ROM Header Field
00h	1	55h	ROM signature, byte 1
01h	1	AAh	ROM signature, byte 2
02h-17h	16h	xx	Reserved (processor architecture unique data)
18h-19h	2	Xx	Pointer to PCI Data Structure

Offset	Length	PCI Data Structure Field
0	4	Signature, the "PCIR" string
4	2	Vendor Identification
6	2	Device Identification
8	2	Device List Pointer
A	2	PCI Data Structure Length
C	1	PCI Data Structure Revision
D	3	Class Code
10	2	Image Length
12	2	Revision Level of the Vendor's ROM
14	1	Code Type
15	1	Last Image Indicator
16	2	Maximum Run-time Image Length
18	2	Pointer to Configuration Utility Code Header
1A	2	Pointer to DMTF CLP Entry Point

Figure 5 Relationships between the various PCI expansion ROM structures

Figure 5 shows the components of a multi-image PCI expansion ROM in detail. As you can see, the pointer to PCI data structure field—in the PCI expansion ROM header—points to the PCI data structure within the first 64KB of the PCI expansion ROM image.

Support for multi-image PCI expansion ROM—which exists since long ago—opens a possibility to support legacy BIOS system and the newer UEFI-based system in one multi-image PCI expansion ROM. As you can see in Table 4, *code type* for EFI and "ordinary" x86/x64 system is different. Therefore, creating multi-image PCI expansion ROM with one of the image consisting of “ordinary” x86 PCI expansion ROM and another consisting of EFI expansion ROM is possible. Setting the correct *code type* field in the respective PCI expansion ROM image is enough to “signal” the platform firmware (BIOS or UEFI) to execute the right image. Refer to Table 4 for details of the *code type* field.

Memory Limitation in PCI Expansion ROM Execution during Power-On Self-Test (POST)

I have explained in the previous sections that the PCI expansion ROM executes after all of the PCI devices’ registers are initialized. This initialization is part of what is known as the Power-On Self-Test (POST) in the x86/x64 architecture.

X86/x64 boots-up in 16-bit real mode. The machine briefly switched to big-real mode (16-bit with access to 4GB memory address space) several times, but most of the time the machine stays in 16-bit real mode. PCI expansion ROMs are not executed in-place, i.e. not executed directly in ROM, but have to be copied to RAM and execute there. In the old days, this probably didn’t create complication because the size of the PCI expansion ROMs are quite small. The space allocated to execute *all* of the PCI expansion ROM is *only* 128KB, from C0000h to DFFFFh. This 128KB space becomes limitation as the PCI devices becomes more complex and requires much more elaborate initialization.

Back in 1997, Intel Corp. and Phoenix Technologies already foreseen this size limitation as a very potential future problem, therefore both of them created a specification to alleviate the problem. It was named *POST Memory Manager (PMM)* Specification. This specification—in its version 1.01 incarnation—is still in use today. This specification allows PCI expansion ROMs to overcome the 128KB memory limitation and reserve more memory for their use. You have to be aware that even the 128KB space had to be shared between all of the PCI expansion ROMs in the system. It’s virtually impossible for single PCI expansion ROM to occupy the entire 128 KB space. This is the root of the problem.

Now, let’s see how PMM might be used by the bad guys to give their malicious PCI expansion ROM more room to breathe. By using PMM services, placing a full-fledged complex routines in the PCI expansion ROM is no more a dream, they could, for example place a rather complete file system infectors (driver if you wish) in the PCI expansion ROM or perhaps a cryptic System Management Mode code. The possibility is quite large. I’ll leave more specific discussion on this matter in later sections of this article. The next section explains details of the PMM services.

POST Memory Manager (PMM) Comes to the Rescue

The code in the PCI expansion ROM must check for the presence of the PMM services before it can use it. A structure, named PMM structure signals the presence of the PMM services. PMM structure is located in the system BIOS address space on a paragraph (16-byte) boundary between addresses E0000h and FFFF0h—or equivalently, between segment E000h and FFFFh. The presence of the PMM structure within that address range indicates PMM services are present and could be called.Table 5 shows contents of the PMM structure.

