Microsoft | Trustworthy Computing

Inside EMET 4.0

Elias Bachaalany

Microsoft Security Response Center (MSRC)

REcon 2013, Montreal

Overview

- Motivation for this presentation
- What EMET is and is not
- Breaking down EMET
  - EMET Agent
  - Mitigation engine
  - Certificate trust crypto extension
  - EMET UI
- Q&A

Motivation

"There is nothing hidden under the sun" -- Old Greek proverb*

* Reverse engineers agree on that!

Motivation

- This talk is **<u>technical</u>**
- Developers will enjoy it
  - Hackers alike ;)
- Giving back to the community:
  - EMET is result of contribution from various talented individuals and the security community
- EMET was never about security by obscurity
- Sharing will open the door to new ideas and mitigation techniques
- Help developers write EMET compatible code

What is EMET

- Stands for: Enhanced Mitigation Experience Toolkit
- Free utility that helps prevent vulnerabilities in software from being successfully exploited
- Employs technology to counter common exploitation techniques
- Works without exact knowledge of the exploit

What EMET is NOT

- It is not an Antivirus: Unlike antivirus, EMET does not rely on signatures rather on the runtime behavior of the program it protects
- It is not a “Fool proof exploit mitigation solution”
  - It helps raise the cost of exploitation
  - Cat and mouse game
  - It is easier to destroy than to build
- Not good against logic bugs: bugs in APIs can also lead to exploitation (without resorting to memory bugs)

EMET Agent

- Responsible for handling:
  - Tray icon notification
  - Certificate trust rule validation
  - Event logging
  - Telemetry
- User mode process (managed code):
  - Supersedes “EMET_notifier.exe” (< 4.0)
  - Runs with the privilege of the logged in user
  - Pluggable: plugins are internally known as Subsystems

Mitigation engine

Overview

- Written in C++ and *some* inline x86 assembly
- Compiled as EMET[64].dll
- Gets injected into processes via "Windows Application Compatibility Infrastructure" aka Shim Infrastructure*
  - A shim database (*.sdb) is created by EMET's UI to define which processes should get EMET shim injected into them
- Works intimately with "EMET Agent"
  - Uses mailslots for IPC

* http://technet.microsoft.com/en-us/library/dd837644(v=WS.10).aspx

Exploit Mitigation Technologies

- List of mitigations technologies:
  - DEP
    - Provided by the OS and configurable via EMET
  - EAF – Export Address Table Access filtering
  - Heapspray protection
  - SEHOP
    - Provided by the OS (Vista+)
    - Configurable via EMET (>=Win7)
  - Mandatory ASLR
    - Provided by the OS (KB2639308)
  - Reserve NULL page
    - Provided by the OS (MS13-031)

Exploit Mitigation Technologies

- ...continued:
  - ROP mitigations
    - Stack Pivot
    - Simulate execution flow
    - Caller checks
  - API behavior checks
    - Memory protection change
    - Loading DLLs from UNC path
  - Hardened protection
    - Deep hooks
    - Anti detours
    - Banned APIs

DEP

- Data Execution Prevention: Prevents code residing in non-executable (**stack**, data, **heap**) memory pages from executing
- Once EMET gets injected into the process, it calls the "SetProcessDEPPolicy" API to turn on/off DEP for the process.

ASLR – Address Space Layout Randomization

- Introduced in Windows Vista
- Randomization of address space layout
- Applications opt-in by linking executable files with /DYNAMICBASE

- EMET brings ASLR to:
  - Modules **not** built with /DYNAMICBASE
  - Only to dynamically loaded DLLs (i.e: delay import, LoadLibrary(), ...)
  - Vista+

ASLR

1. EMET intercepts calls to **ntdll!NtMapViewOfSection**
   - One code path is: kernel32!LoadLibrary() -> ntdll!LdrLoadDll -> ... -> ntdll!NtMapViewOfSection
2. If the DLL was **not** compiled with /DYNAMICBASE && it has a relocation table then:
   - Un-map the section (reverting Step #1)
   - Reserve one page of memory at the preferred image base address
   - Re-map the section -> thus forcing the DLL to get the DLL relocated by the OS (redo Step #1)

SEHOP – SEH Overwrite Protection

- Introduced in Windows Vista SP1 (system wide only)
- Verifies the "Exception Handler Chain Integrity" before dispatching the SEH handlers
- Applications can opt-in by setting the "DisableExceptionChainValidation" Image File Execution Option (IFEO) to zero in Windows 7+
  - EMET will act as a UI to toggle this IFEO option

