Exploiting Windows Kernel Vulnerabilities in Hard Conditions

Nikita Tarakanov

POC, Seoul November 2013

Agenda

- Whoami.exe
- Introduction
- Vulnerability classes overview
- Current exploitation techniques
- Problem
- New ideas
- Conclusion
- Q&A

Whoami.exe

- Security Researcher from Russian Federation
- Like developing exploits and exploitation techniques
- Like challenges (PHDays hack2own winner)
- Like facing/solving problems: automation of vulnerabilities discovery, exploiting unexploitable vulns etc…

Introduction

- Windows kernel vulnerabilities are gaining more and more attention
- Good research in this area these days
- MS enhances security by implementing various exploit mitigations and kills exploit techniques and even vulnerability classes (null pointer dereference etc.)

Windows Kernel Vulnerabilities what for???

- EoP and following attacks: hypervisor, UEFI,SMM, and popular in cybercriminal's world: rootkit/bootkit
- Sandbox escape – great example is win32k.sys pwn and escape from Google Chrome sandbox by Nils and John from MWRLabs at pwn2own 2013

Vulnerability classes overview

- Kernel land “binary” world: null pointer dereferences , memory corruptions, use after frees, race conditions etc…
- Plus a lot of architecture, logic vulnerabilities…

Windows Kernel Vulnerabilities

- Null Pointer Dereference
- Arbitrary Memory Overwrite/Read
- Pool Overflow/Corruption
- Race Condition (infamous double fetch)
- Stack overflow
- Use-after-free
- Etc…

Null Pointer Dereference

- Common vulnerability class in MS kernel components and in 3rd party drivers
- Execute/Write/Read at NULL or NULL + offset
- Very easy to exploit
- Windows 8 killed this class -> BSOD only ☹
- MS13-031 killed this class on x64 ☹ ☹ ☹

CONST Pointer Dereference

- Execute/Write/Read at CONST, where CONST is some r3/r0 address
- Easy to exploit if address is r3 address
- Tricky to exploit if address is r0 address
- SMEP (Windows 8) killed whole class (exec r3 address) ☹

Memory Corruption

- Arbitrary address overwrite
- Stack-based buffer corruption/overflow
- Pool corruption/overflow

Stack-based buffer overflow

- Pretty rare these days
- Stack cookie ☹
- Need control over data: cookie + EIP/RIP ☹
- Easy/Tricky to exploit

Arbitrary address overwrite

- Common vulnerability class in MS kernel components and in 3rd party drivers
- Exploitation techniques are well known: infamous HalDispatchTable, Kernel Objects (j00ru's research), various pointers (Alex's Ionescu research)
- ***Easy*** to exploit

Pool corruption/overflow

- Common vulnerability class in MS kernel components and in 3rd party drivers
- Exploitation techniques are well known: pool metadata, object manager metadata
- Easy/Tricky to exploit

Current exploitation techniques

- Rely on full/partial control of values, so have preconditions
- Pointer overwrite – valid pointer
- Pool overflow – valid values for pool header, object header etc…

Current exploitation techniques

- Preconditions can turn ***easy to exploit*** vulnerability to **unexploitable** one
- Some techniques don't work on win8 or after ms13-031 (Null Page allocation restriction)

Current exploitation techniques Preconditions examples

- Various pointers overwrite – valid pointer
- Pool metadata – valid POOL_HEADER
- Object metadata – valid OBJECT_HEADER
- PoolIndex out-of-bound – Null Page allocation
- TypeIndex 0x00 value – Null Page allocation

Problem

- What if there is arbitrary pointer overwrite vulnerability, but we don’t fully(on don’t control at all) control the value that overwrites the memory?
- For example if value is address of some pool memory and is not predictable
- Example ms13-053

New ideas

- What about overwriting some variable that influences the execution flow indirectly?

