Windows Local Kernel Exploitation

X’CON 2004 Beijing

sk@scan-associates.net
Co-founder, Security Consultant, Software Architect
Scan Associates Sdn Bhd

X'CON 2004

scan associates

Overview

- Windows Privilege Escalations
- Windows Kernel 101
- Device driver communication problem
  - DeviceIOControl
  - Finding
  - Exploiting
- Kernel shellcode
- Locating base address of device
  - Undocumented API (NtQuerySystemInformation)
- Demo

Windows Privilege Escalation

- Exploiting SYSTEM privilege application:
  - Buffer overflow in Still Image Service
  - IIS IDQ.DLL
  - Buffer overflow in POSIX subsystem
- LPC problems
- Named pipe impersonation
- Shatter attack
- Kernel bugs

LPC problems

- Local Procedure Call allows processes to communicate
- Various problems discovered by Todd Sabin
- NtImpersonateClientOfPort()
  - http://www.bindview.com/Support/RAZOR/Advisories/2000/adv_NTPromotion.cfm
  - http://www.bindview.com/Support/RAZOR/Advisories/2000/LPCAdvisory.cfm
- Signedness problem in NTLM Security Support Provider (NTLMSSP) LPC port
  - http://www.bindview.com/Support/RAZOR/Advisories/2001/adv_NTLMSSP.cfm

Named Pipe Impersonation

- A server named pipe can impersonate its client
- Attacker create named pipe before the server create it
- A privileged client connect to our server named pipe, we can impersonate the client to get its privilege
- http://www.blakewatts.com/namedpipepaper.html

IDQ.DLL

- IIS always load certain file with specific filename using SYSTEM privilege
- By creating filename such as:
  - IDQ.DLL
  - httpext.dll
  - httpodbc.dll, etc
- http://www.xfocus.org/exploits/200110/7.html

Shatter Attack

- Send Windows Message to any process
- Basic Shatter:
    - Locate a privileged Windows
    - Send shellcode to target process space
    - Send WM_TIMER message to jump to shellcode in its own space
- Advance Shatter is still just Shatter
- Require Desktop
- Also known as *Local* Local attack
- Limited use

Kernel Bugs

- Problems that exist in Kernel land
- Will give us highest access, same level as the OS
- Windows Kernel is not a well documented
- Generally more complex than user land
- Probably still plenty of ‘fish’
- Kernel bugs is gaining popular **J**

Known Kernel Bugs

- Buffer Overrun in Windows Kernel Message Handling
  - http://www.microsoft.com/technet/security/bulletin/MS03-013.mspx
- Windows VDM TIB
  - http://www.eeye.com/html/research/advisories/AD20040413E.html
- Windows Expand-Down Data Segment
  - http://www.eeye.com/html/research/advisories/AD20040413D.html
- Device Driver Communication Problem
  - http://sec-labs.hack.pl/papers/win32ddc.php

Windows Kernel 101

User Land	Kernel Land
Ring 3	Ring 0
Each process has 2GB memory	Every kernel modules, device driver share the same 2GB memory
Memory address from 0x00000000 to 0x7FFFFFFF	Memory address from 0x80000000 to 0xFFFFFFFF
Sandbox!	Freedom!

…Windows Kernel 101

- Windows kernel land consists of:
  - Kernel
  - Executives
    - Process and Thread manager, I/O Manager, etc
  - Win32 User GDI
  - Device Driver
- The kernel contains many important executives object which control the application in user land