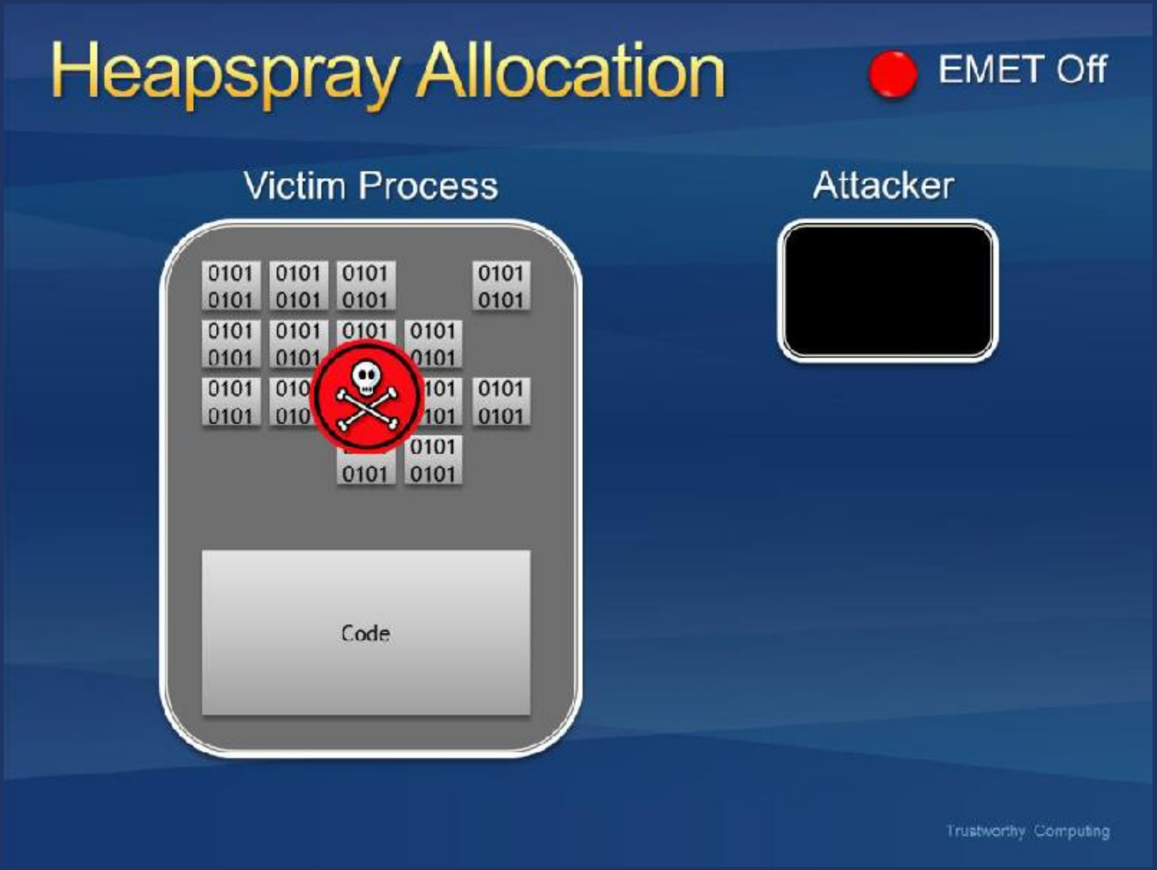
SEHOP

- EMET 4.0 brings SEHOP opt-in to all OS versions prior to Windows 7
  - EMET re-implements the same logic as the OS for exception handler chain integrity checks
  - A vectored exception handler (VEH) is registered so it checks the SEH chain integrity

