EITN50 – Advanced Computer Security

Anatomy of an Exploit

by:

adsec03

Stefan Eng <atn08sen@student.lu.se>

Rasmus Olofzon <muh11rol@student.lu.se>

Introduction

1 Assignments

1.1 Assignment 1 – Running the exploit

Executing the python file exploit.py without the software started resulted in a command prompt displaying the following:

```
[*] Connecting to 192.168.0.1...
```

with nothing else happening.

Analyzing the exploit script reveals that it tries to connect to and exploit an instance of *Easy File Management Web Server v5.3* to trigger a remote buffer overflow. If the exploit is successful, it should close the management software and bring up an instance of the Windows calculator.

After starting the web management software and running the exploit again, this is what happens.

1.2 Assignment 2

In the exploit.py file a `payload` string is constructed. Most of the parts it is constructed with are hex values packed in an unsigned long little endian fashion (e. g. `pack('<L',0x10022aac)`). A relatively low-risk guess is that these hex values are memory addresses for different gadgets used in the attack. This is also in part confirmed by the comments supplied in the code. Some parts are also `\x90` repeated a number of times. `\x90` in x86 architecture corresponds to the NOP operation, which does nothing (stands for no operation). Last in the `payload` is a `shellcode` string, which after consulting `https://defuse.ca/online-x86-assembler.htm` seems to be x86 instructions.

This is a schematic representation of the stack before the overflow has occurred.

```
ESP+F0 > 00000000
⋮
ESP+194 > 00000000
ESP+198 > 01000101
ESP+19C > 024C3088
ESP+1A0 > 005829F8
fmws.005829F8
ESP+1A4 > 005829F8
fmws.005829F8
ESP+1A8 > 00554474
fmws.00554474
ESP+1AC > 00000668
ESP+1B0 > 00000001
ESP+1B4 > 024C3178
ESP+1B8 > 4244794D
ESP+1BC > 706F432C
ESP+1C0 > 67697279
ESP+1C4 > 475F7468
ESP+1C8 > 6E6F4875
ESP+1CC > 00000067
ESP+1D0 > 00000000
ESP+1D4 > 00000000
ESP+1D8 > 00000228
ESP+1DC > 00000003
ESP+1E0 > 00006469
ESP+1E4 > 00000000
⋮
ESP+1E8 > 00000000
ESP+200 > 656D616E
ESP+204 > 00000000
⋮
ESP+21C > 00000000
ESP+220 > 68746170
ESP+224 > 00000000
⋮
ESP+23C > 00000000
ESP+240 > 63736544
ESP+244 > 74706972
ESP+248 > 006E6F69
ESP+24C > 00000000
⋮
ESP+25C > 00000000
ESP+260 > 65636361
ESP+264 > 00007373
ESP+268 > 00000000
⋮
ESP+27C > 00000000
ESP+280 > 61736964
ESP+284 > 00656C62
ESP+288 > 00000000
ESP+28C > 00000000
```

These are the same relative positions after the overflow has occurred.

```
ESP+F0 > 90909090
⋮
ESP+13C > 90909090
ESP+140 > 1001D8C8 ImageLoa.1001D8C8
ESP+224 > 90909090
⋮
ESP+258 > 90909090
ESP+25C > 10010101 ImageLoa.10010101
ESP+260 > A445ABCF
ESP+264 > 10010125 ImageLoa.10010125
ESP+268 > 10022AAC ImageLoa.10022AAC
ESP+26C > DEADBEEF
ESP+270 > DEADBEEF
ESP+274 > 1001A187 ImageLoa.1001A187
ESP+278 > 1002466D ImageLoa.1002466D
ESP+27C > 90909090
⋮
ESP+28C > 90909090
ESP+290 > FDBBCADA
⋮
```

Clarification of which instructions the different gadgets are composed of.