Table 5 PMM Structure

Offset	Name	Size (in Bytes)	Value	Description
00h	SignatureByteOne	1	‘$’	Signature byte 1
01h	SignatureByteTwo	1	‘P’	Signature byte 2
02h	SignatureByteThree	1	‘M’	Signature byte 3
03h	SignatureByteFour	1	‘M’	Signature byte 4
04h	StructureRevision	1	01h	Structure Revision
05h	StructureLength	1	Varies	Length of this structure in bytes
06h	StructureChecksum	1	Varies	Checksum update field
07h	EntryPoint	4	Varies	Segment:offset of PMM Services entry point.
0Bh	Reserved	5	0	Reserved

A PCI expansion ROM follows this procedure to locate and access PMM Services (functions):

1. Search for the four-byte “$PMM” string on paragraph (16-byte) boundaries starting at E0000h, and ending, if not found, at FFFF0h.
2. Verify that the PMM Structure data is valid by performing a checksum. The checksum is calculated by doing a byte-wise sum of the entire PMM Structure and comparing this sum with zero. If the checksum is **not** zero, then the PMM Structure data is not valid and the *EntryPoint* field of the PMM structure should not be called.
3. *Optionally* inspect the *StructureRevision* field to determine the appropriate structure map. The *StructureRevision* field changes if previously reserved fields in the PMM Structure are redefined to be valid fields.
4. Make calls (far calls) to the *EntryPoint* field in the PMM Structure to allocate and free memory as desired.

Calls to PMM services (functions) are defined as if called from the C programming language. The PMM interface uses the standard C (16-bit) large model calling convention. In particular, return values of type unsigned long are returned in the DX:AX register pair. PCI expansion ROMs call the PMM Services by executing a far call to the address specified in the *EntryPoint* field in the PMM structure. PMM services return to the PCI expansion ROM routine via a far return. Values returned to the PCI expansion ROM routine are placed in the DX:AX register pair.

BIOS that provide support for PMM follows this rule: When control is passed to an option ROM—including PCI expansion ROM—from a BIOS that supports PMM, the processor will be in *big real mode*, and Gate A20 will be disabled (segment wrap turned off). This allows access to extended memory blocks (memory above the 1MB limit) using real mode addressing. In big real mode, access to memory above 1MB can be accomplished by using a 32-bit extended index register (EDI, ESI, etc.) and setting the segment register to 0000h. The code snippet in Listing 1 below shows an example of reading data from the extended memory block to AX register.

Listing 1 Accessing extended memory block sample code

```
; Assume here that dx:ax contains the 32-bit address of an already allocated buffer.
; Clear the ES segment register.
push 0000h
pop es

; Put the DX:AX 32-bit buffer address into EBX.
mov bx, dx      ; Get the upper word.
shl ebx, 16     ; Shift it to upper EBX.
mov bx, ax      ; Get the lower word.

; read the first two bytes of the extended memory buffer to ax.
mov ax, es:[ebx]; ES:EBX is used as the memory pointer.
```

Because the control is passed from the BIOS to the PCI expansion ROM in big real mode, code in the PCI expansion ROM should not mess with the state of the CPU. Code in the PCI expansion ROM must not modify the state of Gate A20, put the processor in protected mode, or call BIOS Interrupt 15h, functions 87h or 89h. Any of these actions would alter the big real mode of the processor and could lead to a catastrophic system failure. This requirement put a constraint on the type of code that you can place in the PCI expansion ROM. If you wish to use a C compiler to make PCI expansion ROM code, you have to make sure the compiler supports emitting code in big-real mode form—remember the old DOS32 days?—or just plain real mode code. This is probably amusing, but compilers used during the days of the first Doom game come to the rescue in this situation.