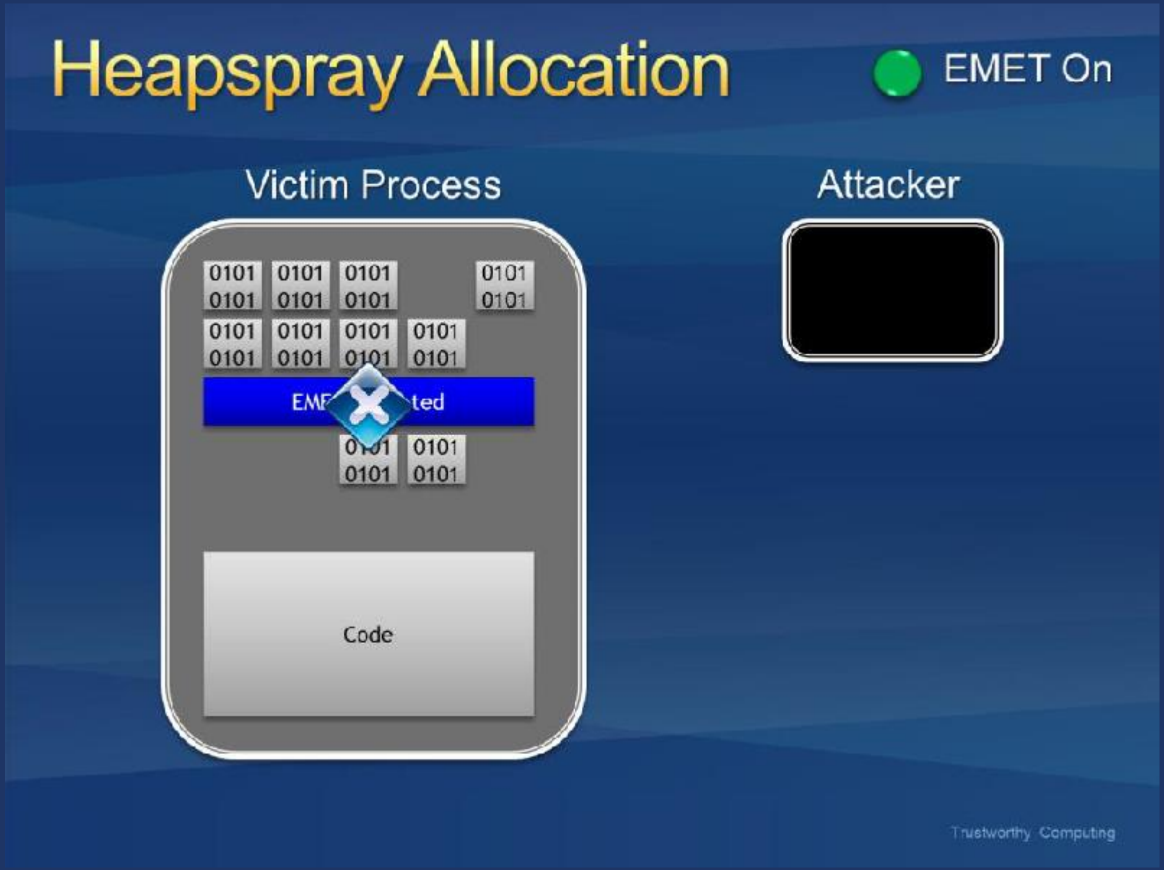
Heap Spray Allocations

- Not provided by the OS
- Aims to break heap spraying by reserving memory pages at the most popular heap-spray addresses:
  - 0x0a0a0a0a;0x0b0b0b0b;0x0c0c0c0c;0x0d0d0d0d;0x09090909;0x14141414; ....
  - This list can be configured from the registry (per-process)
    - _settings_\{app-guid}\heap_pages (REG_SZ)

Heap Spray Allocations



Heap Spray Allocations



EAT Access Filtering (EAF)

- Filters access to the Export Address Table (EAT):
  - Any access to the EAT **must** originate from a loaded module => shell/JITted code will trigger the filter
  - For each thread, EMET adds two debug registers to monitor any access to the EAT of **kernel32.dll** and **ntdll.dll**

```
// Protect EAT for kernel32 and ntdll
eat_hwbp_add_module("kernel32.dll");
eat_hwbp_add_module("ntdll.dll");
```

- This protection is effective against EMET-agnostic shellcode

EAT Access Filtering (EAF)

- Drawbacks:
  - Can be defeated with many tricks
  - DRMed code is allergic to EAF
- Resolving APIs via parsing system structures:

```
shellcode:025405A6 xor     edx, edx
shellcode:025405A8 mov     edx, fs:[edx+30h]; Get PEB
shellcode:025405AC mov     edx, [edx+0Ch]; Get PEB_LDR_DATA struct
shellcode:025405AF mov     edx, [edx+14h]
shellcode:025405AF
shellcode:025405B2
shellcode:025405B2 loc_25405B2:; CODE XREF: shellcode:0254062A↓j
shellcode:025405B2 mov     esi, [edx+28h]
shellcode:025405B5 movzx   ecx, word ptr [edx+26h]
shellcode:025405B9 xor     edi, edi
```

Bottom-up Randomization

- Bottom-up Randomization
  - Reserve a random number of 64K regions via VirtualAlloc()
  - This will make future memory allocations less predictable
  - Provides entropy to images that have been randomized via mandatory ASLR
  - Win8 natively supports this

ROP – Overview

- ROP mitigations are a result of the **BlueHat 2012 Prize**

- EMET's ROP mitigations are based on **Ivan Fratric**'s work* (the 2nd prize winner)

- EMET implements four out of six mitigations from Ivan's **ROPGuard**

* https://code.google.com/p/ropguard/

ROP – Implementation objectives

- The re-implementation emphasizes:
  - **Speed:** ROP checks should be as fast as possible
  - **Code maintainability and portability:** the code must be easy to maintain and to port to other architecture (if needed)
  - **Compatibility:** ROP checks should be compatible with as much applications as possible
  - **Reuse of existing supporting libraries:** Use existing and time proven API hooking and disassembly engine