call_edx	`JMP 0xffffffda` `ADD DWORD PTR [EAX],EDX`

ppr	`POP EBX`
	`POP ECX`
	`RETN`

crafted_jmp_esp	`JMP ESP`
test_bl	`ADD BYTE PTR DS:[EAX],AL`

kungfu	`MOV EAX,EBX`
	`POP ESI`
	`POP EBX`
	`RETN`
	`ADD EAX,5BFFC883`
	`RETN`
	`PUSH EAX`
	`RETN`

1.3 Assignment 3

After attaching to the software in the Olly debugger and putting a breakpoint at the address `0x00468702`, the following instructions were run before the execution hit the shell-code payload.

```
00: CALL DWORD PTR DS:[EDX+28]
01: MOV EDI,DWORD PTR SS:[ESP+264]
02: MOV BL,BYTE PTR DS:[EDI]
03: INC EDI
04: TEST BL,BL
05: MOV BYTE PTR SS:[ESP+40],BL
06: MOV DWORD PTR SS:[ESP+264],EDI
07: JNZ ImageLoa.1001D075
08: MOV EAX,DWORD PTR SS:[ESP+1C]
09: POP EDI
10: POP ESI
11: POP EBP
12: POP EBX
13: ADD ESP,24C
14: RETN
15: POP EBX
16: POP ECX
17: RETN
18: MOV EAX,EBX
```

```
19: POP ESI
20: POP EBX
21: RETN
22: ADD EAX,5BFFC883
23: RETN
24: PUSH EAX
25: RETN
26: JMP ESP
27: NOP
28: ⋮
```

where the marked entries were those deemed most vital for the exploit functionality. The purpose of each of these instructions is documented below.

26 Moves the program execution to the stack pointer. The execution is now under the attackers control.

24-25 The value from `EAX`, which contains the memory address to a `JMP ESP` instruction is pushed to the stack where the subsequent `RETN` executes it.

22-23 Add the calculated value to `EAX` in order to create the correct address for the `JMP ESP` instruction in `EAX`.

18,21 Move the offset of the `JMP ESP` address from `EBX` to `EBX`. The pop instructions are fed `DEADBEEF`.

15-17 Populate `EBX` and `ECX` with values from the stack. Pops address for the zero-value to be used to circumvent the `TEST BL BL` and `JNZ` instructions to `ECX`. Also pops the shifted address for the `JMP ESP` instruction (has to be shifted because it begins with a NULL byte otherwise) to `EBX`.

13 This instruction enables the stack-pivot to occur. The stack-pointer now points to the stack that the attacker created with the overflow.

00 Entry point of the exploit. Since the overflow enables the attacker to write anything to EDX, this call can be used to direct the control flow of the program. The call the executes address contained at the `EDX+28` memory location.

Since the attacker wants to come as close to the stack-pivot, `0x1001d89b` as possible, the data `\x7A\xD8\x01\x10` is found, which corresponds to the address `0x1001D87A` if read by a jump instruction.

To accommodate for the offset, 28 is subtracted and the resulting value of `0x1001D8C8` is written to `EBX` through the overflow.

With the other instructions doing the following:

01-07 Instructions that happens to be "in the way" of the sought after stack-pivot instruction. There is really no interest in these instructions as long as they do not disturb the intended exploit execution. In this case, 07 will disrupt the execution if not handled through the 01 and 04 instructions.

08-12 `POP` and `MOV` instructions that don't really affect the execution and can be ignored.

The filler is there to align the different overflow values to the correct stack-positions. Since there is a instruction that moves a value from the memory-address contained at `ESP+264` before the `JNZ` instruction, the zero-value for circumventing the jump needs to be located at `ESP+264`. And this can be arranged by utilizing the `NOP` as filler.

1.4 Assignment 4

Firstly, since the memory addresses specified in exploit.py are constants, this tells us that at least ImageLoad.dll (where the parts of the payload that are not **`shellcode`** are injected) does not have ASLR enabled. However, let us look for some answers:

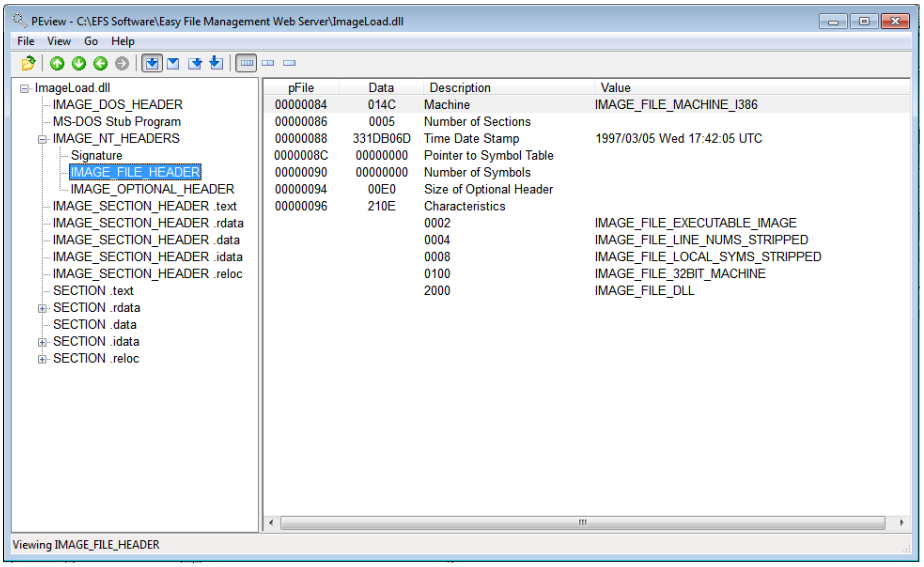
From Microsoft's documentation for ISVs (Independent Software Vendor) (**`https://msdn.microsoft.com/en-us/library/bb430720.aspx`**) we find that:

- "ASLR moves executable images into random locations when a system boots [..]"

- "By default, Windows Vista and later will randomize system DLLs and EXEs, but DLLs and EXEs created by ISVs must opt in to support ASLR using the /DYNAMICBASE linker option."
- "ASLR also randomizes heap and stack memory:
  - When an application creates a heap in Windows Vista and later, the heap manager will create that heap at a random location to help reduce the chance that an attempt to exploit a heap-based buffer overrun succeeds. Heap randomization is enabled by default for all applications running on Windows Vista and later.
  - When a thread starts in a process linked with /DYNAMICBASE, Windows Vista and later moves the thread's stack to a random location to help reduce the chance that a stack-based buffer overrun exploit will succeed"

This tells us that if ASLR is enabled, the location for executables is randomised at system boot, the location for application heap is randomised at heap creation and the location for a thread's stack is randomised at process start. From this

Figure 1: `IMAGE_FILE_RELOCS_STRIPPED` is not set => ImageLoad.dll is relocatable.



(`https://stackoverflow.com/questions/39189477/how-do-i-determine-if-an-exe-or-dll-participa` answer to a StackOverflow question we see how we can check if the ISV has opted in to support ASLR, as described above. This is also described in Microsoft's own documentation on file headers:

- In the `IMAGE_FILE_HEADER` structure we can see that if the flag `IMAGE_¬FILE_RELOCS_STRIPPED = 0x0001` is set in the Characteristics field, this means that the file is not relocatable. (`https://msdn.microsoft.com/en-us/library/windows/desktop/ms680313(v=vs.85).aspx`)
- In the `IMAGE_OPTIONAL_HEADER` structure we can see that the if flag `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE = 0x0040` is set in the DLLCharacteristics field, it means that the file can be relocated at load time (relate this to /DYNAMICBASE from above). (`https://msdn.microsoft.com/en-us/library/windows/desktop/ms680339(v=vs.85).aspx`). A file must be relocatable in order to have ASLR enabled.

As suggested in the StackOverflow answer mentioned above, we installed PEView and checked the file headers for ImageLoad.dll and fmws.exe. See figures 1, 2, 3 and 4 for this.

From this can be seen that ImageLoad.dll is relocatable but ASLR is not enabled, and that fmws.exe is not relocatable and ASLR is not enabled.

Figure 2: `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` is not set => ImageLoad.dll is not relocated at load time => ASLR not enabled.

