Vulnerability Research in Windows Kernel Drivers

From Fuzzing to Exploitation

Jérémy Brun

Overview

1. A bit of theory about Windows Drivers
2. Fuzzing methods
   – Mutation-based
   – Generation-based
3. Vulnerability analysis
   – Crash dump analysis
   – Reverse engineering
4. Privilege Escalation Exploit

1

A bit of theory about Drivers

Introduction

- Software exploitation is becoming more and more difficult …

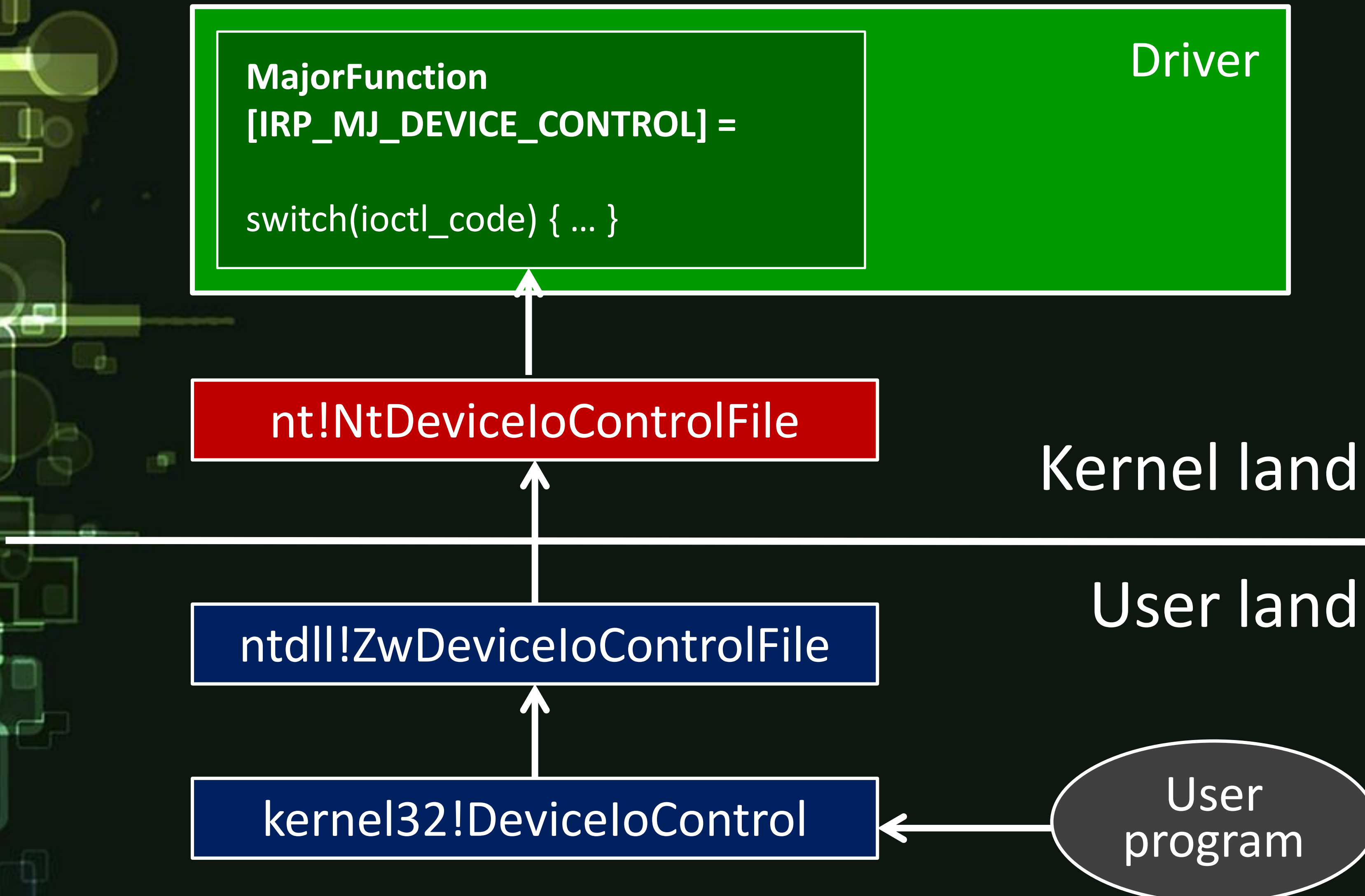
- Many mitigation mechanisms in userland: ASLR, DEP, SEHOP, \GS …

=> More and more interest for kernel vulnerabilities !