Note: EMET 4.0 implements ROP mitigations for 32-bit processes only

ROP – Terms and general premises

- Definition of "Critical functions"
  - They are functions that are important for the attacker to call in order to set stage for a more elaborate code execution
- Some critical functions that are used via ROP:
  - <u>Returning to VirtualProtect:</u> make the stack area executable
  - <u>Returning to VirtualAlloc:</u> allocate executable memory
  - <u>Returning to LoadLibrary:</u> load a remote DLL and achieve code execution

ROP – Terms and general premises

- There are around 50 APIs that EMET deems as "critical functions"
- Critical functions are **hooked** and **redirected** to a common stub that does the extra validation before letting the APIs resume execution
- **Simply put:** only "*proper* use" of critical APIs will be allowed

ROP – Supporting Libraries - MSDIS

- MSDIS is a disassembler library
- It is used internally by
  - The debugger engine (dbgeng): Windbg, cdb, etc…
  - Visual Studio, etc…
- Can disassemble code for various machine architectures
  - X86
  - AMD64
  - ARM, etc…
- Robust and provides many functionalities needed by EMET:
  - Disassembling
  - Code simulation

ROP – Supporting Libraries - Detours

- Detours* is a Microsoft Research Project
- Robust and portable API hooking and binary instrumentation library, supporting:
  - X86
  - AMD64
  - ARM, etc…
- However, Detours, as is, is not enough to support EMET

* http://research.microsoft.com/en-us/projects/detours/

ROP – Supporting Libraries - Detours

- Detours has been modified to support:
  - Redirecting all functions to the same stub
    - To redirect all critical APIs to the same stub (Let us call that stub: **ROPCheck** stub)
  - User callbacks for Pre/Post code generation
    - To generate custom code for each detoured API

```
// Setup the xDetours params
xDetoursParams.PreCodeGen  = RopCheckPreCodeGen;
xDetoursParams.PostCodeGen = RopCheckPostCodeGen;

// Associate the context structure
RopCheckCodeGenStruct RopCheckCodeGenVar = {0};
xDetoursParams.Context = &RopCheckCodeGenVar; //
```

ROP – Supporting Libraries - Detours

- User controlled "copied bytes" count
  - This helps achieve anti-trampoline bypasses
  - It is not fool proof
- Shellcode executing the prolog body in the shellcode then jumping past the detour via "jmp ApiAddr+5" will crash

```
// Randomize the trampoline byte count
xDetoursParams.nCopyBytes = 1 + (rand() % 3);
```

ROP – A typical protected critical function

- In this example, kernel32!VirtualAllocEx() is protected
  - Its original bytes are copied
  - A jump to the detoured function is put instead
    - 5 bytes are consumed (0xE9 + sizeof(DWORD))
  - Note: anti-detours is applied (notice the 0xCC filler)

```
kernel32.dll:77CA5909
kernel32.dll:77CA5909 ; LPVOID __stdcall VirtualAllocEx(HANDLE hProcess,
kernel32.dll:77CA5909 VirtualAllocEx proc near
kernel32.dll:77CA5909 jmp     GuardedVirtualAllocEx
kernel32.dll:77CA5909 VirtualAllocEx endp
kernel32.dll:77CA5909
kernel32.dll:77CA5909 ; ------------------------------------------------
kernel32.dll:77CA590E db 0CCh ; ¦; Anti detours/trampoline bypass
kernel32.dll:77CA590F ; ------------------------------------------------
kernel32.dll:77CA590F
kernel32.dll:77CA590F VirtualAllocEx_after_copied_bytes:
kernel32.dll:77CA590F ; CODE XREF: GuardedVirtualAllocEx+2C↑j
kernel32.dll:77CA590F jmp     o_kernelbase_VirtualAllocEx
```

ROP – A typical protected critical function

- This stub is generated by the pre and post code gen callbacks
- The protected function's new detoured body (1/2):

```
; LPVOID __stdcall GuardedVirtualAllocEx(HANDLE hProcess, LPVOID lpAddre
GuardedVirtualAllocEx proc near; CODE XREF: j_GuardedVirtualAllocEx↓j

hProcess= dword ptr  4
lpAddress= dword ptr  8
dwSize= dword ptr  0Ch
flAllocationType= dword ptr  10h
flProtect= dword ptr  14h

push    0AADC2861h; Function ID = Encoded API Proc Addr
pusha; << Save all GP registers
pushf; << Save the flags

push    esp; Save the stack pointer -> points to the registers array
call    near ptr ROPCheck; DWORD WINAPI RopCheck(PDWORD Registers)
popf; >> Pop the  flags
popa; >> Pop GP regs
add     esp, 4; >> POP func ID
```