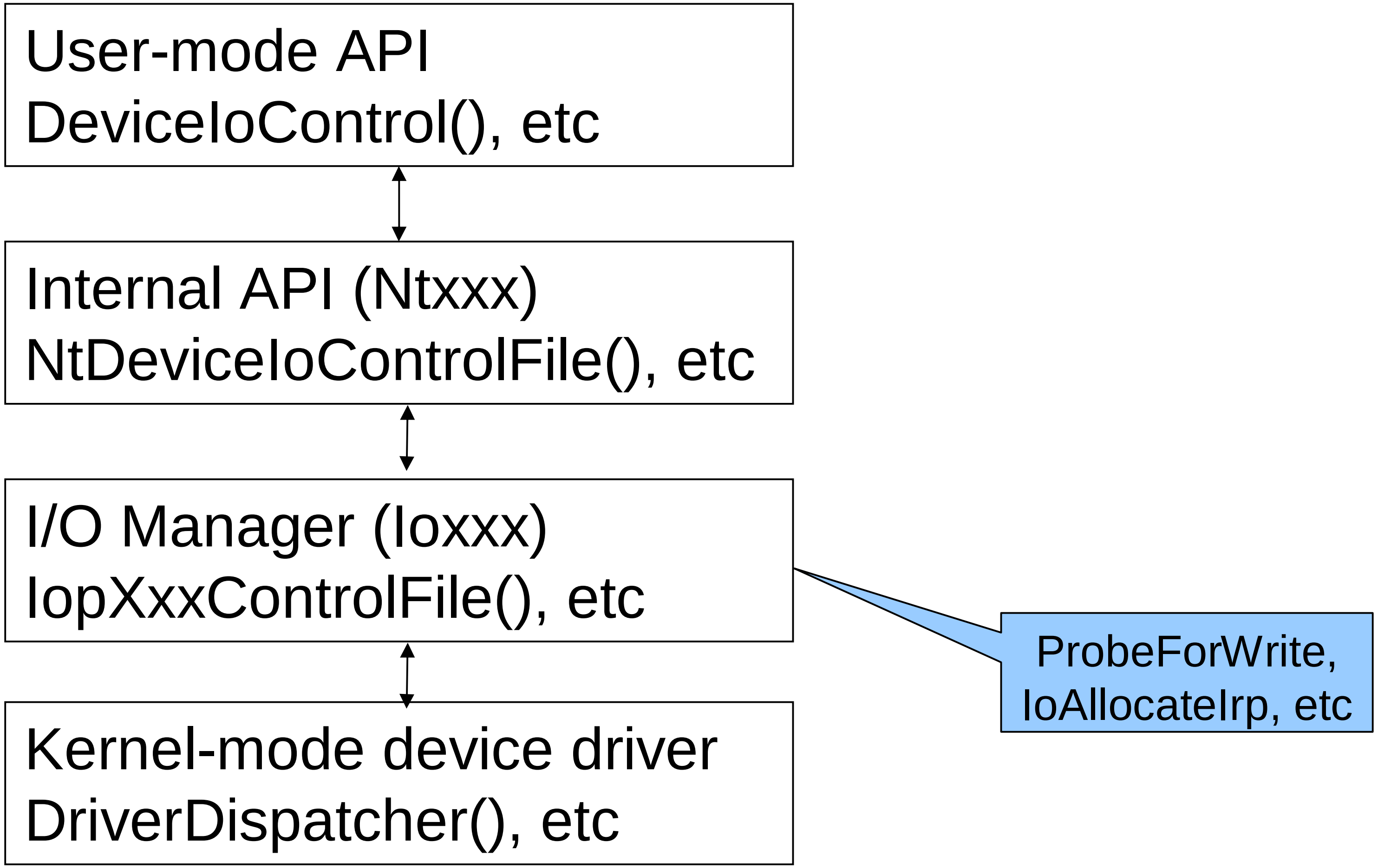
Device Driver

- Loadable Kernel Module (LKM)
- Once in kernel, Device Driver is trusted
- Ability to modify kernel object to change behavior of application in user land
- Application such as Personal firewall, Antivirus, etc sometimes install Device Driver to change behavior of user land:
  - Check all socket connections
  - Check all file access, etc

Device Driver Communication

- Device driver can accept data from user land via:
  - ReadFile / WriteFile()
  - DeviceIoControl()
- Before it can be used, we must open the driver:
  - CreateFile()
- We can access device driver much like a file

Data flow



User-mode API
DeviceIoControl(), etc

Internal API (Ntxxx)
NtDeviceIoControlFile(), etc

I/O Manager (Ioxxx)
IopXxxControlFile(), etc

Kernel-mode device driver
DriverDispatcher(), etc

ProbeForWrite,
IoAllocateIrp, etc

Device Driver Skeleton

- Basic device driver
  - **DriverEntry**()
  - **DriverDispatcher**()
  - **DriverUnload**()
- Data from **DeviceIoControl**() will be process in **DriverDispatcher**()

DeviceIoControl()

- Communication between user land and kernel land
- User program send control code to device driver via **DeviceIoControl**() API
- Device driver receive control code and process
- Device driver return output to user land via output pointer specified by caller

DeviceIoControl

```
BOOL DeviceIoControl(
      HANDLE hDevice, // handle to device
      DWORD dwIoControlCode, // operation
      LPVOID lpInBuffer, // input data buffer
      DWORD nInBufferSize, // size of input data
                                         //buffer
      LPVOID lpOutBuffer, // output data buffer
      DWORD nOutBufferSize, // size of output
                                         //data buffer
      LPDWORD lpBytesReturned, // byte count
      LPOVERLAPPED lpOverlapped  //overlapped
                                              //information
);
```