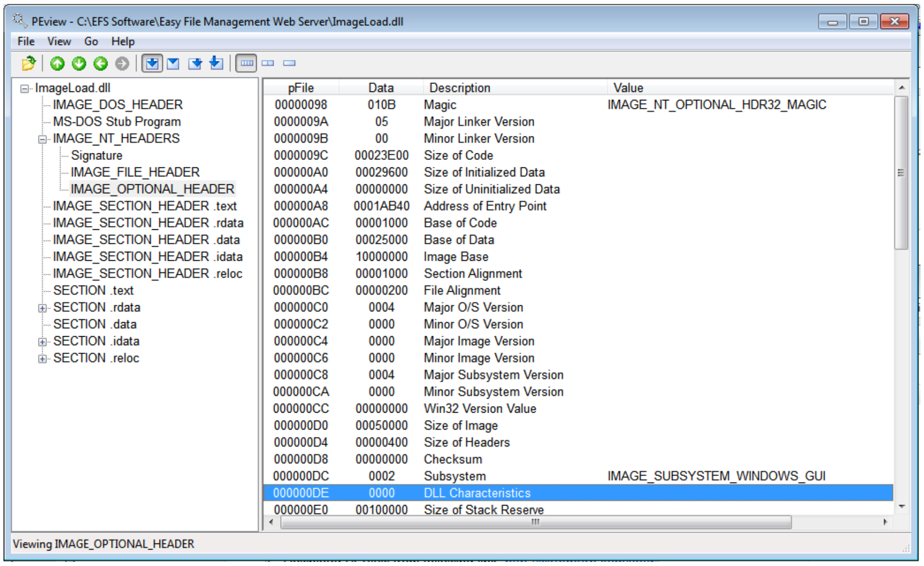


Figure 3: `IMAGE_FILE_RELOCS_STRIPPED` is set => fmws.exe is not relocatable.

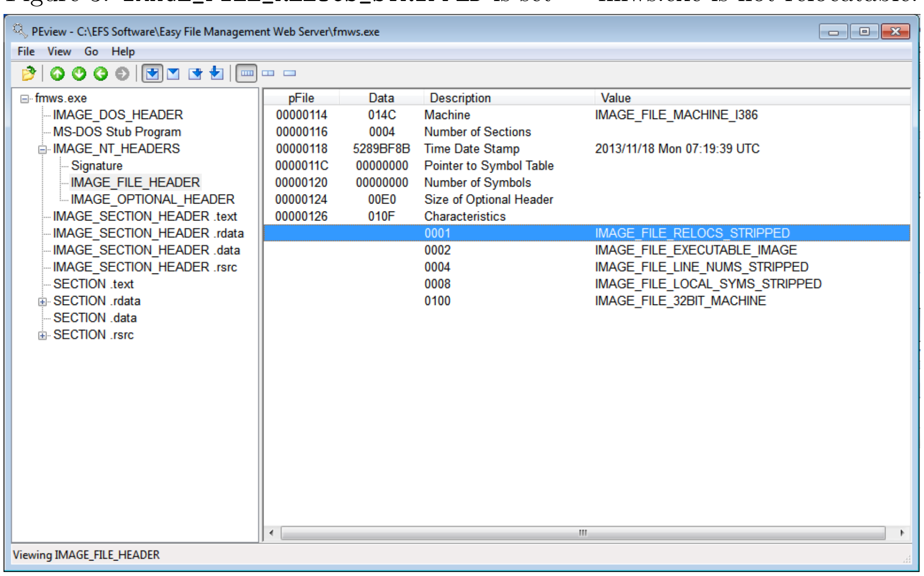
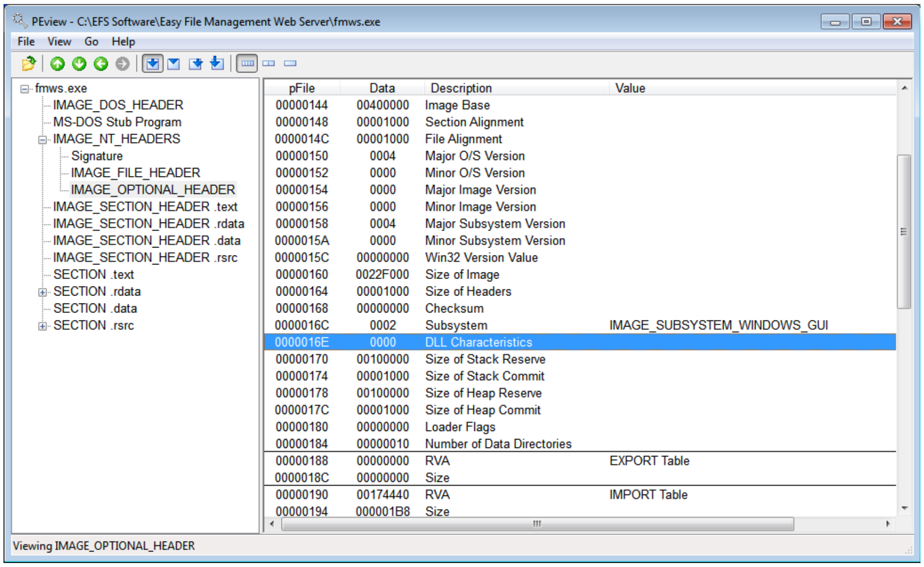


Figure 4: `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` is not set => fmws.exe is not relocated at load time => ASLR not enabled.