ROP – A typical protected critical function

- The protected function's new detoured body (2/2):

```
// Start pushing original arguments
push    dword ptr [eax+14h]
push    dword ptr [eax+10h]
push    dword ptr [eax+0Ch]
push    dword ptr [eax+8]
push    dword ptr [eax+4]
push    offset after_API_call
// Copied bytes
mov     edi, edi
push    ebp
mov     ebp, esp
pop     ebp
jmp     VirtualAllocEx_after_copied_bytes
; ---------------------------------------------------------------------------
db 0CCh ; ¦; Special Marker
; ---------------------------------------------------------------------------

after_API_call:; DATA XREF: GuardedVirtualAllocEx+21↑o
push    eax; << API return value
call    near ptr ROPCheckEnd; Post ROP checks (restore LastError, ...)
retn    14h; stdcall, purge params and return
GuardedVirtualAllocEx endp
```

ROP – A typical protected critical function

- ROPCheck() will be called before resuming execution in the original API
- In short, ROPCheck() does the following:

```
// This function is common to all detoured functions
DWORD WINAPI RopCheck(PDWORD Registers)
{
  // Parse parameters
  PDWORD pRSP           = (PDWORD)Registers[R_X86_ESP];
  PBYTE CalledAddressE = (PBYTE)*pRSP;
  PBYTE CalledAddress  = (PBYTE)DecodePointer(CalledAddressE);
  PBYTE ReturnAddress  = (PBYTE)*(pRSP+1);
  // Check banned APIs
  // Check stack pointer
  // Check the caller
  // Simulate execution flow
  // Special checks on LoadLibrary family
  // Special checks on VirtualAlloc/VirtualProtect family
  // ANY VIOLATION -> Report and Terminate the program
  // Otherwise: Resume API execution
}
```

ROP – The mitigations

- We covered all the background material
- Any questions so far?
- Let us now describe each ROP mitigation

ROP – Stack Pivot

- The attacker sometimes has control over the heap data and not the stack
- A “stack pivot” gadget is used to swap the stack pointer with an attacker controlled register (pointing to controlled data, usually on the heap)
- The typical gadget (if EAX was under the attacker’s control):
  - XCHG EAX, ESP
  - RET

ROP – Stack Pivot

Upon entering a critical function, EMET checks if ESP is within the thread's defined stack area (in the TIB)

```
DWORD StackBottom, StackTop;
GetStackInfo(&StackBottom, &StackTop);
if (((DWORD_PTR)pRSP < StackBottom) || ((DWORD_PTR)pRSP >= StackTop))
  ReportStackPivot(...);
```

ROP – Caller checks

- EMET disassembles backwards from the return address (and upwards) and verifies that TARGET is **CALL**ed and not **RET**urned or **JMP**ed into

- Normal API call scenario:
  - PUSH argN
  - PUSH …
  - PUSH arg1
  - CALL kernel32!VirtualAlloc ; <- target
  - TEST EAX, EAX ; <- Return address
  - JE loc_123

ROP – Caller checks

- ROP scenario (memory @ EAX is attacker controlled):
  - EAX -> memory contents
    - [address of **VirtualAlloc**, GADGET2_ADDRESS, arg1, …, argN, unused, OtherApiCall, GADGET3_ADDRESS, arg….]
- After a bug is triggered and EIP is controlled, the starting gadget could be a stack-pivot gadget:
  - XCHG EAX, ESP
  - RET <- returns to VirtualAlloc, then returns to Gadget 2
- Gadget 2:
  - POP EBP
  - RET <- returns to OtherApiCall then returns to Gadget 3

ROP – Caller checks

- A critical function (1) is reached (in this case VirtualAlloc)
- The return address is captured
- The registers are captured and passed to MSDIS
  - All general purpose registers are required to resolve indirect call target
- Heuristically disassemble backwards from the return address until we could disassemble a call

ROP – Caller checks

- Compute the call target and see if it leads back to the critical function (1)
- If no **CALL** instruction was found then we probably have a ROP or JOP scenario
  - Notify the user and terminate the process

ROP – Caller checks

- **<u>Backward disassembly table</u>**
  - The order of the instruction length is based on the most frequent "CALL opcode" sequence found in the majority of programs
  - This ordering increases the likelihood of finding a **CALL** in the first iteration

```
//
// Call OpCode check priority and instruction length
static const unsigned char CallOp32[] =
{
  6, // call [reg+disp32], call [loc32]
  5, // call rel
  2, // call reg, call [reg]
  3, // call [reg+disp8]
  7, // call [reg1+reg2+disp32] and other calls
};
```

ROP – Caller checks

Checking if previous instruction is a call

```
static bool CheckPreviousInstructionIfCall(
  DIS *Dis,
  PBYTE ReturnAddress,
  PBYTE CallTarget,
  PDWORD Registers)
{
  bool ok = false;
  // Bind the registers with this instance
  Dis->PvClientSet(Registers);
  // Try to disassemble and see if it is a call instruction
  for (size_t i=0; i<_countof(CallOp32); i++)
  {
    PBYTE DisAsmBuf = ReturnAddress - CallOp32[i];
    if (Dis->CbDisassemble((DIS::ADDR)DisAsmBuf, DisAsmBuf, 20) == 0)
      continue;
    DIS::INSTRUCTION Instr;
    DIS::OPERAND Opr[2];
    if (   !Dis->FDecode(&Instr, Opr, _countof(Opr))
        || Instr.opa != DISX86::opaCall)
      continue;
    ok = CheckCallTarget(
        Dis,
        &Opr[0],
        CallTarget);
    if (ok)
      break;
  }
  return ok;
}
```

