Skypher

Bypassing Export address table Address Filter (EAF)

Posted by SkyLined on November 17th, 2010 in Uncategorized · 0 Comments

In early September this year Microsoft released their Enhanced Mitigation Experience Toolkit v2.0 (EMET), which includes a new mitigation called Export address table Address Filter (EAF). I decided to have a look at how this mitigation attempts to prevent exploits from succeeding and how an attacker might bypass it.

EAF works by setting a hardware breakpoint on the export address tables of the ntdll.dll and kernel32.dll modules in a process. When the breakpoint is triggered, EMET tries to determin if the code that is trying to access the export address table is valid code for that process or malicious code injected into the process through an exploit.

This works because most exploits will at some point inject and run shellcode into the target process and one of the first thing most shellcodes do is determine where certain functions are loaded in memory by reading the export address table of ntdll.dll and/or kernel32.dll. At that point EAF should detect the shellcode and terminate the process, preventing the exploit from succeeding.

I tested the effectiveness of EAF against one of my shellcodes. All my shellcodes use the same common techniques to find the base address of functions, which is to scan the export address table. So, EAF should prevent my shellcode and most others from working.

I use testival to run my shellcode. The shellcode shows a popup dialog box and triggers and int3 breakpoint. Here's the output of w32-testival.exe for a successful run:

```
C:\Dev\Shellcode\w32-msgbox-shellcode>w32-testival [$]=ascii:w32-msgbox-shellcode.bin eip=$ --verbose --eh --eh
Allocating 0x1000 bytes of memory... ok. (address: 0x00030000)
Setting data and registers:
[0x00030000] = 8C bytes of data.
eax = 0xDEADBEEF (default)
ecx = 0xDEADBEEF (default)
edx = 0xDEADBEEF (default)
ebx = 0xDEADBEEF (default)
esp = ??? (unmodified)
ebp = 0xDEADBEEF (default)
esi = 0xDEADBEEF (default)
edi = 0xDEADBEEF (default)
```

```
eip = 0x00030000 ($)
Registering Structured Exception Handler (SEH)...ok.
Registering Vectored Exception Handler (VEH)...ok.
Executing shellcode by jumping to 0x00030000...First chance debugger breakpoint
exception at 0x0003008B.
Second chance debugger breakpoint exception at 0x0003008B.

C:\Dev\Shellcode\w32-msgbox-shellcode>
```

(You cannot see the popup dialog box in this output of course, but you will notice that the int3 breakpoint at the end of the shellcode was executed).

After enabling EMET I tried executing the shellcode again and found that it still worked!? So I contacted my friends at MS who developed the tool. They explained that in order to install the mitigations, EMET needs to create a new thread in the target process, which means the protection is not enabled immediately. So, I added a new feature to Testival to allow me to wait a bit before executing the shellcode and tried again. This time EMET successfully block the shellcode:

```
C:\Dev\Shellcode\w32-msgbox-shellcode>w32-testival [$]=ascii:w32-msgbox-
shellcode.bin eip=$ --verbose --eh --eh --delay=1000
Allocating 0x1000 bytes of memory... ok. (address: 0x00030000)
Setting data and registers:
[0x00030000] = 8C bytes of data.
eax = 0xDEADBEEF (default)
ecx = 0xDEADBEEF (default)
edx = 0xDEADBEEF (default)
ebx = 0xDEADBEEF (default)
esp = ??? (unmodified)
ebp = 0xDEADBEEF (default)
esi = 0xDEADBEEF (default)
edi = 0xDEADBEEF (default)
eip = 0x00030000 ($)
Registering Structured Exception Handler (SEH)...ok.
Registering Vectored Exception Handler (VEH)...ok.
Waiting for 1000 milliseconds...ok.
Executing shellcode by jumping to 0x00030000...First chance single step exception
at 0x00030054: A trace trap or other single-instruction mechanism signaled that
one instruction has been executed.
Second chance exception 0xC0000409 at 0x00030054.

C:\Dev\Shellcode\w32-msgbox-shellcode>
```

There is no popup this time and instead of the int3 at the end of the shellcode terminating the process, the process is terminated before the shellcode is finished

by a single step exception. This is because the EAF mitigation has detect that the shellcode accessed the export address table and triggered the single step exception to terminate the application rather than allow the shellcode to continue.