Target = Third-party drivers

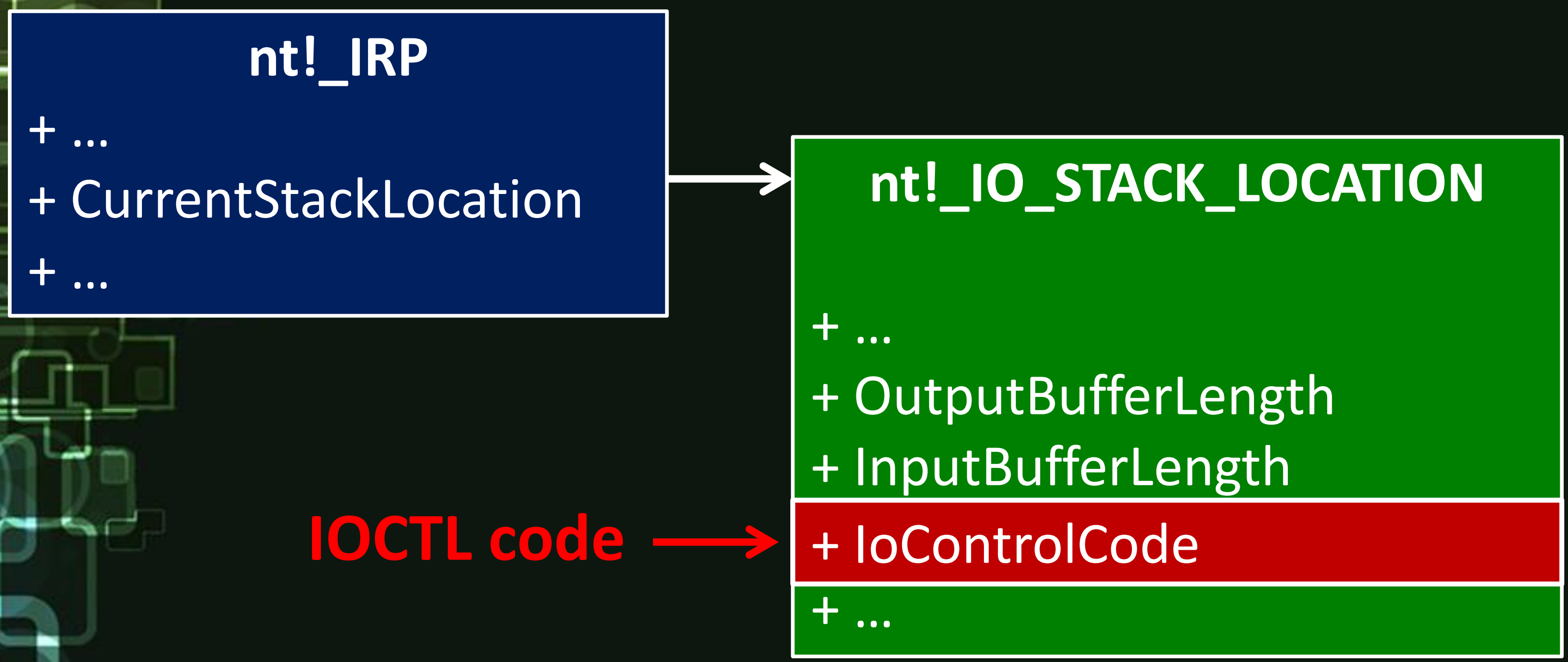
- Kernel driver = Software running with ring0 privileges
- Antivirus, Anti-rootkits… install drivers because some features need kernel access
- Example: Checking the integrity of SSDT