Mitigation Engine - ROP – Caller checks

- It is not as **simple** as that!
- The compiler legitimately does some weird stuff:

```
MSO.DLL:6780D7B5 55               push    ebp
MSO.DLL:6780D7B6 8B EC            mov     ebp, esp
MSO.DLL:6780D7B8 5D               pop     ebp
MSO.DLL:6780D7B9 FF 25 24 1A+     jmp     off_673A1A24 ; API address
```

- The program may be using Detours (or alike) itself:

```
OLEACC.dll!6E1EB2B1  push        ebp
OLEACC.dll!6E1EB2B2  mov         ebp,esp
OLEACC.dll!6E1EB2B4  mov         eax,dword ptr [lpfnVirtualAllocEx]
OLEACC.dll!6E1EB2B9  test        eax,eax
OLEACC.dll!6E1EB2BB  je          MyVirtualAllocEx+0Eh
OLEACC.dll!6E1EB2C1  pop         ebp
OLEACC.dll!6E1EB2C2  jmp         eax ; VirtualAllocEx
```

- All those above (and more) are legitimate cases, we have to handle them!
- CallerCheck have to find the right balance:
  - Handle legitimate cases while blocking real ROP attempts
  - There is no perfect solution

ROP – Simulate Execution Flow

- EMET simulates execution forward from a critical function call
- Simulate forward and follow the return addresses
  - 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**

ROP – Simulate Execution Flow

- In the case of chained ROP gadgets:
  - After a critical function returns, it will be followed by another gadget
    - and not a **CALL** instruction (in most cases)
  - Each gadget will execute a few simple instructions and **RET**urn again to the following gadget OR to another critical API

ROP – Simulate Execution Flow

- Sample memory dump with pointers to gadgets and parameters values

```
0x0018CB98  00000000 aaaaaaaa 6da612cc 6d8ff623 aaaaaaaa aaaaaaaa
0x0018CBB0  aaaaaaaa 6d81bdd7 aaaaaaaa aaaaaaaa aaaaaaaa 00002000
0x0018CBC8  6d802a88 6da612cc 6d97ed06 aaaaaaaa aaaaaaaa aaaaaaaa
0x0018CBE0  6d970a50 aaaaaaaa aaaaaaaa aaaaaaaa 6d8011ac 00000000
0x0018CBF8  00004000 aaaaaaaa 6d824c7c aaaaaaaa aaaaaaaa 6d808150
0x0018CC10  6d85a181 aaaaaaaa aaaaaaaa 00003000 aaaaaaaa 6d96fa23
0x0018CC28  000001b8 90909000 90909090 41284068 9090c300 90909090
0x0018CC40  90909090 90909090 90909090 90909090 90909090 90909090
0x0018CC58  90909090 90909090 90909090 90909090 90909090 90909090
0x0018CC70  90909090 90909090 90909090 90909090 90909090 90909090
0x0018CC88  90909090 90909090 90909090 90909090 90909090 90909090
0x0018CCA0  90909090 90909090 90909090 90909090 90909090 90909090
0x0018CCB8  90909090 90909090 90909090 90909090 90909090 90909090
```