Idea #1 KiServiceLimit

- KiServiceLimit contains number of system calls

```
000000014007E0F2 KiSystemServiceRepeat proc near         ; CODE XREF: KiSystemServiceExit+1E0↓j
000000014007E0F2                 lea     r10, KeServiceDescriptorTable
000000014007E0F9                 lea     r11, KeServiceDescriptorTableShadow
000000014007E100                 test    dword ptr [rbx+100h], 80h ; check GUI flag
000000014007E10A                 cmovnz  r10, r11
000000014007E10E                 cmp     eax, [rdi+r10+10h] ; eax - holds system number to invoke
000000014007E10E                                            ; checks against KiServiceLimit
000000014007E113                 jnb     loc_14007E402    ; quit if bigger
000000014007E119                 mov     r10, [rdi+r10]
000000014007E11D                 movsxd  r11, dword ptr [r10+rax*4]
000000014007E121                 mov     rax, r11
000000014007E124                 sar     r11, 4
000000014007E128                 add     r10, r11         ; r10 contains address of system service
```

- What if smash this value with bigger one?

Idea #1 x64 calling convention

- Parameters passing

Parameter type	How passed
Floating point	First 4 parameters – XMM0 through XMM3. Others passed on stack.
Integer	First 4 parameters – RCX, RDX, R8, R9. Others passed on stack.
Aggregates (8, 16, 32, or 64 bits) and __m64	First 4 parameters – RCX, RDX, R8, R9. Others passed on stack.
Aggregates (other)	By pointer. First 4 parameters passed as pointers in RCX, RDX, R8, and R9
__m128	By pointer. First 4 parameters passed as pointers in RCX, RDX, R8, and R9

Idea #1 Execution hijack

- Stack desynchronisation -> EIP hijack
- JMP reg/[reg], call reg/[reg] etc

Idea #1 Preconditions

- Value has to be bigger than KiServiceLimit
- Overwrite has to be “clean”

Idea #2 DKOHM

- kd> dt nt!_OBJECT_HEADER
- +0x000 PointerCount : Int4B
- +0x004 HandleCount : Int4B
- +0x004 NextToFree : Ptr32 Void
- +0x008 Lock : _EX_PUSH_LOCK
- **+0x00c TypeIndex : UChar**
- +0x00d TraceFlags : UChar
- +0x00d DbgRefTrace : Pos 0, 1 Bit
- +0x00d DbgTracePermanent : Pos 1, 1 Bit
- +0x00e InfoMask : UChar
- +0x00f Flags : UChar
- +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
- +0x010 QuotaBlockCharged : Ptr32 Void
- +0x014 SecurityDescriptor : Ptr32 Void
- +0x018 Body : _QUAD