Communication with Drivers



I/O Request Packets (IRPs)

- Communication is performed by I/O Manager using IRPs:



I/O Control Codes (IOCTLs)

- IOCTL code <-> functionality provided by the driver

31	30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14	13	12 11 10 9 8 7 6 5 4 3 2	1 0
Common	Device Type	Required Access	Custom	Function Code	Transfer Type

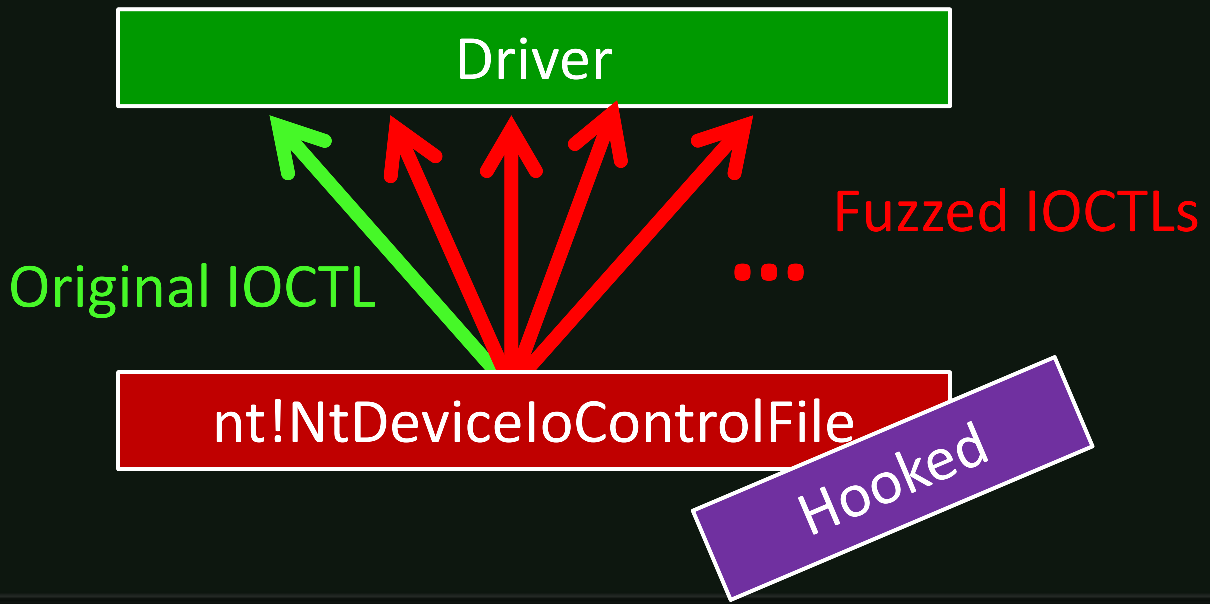
- **For 1 driver = only Function code + Transfer type can change**