ROP – Simulate Execution Flow

```
0x00000000, // null to avoid crashing
0xAAAAAAAA, // unused
0x6DA612CC, // writeable memory to avoid crashing
0x6D8FF623, // (2) return to register load
0xAAAAAAAA, // (2.A) ESI, unused
0xAAAAAAAA, // (2.B) EBX, unused
0xAAAAAAAA, // (2.C) EBP, unused
0x6D81BDD7, // (3)
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0x00002000, // (3.A) ECX, subtract from EDX to point to shellcode
0x6D802A88, // (4)
0x6DA612CC, // (4.A) EDI, address to save shellcode pointer
0x6D97ED06, // (5)
0xAAAAAAAA, // (5.A) EDI, unused
0xAAAAAAAA, // (5.B) ESI, unused
0xAAAAAAAA, // (5.C) EBP, unused
0x6D970A50, // (6)
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0x6D8011AC, // (7)
0x00000000, // (6.A) null to alloc anywhere
0x00004000, // (6.B) alloc_size
0xAAAAAAAA, // unused
0x6D824C7C, // (8)
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0x6D808150, // (9)
0x6D85A181, // (10)
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0x00003000, // memmove size (<= alloc_size - 1)
0xAAAAAAAA, // (10.A) unused
0x6D96FA23, // (11) exec shellcode
```

```
6D8011AC 83 C4 0C        add     esp,0Ch
6D8011AF C3              ret
```

```
6D8FF623 8B D6           mov     edx,esi
6D8FF625 5E              pop     esi
6D8FF626 8B C3           mov     eax,ebx
6D8FF628 5B              pop     ebx
6D8FF629 5D              pop     ebp
6D8FF62A C2 0C 00        ret     0Ch
```

```
6D81BDD7 59              pop     ecx
6D81BDD8 C3              ret
```

```
6D802A88 5F              pop     edi
6D802A89 C3              ret
```

```
6D97ED06 2B D1           sub     edx,ecx
6D97ED08 89 17           mov     dword ptr [edi],edx
6D97ED0A 5F              pop     edi
6D97ED0B 5E              pop     esi
6D97ED0C 5D              pop     ebp
6D97ED0D C2 0C 00        ret     0Ch
```

```
6D970A50 55              push    ebp
6D970A51 8B EC           mov     ebp,esp
6D970A53 8B 45 0C        mov     eax,dword ptr [ebp+0Ch]
6D970A56 85 C0           test    eax,eax
6D970A58 75 04           jne     6D970A5E
[illegible]              mov     al,1
[illegible]              [illegible]
6D970A5D C3              ret
[illegible]              [illegible]
6D970A60 68 00 10 00 00  push    1000h
[illegible]              push    eax
[illegible]              mov     eax,dword ptr [ebp+8]
6D970A69 50              push    eax
6D970A6A FF 15 84 F0 9E 6D  call  dword ptr ds:[6D9EF084h]
6D970A70 85 C0           test    eax,eax
6D970A72 0F 95 C0        setne   al
6D970A75 5D              pop     ebp
6D970A76 C3              ret
```

API call to VirtualAlloc() happens at 0x6D970A6A thus triggering EXEC flow simulation

ROP – Simulate Execution Flow

```
0x00000000, // null to avoid crashing
0xAAAAAAAA, // unused
0x6DA612CC, // writeable memory to avoid crashing
0x6D8FF623, // (2) return to register load
0xAAAAAAAA, // (2.A) ESI, unused
0xAAAAAAAA, // (2.B) EBX, unused
0xAAAAAAAA, // (2.C) EBP, unused
0x6D81BDD7, // (3)
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0x00002000, // (3.A) ECX, subtract from EDX to point to shellcode
0x6D802A88, // (4)
0x6DA612CC, // (4.A) EDI, address to save shellcode pointer
0x6D97ED06, // (5)
0xAAAAAAAA, // (5.A) EDI, unused
0xAAAAAAAA, // (5.B) ESI, unused
0xAAAAAAAA, // (5.C) EBP, unused
0x6D970A50, // (6)
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0x6D8011AC, // (7)
0x00000000, // (6.A) null to alloc anywhere
0x00004000, // (6.B) alloc_size
0xAAAAAAAA, // unused
0x6D824C7C, // (8)
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0x6D808150, // (9)
0x6D85A181, // (10)
0xAAAAAAAA, // unused
0xAAAAAAAA, // unused
0x00003000, // memmove size (<= alloc_size - 1)
0xAAAAAAAA, // (10.A) unused
0x6D96FA23, // (11) exec shellcode
```

```
6D8011A7 E8 04 FF FF FF    call   6D8010B0
6D8011AC 83 C4 0C          add    esp,0Ch
6D8011AF C3                ret
```

```
6D824C7C 8B C8             mov    ecx,eax
6D824C7E 8B 41 14          mov    eax,dword ptr [ecx+14h]
6D824C81 83 C4 04          add    esp,4
6D824C84 03 C1             add    eax,ecx
6D824C86 5D                pop    ebp
6D824C87 C3                ret
```

```
6D808150 A1 CC 12 A6 6D    mov    eax,dword ptr ds:[6DA612CCh]
6D808155 C2 08 00          ret    8
```

```
6D85A181 50                push   eax
6D85A182 51                push   ecx
6D85A183 FF 15 D4 F1 9E 6D call   dword ptr ds:[6D9EF1D4h]
6D85A189 83 C4 0C          add    esp,0Ch
6D85A18C 5D                pop    ebp
6D85A18D C3                ret
```

```
6D96FA23 FF E0             jmp    eax
```