Now that you have known the entry point to use PMM functions, the time has come to find out what functions offered by the PMM entry point. But before that I will show you the generic form of the function. Listing 2 shows the function prototype. The *entryPoint* in Listing 2 is the entry point defined in the *EntryPoint* field in the PMM structure shown in Table 5.

Listing 2 PMM functions entry point

```
unsigned long (*entryPoint)(
unsigned int functionNumber, // this is where you specify which function to call
        … // other function parameters, depending on the function being called
);
```

You invoke the functions provided by PMM by using their function number, i.e. function number is set to 0 for function 0, etc. There are three functions offered by the PMM entry point:

1. Function 0: `pmmAllocate()`. This function allocates memory either in the conventional memory (within the first 1MB) or in extended memory (above the first 1MB). There is an optional "handle" in the parameter to call this function. The "handle" is used to identify the memory block that's allocated. It's important only if you want to access that same memory block in your PCI expansion ROM long after the memory block has been allocated.

2. Function 1: `pmmFind()`. This function search for the memory block by using the "handle" that's used in a `pmmAllocate()` call previously and returns a pointer to (the address of) the allocated memory block.
3. Function 2: `pmmDeallocate()`. This function frees the memory block previously allocated in the call to `pmmAllocate()` function.

Now that the PMM functions are clear, I'm going to move to the details of those interface functions.

Function 0: pmmAllocate()

The `pmmAllocate()` function attempts to allocate a memory block of the specified *type* and *size*, and returns the address of the memory block to the caller. The address returned by `pmmAllocate()` function is a "far" address, i.e. 32-bit physical address of the allocated memory block. The memory block is a contiguous array of paragraphs (16-bytes) whose size is specified by the *length* parameter. The contents of the allocated memory block are undefined and must be initialized by the PCI expansion ROM. Listing 3 shows `pmmAllocate()` function prototype.

Listing 3 pmmAllocate() function prototype

```
unsigned long (*entryPoint)(
        unsigned int function, // 0 for pmmAllocate
        unsigned long length, // in paragraphs (multiple of 16-bytes)
        unsigned long handle, // handle to assign to memory block
        unsigned int flags // bit flags specifying options
);
```

The *entryPoint* identifier in Listing 3 is of type function pointer, and is a "far" address. It must be assigned the value of the *EntryPoint* field in the PMM structure before it can be called. The meaning of the parameters in the `pmmAllocate()` function as follows:

- *function*, 0 for pmmAllocate. Invalid values for the function parameter (0x0003…0xFFFF) cause an error value of 0xFFFFFFFF to be returned, signaling that the function is not supported.
- *length*, The size of the requested memory block in paragraphs, or if length is 0x00000000, no memory is allocated and the value returned is the size of the largest memory block available for the memory type specified in the *flags* parameter (below). The alignment bit in the flags register is ignored when calculating the largest memory block available.
- *handle*, identifier to be associated with the allocated memory block, specified by the routine in the PCI expansion ROM. A handle of 0xFFFFFFFF indicates that no identifier should be associated with the block. Such a memory block is known as an "anonymous" memory block and cannot be found using the pmmFind function (see below). If a specified handle for a requested memory block is already used in a currently allocated memory block, the error value of 0x00000000 is returned.
- *flags*, a "bitmap" used by the PCI expansion ROM routine to designate options regarding memory allocation. Table 6 shows the possible values for the flag.

  Table 6 pmmAllocate() flag

Bit(s) Position	Field	Value	Description
0..1	MemoryType	1..3	0 = Invalid 1 = Requesting conventional memory block (0 to 1MB). 2 = Requesting extended memory block (1MB to 4GB). 3 = Requesting either a conventional or an extended memory block, whichever is available.
2	Alignment	0..1	0 = No alignment. 1 = Use alignment from the length parameter.
3	Attribute	0..1	0 = Temporary memory use during POST 1 = Permanent memory use
4..15	Reserved	0	Reserved for future expansion, must all be zero.