2

Fuzzing Methods

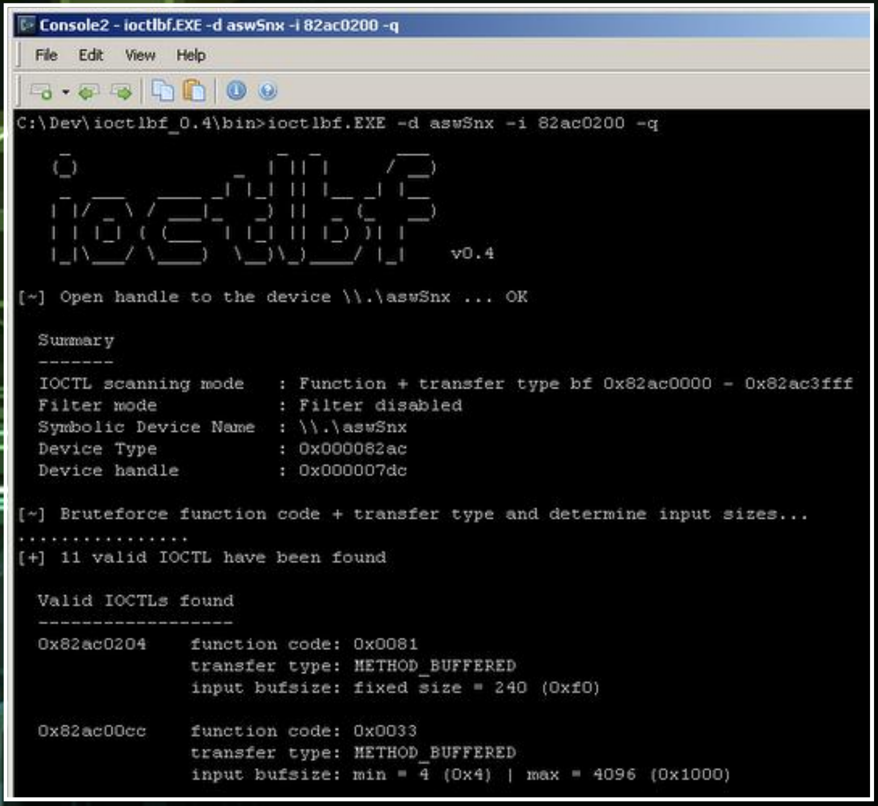
Mutation-based IOCTL Fuzzing

- Best tool: “ioctlfuzzer” (http://code.google.com/p/ioctlfuzzer)
- Capture valid IOCTL buffers, and add anomalies in it.



Generation-based IOCTL Fuzzing

- Homemade tool: “ioctlbf” (http://code.google.com/p/ioctlbf)