ROP – Simulate Execution Flow

- The number of simulated instructions can be tweaked in the registry (**default value is 15**)
  - EMET_settings_\{app-guid}\SimExecFlowCount = REG_DWORD
- Example of instruction simulation code:

```
// Simulate a few instructions
switch ((DISX86::OPA)Instr.opa)
{
  // LEAVE
  case DISX86::opaLeave:
    // LEAVE = MOV ESP, EBP ; POP EBP
    simctrl->StackPtr = simctrl->FramePtr; // ESP = EBP
    // POP EBP
    simctrl->FramePtr = *(PDWORD)simctrl->StackPtr;
    simctrl->StackPtr += 4; // ESP += 4
    simctrl->last_push = false;
    break;
  //
  // MOV
  //
  case DISX86::opaMov:
    if (Opr[0].opcls == DIS::opclsRegister && Opr[0].rega1 == DISX86::regaEsp
        && Opr[1].opcls == DIS::opclsRegister && Opr[1].rega1 == DISX86::regaEbp)
    {
      // Set ESP = EBP
      simctrl->StackPtr = simctrl->FramePtr;
    }
    break;
  // RET
  case DISX86::opaRet:
    // Update code pointer
    simctrl->CodeAddress = (PBYTE)(*(PDWORD)simctrl->StackPtr);
    // Get the return operand
    simctrl->StackPtr   += 4 + (Instr.coperand == 0 ? 0 : (DWORD)Opr[0].dwl);
    return SIM_RET;
```

ROP – API special checks

- There are two checks under this mitigation
  - LoadLibrary checks
  - Stack area memory protection change check

ROP – API special checks

Load library checks

- Hooks APIs that loads libraries
  - LoadLibrary(), LoadLibraryEx(), ...
  - Disallow loading of libraries from UNC path
    - Some ROP gadgets try to load a remote DLL from a WebDav share
    - If the DLL loads, the attacker can execute code and elevate privilege
  - This mitigation won't flag if a DLL:
    - is loaded as resource
    - does not exist
  - This mitigation is not fool-proof
    - It works with EMET agnostic exploits

ROP – API special checks

Memory protection change

- This mitigation will trigger under the following situations:
  - A memory protection API is called
    - VirtualProtect, VirtualProtectEx, …
  - …and the target address belongs to the thread's stack area (defined in the TIB)

Mitigation Hardening

- EMET 4.0 introduces new protection against known bypasses
  - Down-level API hooking
  - Anti-Detours (explained before)
  - Banned APIs
- EMET 4.0 improved the speed for ROP checks

Mitigation Hardening

- Down-level API hooking
  - Not only kernel32!* critical functions are hooked
  - Now kernelbase!* and ntdll!* are hooked too

- For instance, kernel32!VirtualAlloc code path is:
  1. Kernel32!VirtualAlloc
  2. Kernelbase!VitualAlloc
  3. ntdll!NtAllocateVirtualMemory
- EMET will hook all three APIs **<u>but will only do the ROP checks once</u>** depending on the code path taken

Mitigation Hardening

- Banned API: EMET now has the ability to block certain APIs
- As of EMET 4.0, **ntdll!LdrHotPatchRoutine** is the only banned API
- When a banned API is called: the program will terminate

Mitigation Hardening

- Speed improvement:
  - Certain critical APIs will be quickly evaluated during runtime to see if they are really critical or not
  - Critical function no longer deemed critical will resume execution without spending time inside RopCheck()
  - For example, VirtualAlloc is not critical if the page protection parameter does not have the PAGE_EXECUTE* bit

Mitigation Hardening

```
// OPTIMIZATION:
// ---------------
// No need to do ROPChecks if a known "critical" function
// is called in a safe manner

FuncParamValidator_t FpV;
if (FpV.Parse(CalledAddress, pRSP))
{
  // Is this function used safely?
  if (FpV.IsSafe())
  {
    // Function parameters deemed safe, just skip the checks
    return ...;
  }
}
```

Certificate trust crypto extension

- The new certificate trust pinning feature is a two part implementation:
  - Native: implemented as a CryptoExtension*
  - Managed code: implemented as a subsystem hosted by “EMET Agent”
- The crypto extension will collect the certificates in question from the context of the caller process (example: Internet Explorer) and send them via IPC to “EMET Agent”

* http://msdn.microsoft.com/en-us/library/windows/desktop/aa382405(v=vs.85).aspx

Certificate trust crypto extension

- The rule validation algorithm and description is found in EMET's User Guide
- http://blogs.technet.com/b/srd/archive/2013/05/08/emet-4-0-s-certificate-trust-feature.aspx
- http://blogs.technet.com/b/srd/archive/2013/04/18/introducing-emet-v4-beta.aspx

EMET UI

- EMET UI is composed of two tools (managed code):
  - Graphical user interface (EMET_GUI)
  - Text user interface (EMET_conf)

- The UI must run elevated
  - It re-writes the SDB file to include the new programs to be protected by EMET
  - Manages EMET configuration
    - General settings
    - Cert trust settings
    - Etc…

Questions?

Download EMET from:

http://www.microsoft.com/emet

http://aka.ms/emet/

Please send comments to:

emet_feedback@microsoft.com

Microsoft