- *flags.MemoryType*, specifies whether the requested memory block should be allocated from conventional memory (within the first 1MB), extended memory (above 1MB), or from either conventional or extended memory. At least one of the bits in this field must be set. If bit 0 is set, the PMM will attempt to allocate only from conventional memory. If bit 1 is set, the PMM will attempt to allocate only from extended memory. If both bits 0 and 1 are set, the PMM will first try to allocate from conventional memory, and if that fails, the PMM will then attempt to allocate from extended memory.
- *flags.Alignment*, specifies whether the requested memory block should be aligned on a specific memory address boundary or not. The alignment is defined as the least significant set bit of the length parameter. For example, if the length parameter was 0x00000500, then the alignment would be on a 0x100-paragraph boundary. This is equivalent to a 20KB buffer that is aligned on a 4KB boundary. If the length parameter is 0x00000000, the alignment bit is ignored.
- *Flags.Attribute*, this is a new flag defined in the PCI Firmware Specification Revision 3.0. It determines whether the memory allocated during PCI expansion ROM initialization is permanent or temporary, i.e. whether the contents of the allocated memory should be preserved and not usable/accessible to the Operating System (OS), akin to SMM memory. The protection mechanism is through the BIOS interrupt 15, E820h call. The BIOS will signal the caller of that interrupt that the permanent memory reserved by the PMM is not available for use by the OS.
- *flags.Reserved*, must all be zero.

The most interesting flag in the `pmmAllocate()` function is the *MemoryType* flag and *Attribute*. If you request an extended memory block in your call to `pmmAllocate()` and set the attribute to *permanent*, the memory allocated would behave like SMM memory which is not accessible to the OS and reside above the 1MB limit. This is probably a boon for malware creators, but for the defender, they want to make sure they scan those areas for malicious code.

Function 1: pmmFind()

The `pmmFind()` function returns the address of the memory block associated with the specified *handle*. Recall that the *handle* is specified as handle parameter in the call to `pmmAllocate()` function previously. The `entryPoint` identifier is of type function pointer, and is a far address. It must be assigned the value of the `EntryPoint` field in the PMM structure before it can be called. The return value is of type unsigned long (32-bit), and represents the 32-bit physical memory address of the memory block that is associated with the handle. A return value of `0x00000000` indicates that the handle does not correspond to a currently allocated memory block.

Listing 4 pmmFind() function prototype

```
unsigned long (*entryPoint)(
        unsigned int function, // 1 for pmmFind
        unsigned long handle // handle assigned to memory block
);
```

The meaning of the parameters in the `pmmFind()` function as follows:

- *function*, 1 for `pmmFind()`. Invalid values for the function parameter (0x0003…0xFFFF) cause an error value of 0xFFFFFFFF to be returned, signaling that the function is not supported.
- *handle*, an identifier specified by the PCI expansion ROM that was assigned to a memory block when the `pmmAllocate()` function was called. If called with an anonymous handle (0xFFFFFFFF) or a currently undefined handle, a value of 0x00000000 will be returned indicating that no associated memory block was found.

This function is not very important with respect to creating malicious PCI expansion ROM. However, it could be beneficial if the need to "tag" certain allocated PMM memory arises.

Function 2: pmmDeallocate()

This function is the counterpart of the `pmmAllocate()` function. The `pmmDeallocate()` function frees the specified memory block that was previously allocated by `pmmAllocate()`. Memory blocks are not cleared to all zeros by the PMM before being deallocated. Memory below 1MB is cleared by the BIOS before OS boot, but deallocated extended memory blocks in particular will not explicitly be cleared. Any code that calls the PMM should not rely on the contents of deallocated memory blocks.