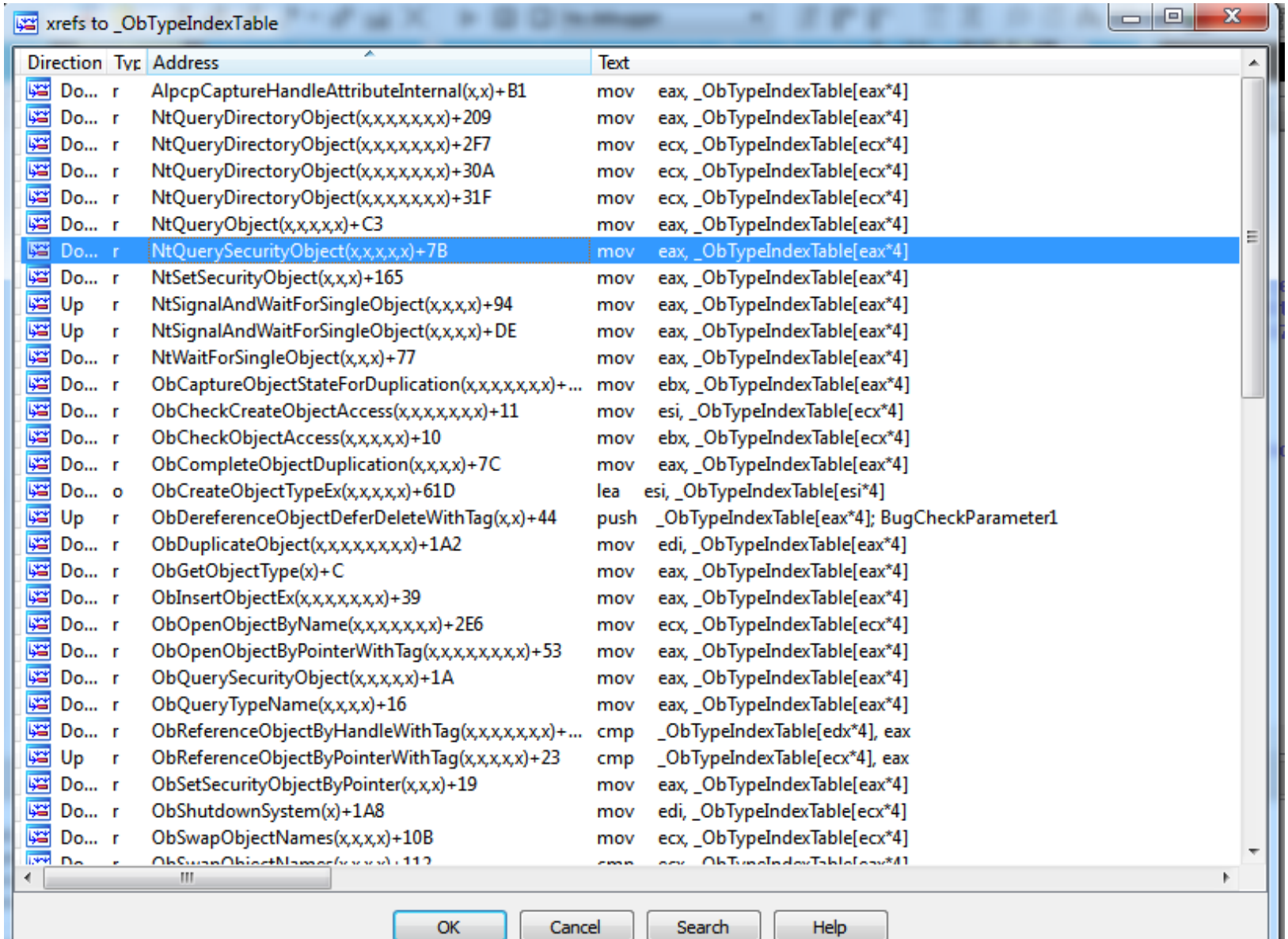
Idea #2 ObTypeIndexTable

- kd> dd nt!ObTypeIndexTable L40
- 81a3edc0 00000000 bad0b0b0 8499c040 849aa390
- 81a3edd0 84964f70 8499b4c0 84979500 84999618
- 81a3ede0 84974868 849783c8 8499bf70 84970b40
- 81a3edf0 849a8888 84979340 849aaf70 849a6a38
- 81a3ee00 8496df70 8495b040 8498cf70 84930a50
- 81a3ee10 8495af70 8497ff70 84985040 84999e78
- 81a3ee20 84997f70 8496c040 849646e0 84978f70
- 81a3ee30 8497aec0 84972608 849a0040 849a9750
- 81a3ee40 849586d8 84984f70 8499d578 849ab040
- 81a3ee50 84958938 84974a58 84967168 84967098
- 81a3ee60 8496ddd0 849a5140 8497ce40 849aa138
- 81a3ee70 84a6c058 84969c58 8497e720 85c62a28
- 81a3ee80 85c625f0 00000000 00000000 00000000

Idea #2 Execution hijack

- We can make fake entry in ObTypeIndexTable
- Next corrupt TypeIndex field in Object to point to the fake entry
- Gaining full control of execution flow