1. Scan for valid IOCTL codes,
2. Determine supported buffer sizes,
3. Fuzz chosen IOCTL in various mode

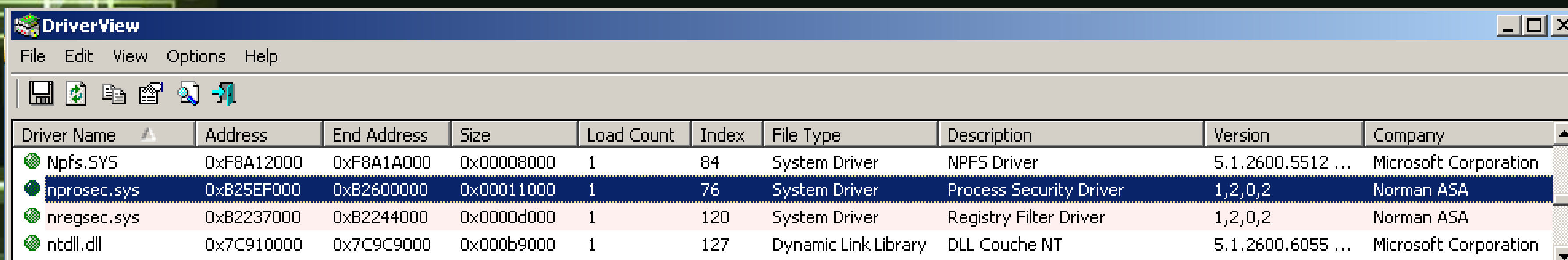
Pros/Cons

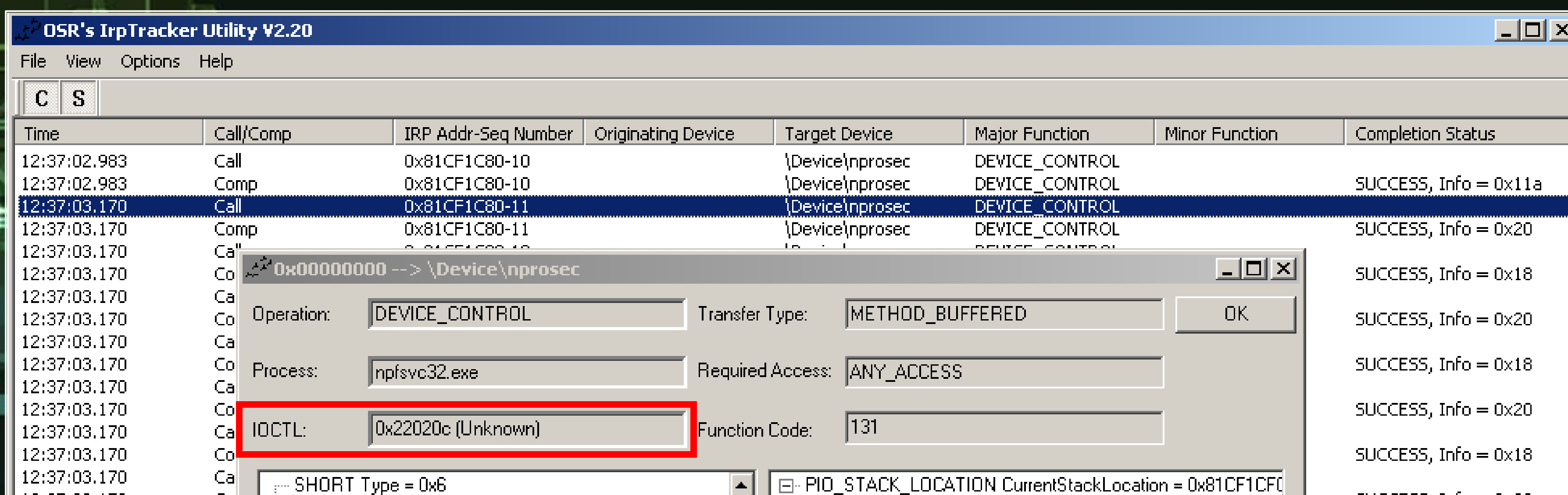
Mutation-based	Generation-based
- Good code coverage	- Able to fuzz IOCTLs not often/never used by applications (debug…)
- Depends on the activity - Some IOCTLs may be missed	- Usually, takes more time to find bugs

=> Both methods are complementary

A case study: Norman Security Suite 8

- Target driver = nprosec.sys





Let’s FuZz it !

3

Vulnerability Analysis

Crash Dump Analysis (1/2)

- kd> !analyze –v

```
PAGE_FAULT_IN_NONPAGED_AREA (50)
WRITE_ADDRESS:  fffffffe

eax=00000010 ebx=81ab6000 ecx=0012d6a6 edx=00000002 esi=81ab6000 edi=fffffffe
eip=80536a20 esp=b1246bb8 ebp=b1246bc0 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010202

nt!memcpy+0xa0:
80536a20 8807            mov     byte ptr [edi],al         ds:0023:fffffffe=??

STACK_TEXT:
[…]
b1246d34 8053d6d8 000007e8 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
b1246d34 7c91e514 000007e8 00000000 00000000 nt!KiFastCallEntry+0xf8
001076d8 7c91d28a 7c801675 000007e8 00000000 ntdll!KiFastSystemCallRet
001076dc 7c801675 000007e8 00000000 00000000 ntdll!ZwDeviceIoControlFile+0xc
0010773c 004168df 000007e8 00220210 001177b8 kernel32!DeviceIoControl+0xdd
[…]
```