Listing 5 pmmDeallocate() function prototype

```
unsigned long (*entryPoint)(
        unsigned int function, // 2 for pmmDeallocate
        unsigned long buffer // value returned by pmmAllocate
);
```

The meaning of the parameters in the `pmmDeallocate()` function as follows:

- *function*, 2 for pmmDeallocate. Invalid values for the function parameter (0x0003…0xFFFF) cause an error value of 0xFFFFFFFF to be returned, signaling that the function is not supported.

- *buffer*, the 32-bit physical address of the memory block. This is the value that was returned by the call to `pmmAllocate()` function previously.

This function is important if you want to deallocate memory that's previously allocated with `pmmAllocate()` call. However, if you allocated a permanent memory in your call to pmmAllocate()—see the attribute flag in the `pmmAllocate()` explanation—you probably don't need this function.

At this point all of the functions needed to allocate and work with larger memory pool are clear. The 128KB limitation no longer exists. However, you might want to obfuscate the presence of your code in the PCI expansion ROM by going several more steps further. The next section explains what we could do for that.

The Role of Executable Compressors

Now that you have your arsenal to fight against the memory limitation in your PCI expansion ROM code, it's not enough though. Most PCI expansion card flash ROM is very limited in terms of capacity; probably 128KB is the largest so far. The same is true with BIOS flash ROM chip, the size of BIOS flash ROM chip maxed out at 1MB; probably there are some exotic boards with 2MB. The way to overcome this limitation is to compress your PCI expansion ROM code (and data), so that you can get the most out of those limited flash ROM chip.

A quick survey over available executable file compressor shows that UPX is probably the best for the PCI expansion ROM code compression, because it supports compressing 16-bit x86 code. Entry point for the decompressor routine in a UPX-compressed file resides in the beginning of the compressed binary. Therefore, it's easy for you to integrate UPX-compressed binary into your code. The compression ratio is also quite good. However, because UPX focuses on executable code compression, its data compression capability is probably not as good as data-focused compressor out there. Nonetheless, 16-bit decompressor is quite hard to come by these days.

Now, let me show you the possible layout of a UPX-compressed PCI expansion ROM. Figure 6 shows the layout of PCI expansion ROM containing a UPX-compressed module.

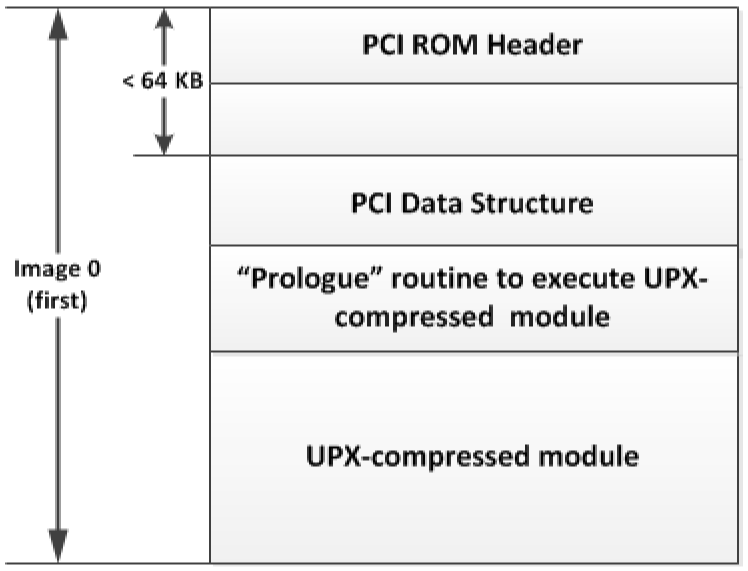


Figure 6 PCI Expansion ROM containing UPX-compressed module

Figure 6 doesn't show that you should minimize the distance between the PCI ROM header and PCI data structure. However, you should do that, because you want to maximize the space for the UPX-compressed module. The "prologue" section contains the routine to pass execution into the UPX-compressed module. Well, you could innovate here and do whatever you like, just make sure you don't make a code that overwrite important part of the PCI Expansion ROM and make sure the final execution of the PCI expansion ROM returns to the motherboard BIOS with a "far" return.