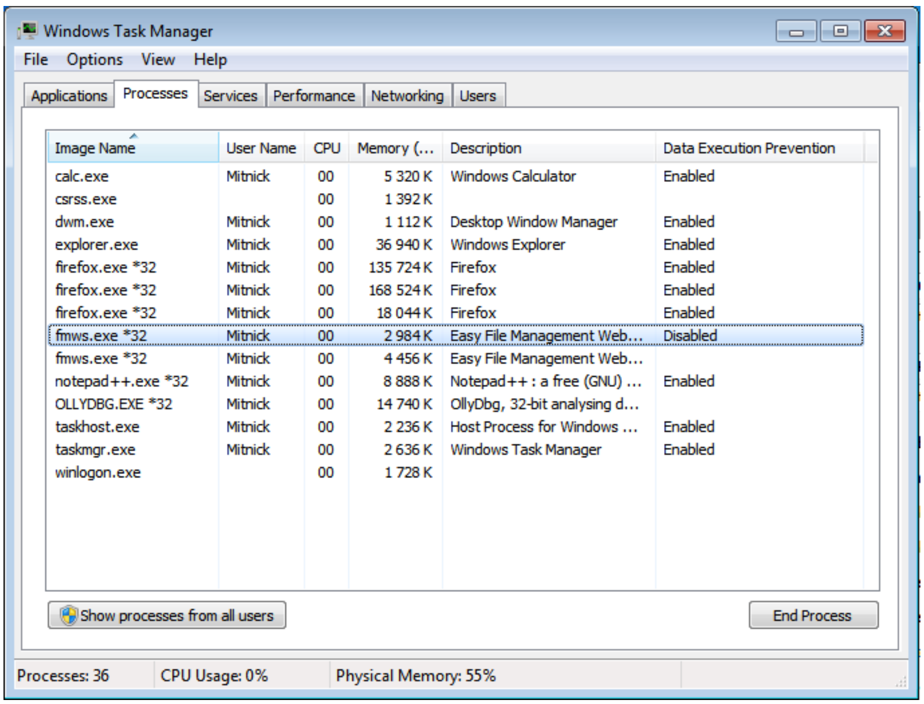
1.5 Assignment 5

The exploit utilizes the unprotected parts of the code to find gadgets consisting of ”trusted” instructions inside of the genuine program code. Since the gadgets are located in non-protected memory, they can be referenced through static memory locations pushed to the stack. This in turn enables the exploit to chain different gadgets together to manipulate the stack and register in a way that ultimately enables the attacker to force the execution onto the stack and execute the shell-code payload.

1.6 Assignment 6

One way to check if DEP is enabled for a program in Windows 8 is to open the Task Manager, go to the ’Processes’ tab, open ’View’ > ’Select Columns...’ and make sure the ’Data Execution Prevention’ check button is checked. Press ’OK’. This then shows in the table if DEP is enabled for the different processes. When doing this on the virtual machine we could see that DEP was disabled for fmws.exe (see figure 5).

Figure 5: Screenshot of task manager with column DEP enabled, showing that DEP is disabled for fmws.exe.



1.7 Assignment 7

We have found different approaches to write an exploit that bypasses DEP:

- Find gadgets so that we have an uninterrupted ROP chain (without e. g. NOPs). A tool that could help find such a chain is `mona.py`, `https://¬ github.com/corelan/mona`
- "change the DEP settings for the current process before running shellcode" (`https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-`
- Use Windows functions. (`https://www.exploit-db.com/exploits/42304/`) and (`https://www.exploit-db.com/exploits/42186/`) both use Windows function calls (namely to `VirtualProtect()`, which is also described in corelan site linked above (rubik's cube) as a strategy for bypassing DEP.