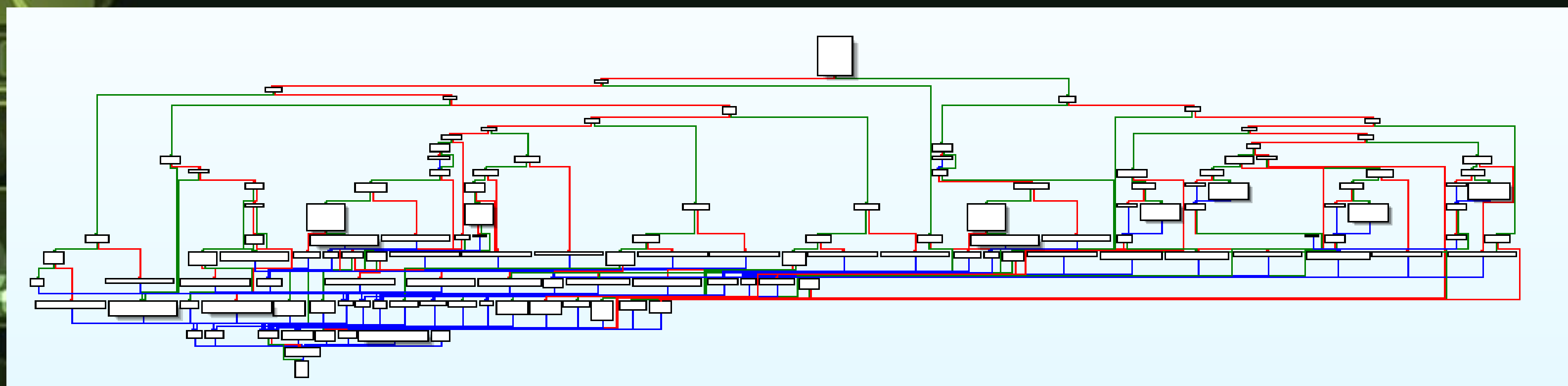
Crash Dump Analysis (2/2)

- What was the buffer which caused the crash ?

```
kd> dd b1246d34        @ call nt!ntDeviceIoControlFile

b1246d34  b1246d64 8053d6d8 000007e8 00000000
b1246d44  00000000 00000000 00107718 00220210   IOCTL
b1246d54  001177b8 00000008 001077b8 00000008

          @inBuffer size(inBuffer)

kd> dd 001177b8

001177b8  fffffffe 0012d6a8 00000000 00000000

          inBuffer content
```

Reverse Engineering (1/2)

- **Step #1:** Find the IOCTL dispatch function



- **Step #2:** Find the vulnerable IOCTL handler

```
v2 = Irp;
v3 = Irp->Tail.Overlay.CurrentStackLocation;
v4 = 0;
v52 = 0;
ioctl = *((_DWORD *)v3 + 3);
if ( ioctl <= 0x220250 )
{
  if ( ioctl != 0x220250 )
  {
    v6 = ioctl - 0x220004;
    if ( v6 )
    {
      v7 = v6 - 0x200;
      if ( v7 )
      {
        v8 = v7 - 4;
        if ( v8 )
        {
```

Reverse Engineering (2/2)

- **Step #3:** Locate the vulnerable function

```
v52 = buffer_off4 >> 3;
v4 = sub_12756(*(void **)&buffer->off0, (int)&v52, 0);
if ( v4 == 0xC0000023 )
  v4 = 0;
Irp->IoStatus.Status = v4;
```

```
signed int __stdcall sub_12756(void *a1, int a2, char a3)
{
  PVOID v3; // ebx@1
  signed int result; // eax@2
  int i; // edi@3
  int v6; // ecx@4
  unsigned int v7; // [sp+1Ch] [bp-20h]@1
  KIRQL NewIrql; // [sp+23h] [bp-19h]@3

  v7 = 0;
  v3 = ExAllocatePoolWithTag(0, 8 * *(_DWORD *)a2, 0x72667542u);
  if ( v3 )

 [...]
    KfReleaseSpinLock(&SpinLock, NewIrql);
    memcpy(a1, v3, 8 * *(_DWORD *)a2);
    *(_DWORD *)a2 = v7;
```