C Code in PCI Expansion ROMs

It is now clear how to make a large PCI expansion ROM for your needs. However, you're not really comfortable to use assembly language and you want to use C as your language to develop PCI expansion ROM. What would you do? Unfortunately, it's (almost) impossible to use only C language, you still need some assembly language here and there. However, it's possible to use only C language for the UPX-compressed module that you see in Figure 6.

Now, let's explore what options that you have if you want to use a C compiler. I've been toying with using GCC to develop an incomplete "bootloader" in a PCI expansion ROM long ago. You can see the complete explanation at: https://sites.google.com/site/pinczakko/low-cost-embedded-x86-teaching-tool-2. However, the code in that link switches the machine to 32-bit protected mode, because it was designed as a sort of "precursor" to a small operating system. Moreover, it resides in an Initial Program Loader (IPL) device (a storage controller card), i.e. a device that the BIOS regards as having the capability to boot into an OS, akin to a hard drive. Therefore, it could hook into interrupt 19h to boot the system and switching to 32-bit protected mode is just fine in that scenario. If all you want is to use the "large" memory space available from the PMM functions, you should use another approach.

Using the mainline GCC that runs in Linux or Unix to develop your C compiled PCI expansion ROM module is close to impossible. There are other better routes though. In this case, you could opt to use

OpenWatcom (www.openwatcom.org) or DJGPP (http://www.delorie.com/djgpp/). I will leave the rest of the task to another article, but you could also try it out yourself. There are probably other compilers out there that can emit 16-bit real mode or 16-bit big real mode code. It's wise to find one that best suit your needs.

Potential Malicious Payload in PCI Expansion ROM

After all of the explanation, what would be the benefit of the larger memory footprint available to the malware "creator"? Let's see:

1. The larger memory footprint enables malware creator to place (at least) a simple file system infector inside the PCI expansion ROM, a compressed one. During PCI expansion ROM execution, the compressed file system infector could have the memory it requires through memory allocation with the PMM functions, provided that the BIOS implemented PMM—which is most likely the case in the last 3 to 5 years.
2. Another issue is malware creator might abuse the presence of the "permanent" memory allocated for PCI expansion ROM through the `pmmAllocate()` function, by using the permanent memory flag during the call to `pmmAllocate()`.
3. A rogue but simple network "interceptor" code might be possible given the jump in the memory footprint as well, and if it hides in the "permanent" memory, it could be troublesome.

This article has shown the building block required to carry out the task of creating a malicious PCI expansion ROM that is not limited by the scarce memory footprint during POST. This is of course a double-edged sword but it's important to create awareness in the computer security of the possibility. Otherwise, most of us would be left blind in the wake of such an advanced type of attack.

Malicious PCI Expansion ROM Detection

I have yet to see a real world malicious PCI expansion ROM. However, it should be noted that this kind of malware is not directly accessible to processes running in the host operating system. How would you detect such malware? Unfortunately, this task is the same as scanning most of the physical memory address space to make sure there's no malicious PCI expansion ROM in the "permanent" memory section of the RAM (memory allocated by `pmmAllocate()` with the permanent flag set to 1). One of the heuristic that comes to mind is whether the routine in the PCI expansion ROM contains function(s) that accesses the operating system. Well, you could think of other heuristics as well.

Now, let me talk a bit about the very near future. Windows 8, in one or more of its incarnation would *only* support signed PCI expansion ROM or EFI expansion ROM. I have yet to see, how it plays out in the real implementation, but that is a step forward. However, you should be aware that it would take years to migrate to Windows 8 and the installed base with present day PCI expansion ROM is far larger. I'll leave this matter for another article.

To sum up; breaking through the very small memory footprint during POST is possible through the use of the PMM functions.