So, EAF appears to works well against exploits that use common shellcode (except maybe in the first few milliseconds after the process is started). So, how do we bypass it?

I assumed that EAF works by checking where the instruction that is accessing the export address table is located: if it is located inside the code segment of a module loaded in memory, it must be valid code and allowed to continue, and otherwise it must be malicious and the process should be terminated. So, I decided to create a new version of my shellcode that uses a ret-into-libc-like approach to reading the address table. First, the shellcode should find out where ntdll.dll is loaded in memory as usual. Second, it should find out where the code segment for ntdll.dll is located. Third is should scan the code segment for a specific instruction sequence that can be used to read arbitrary memory. and Finally, it should call this instruction sequence to read the export address table, rather than read it directly. If all goes well, EAF will not intervene because it assumes that ntdll.dll is attempting to read the export address table rather than my shellcode.

It turns out that the RtlGetCurrentPeb function has a sequence that I can use and that this is static across OS versions and SPs. Even better, it is 4 bytes long, which means it is easy to write code that finds it:

ntdll32!RtlGetCurrentPeb:
64a118000000 mov eax,dword ptr fs:[00000018h]
8b4030 mov eax,dword ptr [eax+30h]
c3 ret

This code can be used to read arbitrary memory by setting EAX to point 30 bytes before the memory address you want to read and calling the second instruction.

It also turns out that the export address table is located at the start of the code segment of ntdll.dll. The first time I tested my code, it was blocked by EAF while scanning for the instruction sequence in the code segment. Luckily, the RtlGetCurrentPeb function is not located anywhere near the start of the code segment, so it was relatively easy to avoid this by having my shellcode skip over the first 0×1000 bytes of the code segment when scanning for the sequence.

So, here is the result for my modified shellcode, which is only 30 bytes larger than the original:

C:\Dev\Shellcode\w32-msgbox-shellcode>w32-testival [$]=ascii:w32-msgbox-shellcode-eaf.bin eip=$ –verbose –eh –eh –delay=1000
Allocating 0×1000 bytes of memory… ok. (address: 0×00030000)

Setting data and registers:
[0x00030000] = AB bytes of data.
eax = 0xDEADBEEF (default)
ecx = 0xDEADBEEF (default)
edx = 0xDEADBEEF (default)
ebx = 0xDEADBEEF (default)
esp = ??? (unmodified)
ebp = 0xDEADBEEF (default)
esi = 0xDEADBEEF (default)
edi = 0xDEADBEEF (default)
eip = 0×00030000 ($)
Registering Structured Exception Handler (SEH)…ok.
Registering Vectored Exception Handler (VEH)…ok.
Waiting for 1000 milliseconds…ok.
Executing shellcode by jumping to 0×00030000…First chance single step exception at 0x76FBA045: A trace trap or other single-instruction mechanism signaled that one instruction has been executed.
First chance single step exception at 0x76FAFFCE: A trace trap or other single-instruction mechanism signaled that one instruction has been executed.
First chance single step exception at 0x76FAFFCE: A trace trap or other single-instruction mechanism signaled that one instruction has been executed.
First chance single step exception at 0x76FAFFCE: A trace trap or other single-instruction mechanism signaled that one instruction has been executed.
<snip>
First chance single step exception at 0x76FAFFCE: A trace trap or other single-instruction mechanism signaled that one instruction has been executed.
First chance debugger breakpoint exception at 0x000300AA.
Second chance debugger breakpoint exception at 0x000300AA.

C:\Dev\Shellcode\w32-msgbox-shellcode>