Kernel Pointer Dereference

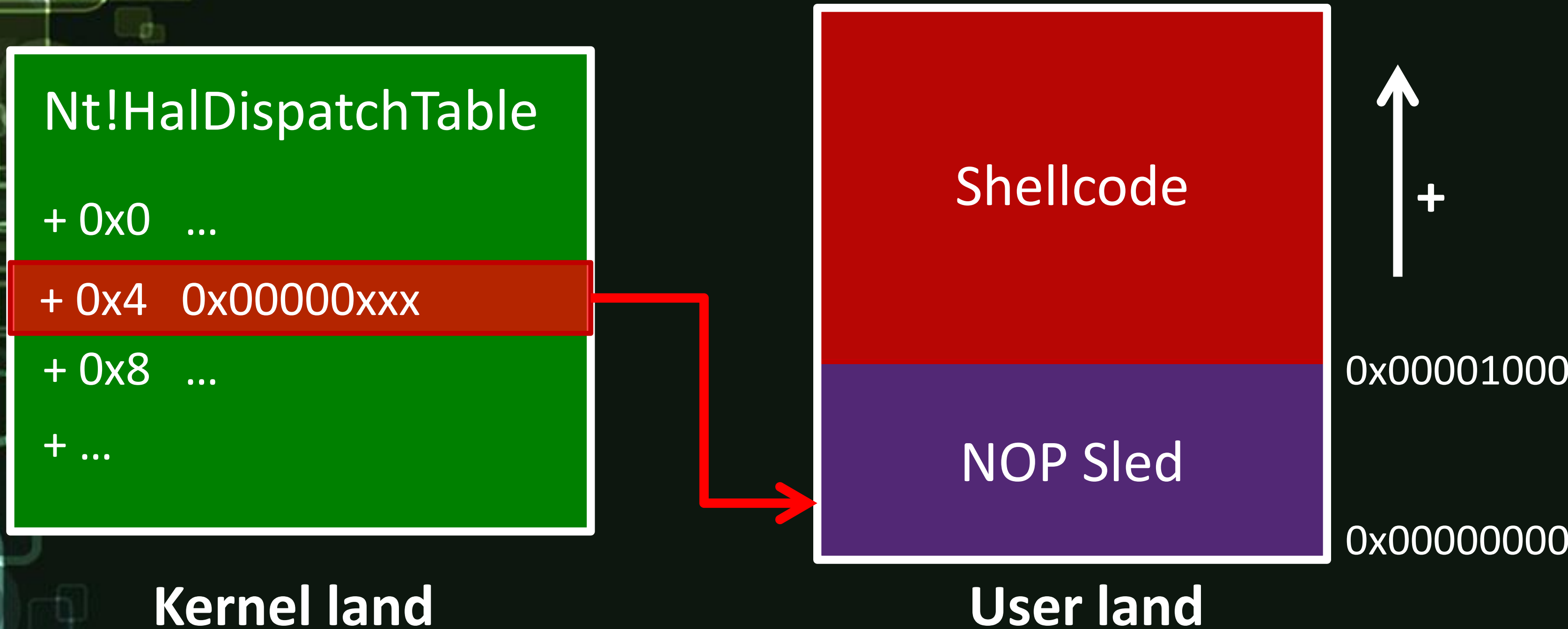
- **"Write-What-Where vulnerability":**
  - **What** = 2 DWORDs (min size) not controlled, but 1st one always below 0x00000FFF
  - **Where** = fully controlled address, 1st DWORD of the input buffer
- **It's enough for exploitation !**

4

Privilege Escalation Exploit

Exploitation Method

- **What =** 0x00000xxx (address in userland)
- **Where =** Address of a pointer in a kernel dispatch table (*nt!HalDispatchTable*)



Exploit development (1/2)

1. **Shellcode** = Steal "Access Token" (Security context) of the "System" process

   – Replace the address of the Token in nt!_EPROCESS corresponding to exploit's process.
   – Go back to Ring3

2. **Send IOCTL** 0x00220210 with buffer:

@nt!HalDispatchTable+4	0x00000008

Exploit development (2/2)

3. **Call *NtQueryIntervalProfile* API**

=> Call [nt!HalDispatchTable+4] in ring0

=> Redirect execution flow to NOP sled

=> Shellcode is finally executed !

GAME OVER

Demo !

Sploit available at

http://www.exploit-db.com/exploits/17902/

Questions ? …