Idea #2 xrefs


xrefs to _ObTypeIndexTable
Direction Typ Address Text
Do... r AlpcpCaptureHandleAttributeInternal(x,x)+B1 mov eax, _ObTypeIndexTable[eax*4]
Do... r NtQueryDirectoryObject(x,x,x,x,x,x,x,x)+209 mov eax, _ObTypeIndexTable[eax*4]
Do... r NtQueryDirectoryObject(x,x,x,x,x,x,x,x)+2F7 mov ecx, _ObTypeIndexTable[ecx*4]
Do... r NtQueryDirectoryObject(x,x,x,x,x,x,x,x)+30A mov ecx, _ObTypeIndexTable[ecx*4]
Do... r NtQueryDirectoryObject(x,x,x,x,x,x,x,x)+31F mov ecx, _ObTypeIndexTable[ecx*4]
Do... r NtQueryObject(x,x,x,x,x)+C3 mov eax, _ObTypeIndexTable[eax*4]
Do... r NtQuerySecurityObject(x,x,x,x,x)+7B mov eax, _ObTypeIndexTable[eax*4]
Do... r NtSetSecurityObject(x,x,x)+165 mov eax, _ObTypeIndexTable[eax*4]
Up r NtSignalAndWaitForSingleObject(x,x,x,x)+94 mov eax, _ObTypeIndexTable[eax*4]
Up r NtSignalAndWaitForSingleObject(x,x,x,x)+DE mov eax, _ObTypeIndexTable[eax*4]
Do... r NtWaitForSingleObject(x,x,x)+77 mov eax, _ObTypeIndexTable[eax*4]
Do... r ObCaptureObjectStateForDuplication(x,x,x,x,x,x,x,x)+... mov ebx, _ObTypeIndexTable[eax*4]
Do... r ObCheckCreateObjectAccess(x,x,x,x,x,x,x,x)+11 mov esi, _ObTypeIndexTable[ecx*4]
Do... r ObCheckObjectAccess(x,x,x,x,x)+10 mov ebx, _ObTypeIndexTable[ecx*4]
Do... r ObCompleteObjectDuplication(x,x,x,x)+7C mov eax, _ObTypeIndexTable[eax*4]
Do... o ObCreateObjectTypeEx(x,x,x,x,x)+61D lea esi, _ObTypeIndexTable[esi*4]
Up r ObDereferenceObjectDeferDeleteWithTag(x,x)+44 push _ObTypeIndexTable[eax*4]; BugCheckParameter1
Do... r ObDuplicateObject(x,x,x,x,x,x,x,x,x)+1A2 mov edi, _ObTypeIndexTable[eax*4]
Do... r ObGetObjectType(x)+C mov eax, _ObTypeIndexTable[eax*4]
Do... r ObInsertObjectEx(x,x,x,x,x,x,x,x)+39 mov eax, _ObTypeIndexTable[eax*4]
Do... r ObOpenObjectByName(x,x,x,x,x,x,x,x)+2E6 mov ecx, _ObTypeIndexTable[ecx*4]
Do... r ObOpenObjectByPointerWithTag(x,x,x,x,x,x,x,x,x)+53 mov eax, _ObTypeIndexTable[eax*4]
Do... r ObQuerySecurityObject(x,x,x,x,x)+1A mov eax, _ObTypeIndexTable[eax*4]
Do... r ObQueryTypeName(x,x,x,x)+16 mov eax, _ObTypeIndexTable[eax*4]
Do... r ObReferenceObjectByHandleWithTag(x,x,x,x,x,x,x,x)+... cmp _ObTypeIndexTable[edx*4], eax
Up r ObReferenceObjectByPointerWithTag(x,x,x,x,x)+23 cmp _ObTypeIndexTable[ecx*4], eax
Do... r ObSetSecurityObjectByPointer(x,x,x)+19 mov eax, _ObTypeIndexTable[eax*4]
Do... r ObShutdownSystem(x)+1A8 mov edi, _ObTypeIndexTable[ecx*4]
Do... r ObSwapObjectNames(x,x,x,x)+10B mov ecx, _ObTypeIndexTable[ecx*4]
OK
Cancel
Search
Help

Idea #2 Preconditions

- Value should be in range of non-existed object types (highest_index < N <= 0xff)
- By partially overwrite – allocable address must be constructed

Conclusion

- MS should disable/abandon API that allows various information leakage of kernel memory layout
- MS should defend (somehow) such critical variables like KiServiceLimit
- MS should probably implement control flow integrity in kernel land

Q&A

- @NTarakanov

References