lpOutBuffer

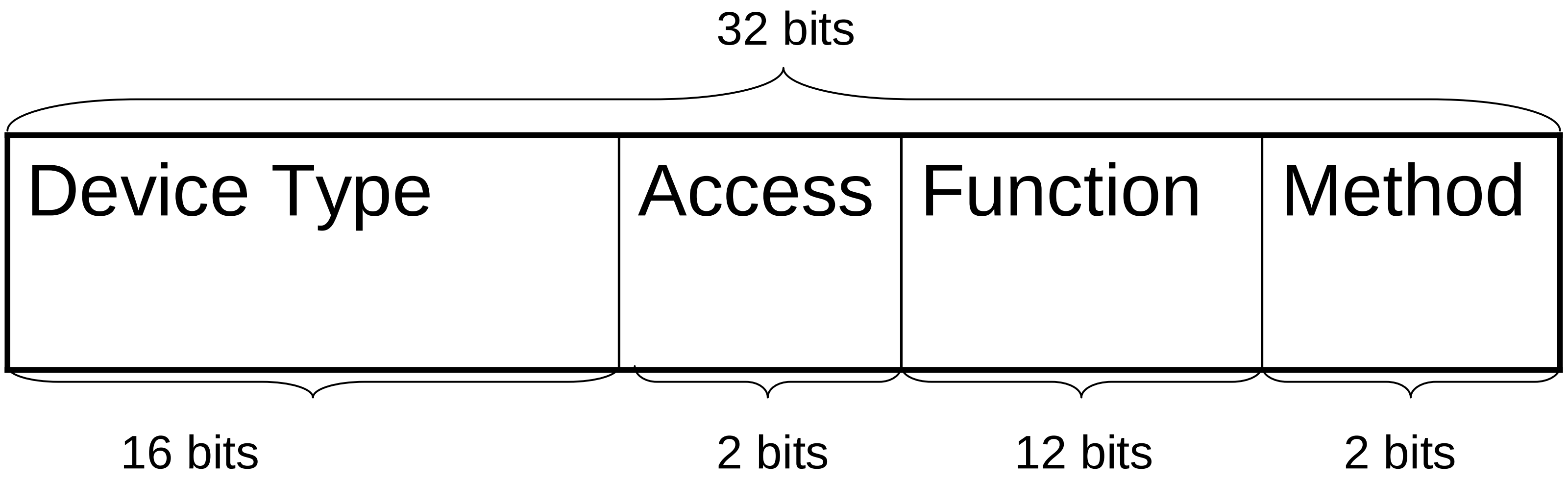
- What if output buffer is a memory address in kernel?
- Will we be able to overwrite any kernel address?
- What if we point it to overwrite important token?
- What if we overwrite function pointer?
- (Un)Fortunately, I/O Manager provides buffer handling for device driver

Type of buffer management

- Buffered I/O (Method 0)
  - I/O manager allocates enough buffer copy from/to sender's data
- Direct I/O (Method 1 and 2)
  - Sender's buffer is lock and I/O manager pass the pointer of the memory to driver
- Neither I/O (Method 3)
  - No buffer management

CTL_CODE

- #define CTL_CODE(DeviceType, Function, Method, Access) (((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method);



32 bits

Device Type	Access	Function	Method
16 bits	2 bits	12 bits	2 bits

Neither I/O

- Device I/O Control Code that ends with 011b
  - 0xXXXXXXX3
  - 0xXXXXXXX7
  - 0xXXXXXXXB
  - 0xXXXXXXXF
- Output pointer can be anywhere, including kernel land
- Arbitrary memory write

Finding Neither I/O

- Source code and Header file
- Application hooking
  - **strace –p PID**
- Hook system wide ***DeviceIoControl***
  - From the book, “Undocumented Windows 2000 Secrets”
  - **C:\w2k_hook *DeviceIoControl***

Find Neither I/O by Source

- Bug found by mslug (https://www.xfocus.net/bbs/index.php?act=SE&f=16&t=32580&p=115340&hl=)
  - #define BIOCGSTATS 9031 //0x2347
- Other potential targets in Packet.h:
  - #define BIOCISDUMPENDED 7411 //0x1CF**3**
  - #define BIOCSRTIMEOUT 7416 //0x1CF8
  - #define BIOCSMODE 7412 //0x1CF4
  - #define BIOCSWRITEREP 7413 //0x1CF5
  - #define BIOCSMINTOCOPY 7414 //0x1CF6
  - #define BIOCGEVNAME 7415 //0x1CF**7**
  - #define BIOCSENDPACKETSSYNC 9033 //0x2349
  - #define BIOCSETDUMPLIMITS 9034 //0x234A

Find Neither I/O via System Hook

- `C:\w2k_hook *DeviceIoControl*`
  - `1CF:s0=NtDeviceIoControlFile(!2B8.3B4="\??\NAVAP",p,p,p,i0.4,n222A87,p3CFFEF8,n20,p3CFFEF0,n4)1C4963F2B6F71D0,530,3`
  - `18D:s0=NtDeviceIoControlFile(!5C8.344="\Device\Tcp",p330,p,p,i0.38,n120003,p6F4D8,n24,pB01E90,n8000)1C494FBFF5C1960,42C,A`
  - `606:s0=NtDeviceIoControlFile(!E4.898="\Device\Afd\Endpoint",p1E4,p,p,i0.0,n12047,p1A2F6F0,nD4,p,n0)1C495035A74B1E0,648,1D`
  - `1:s0=NtDeviceIoControlFile(!354.120="\??\shadow",p,p,p,i0.0,n140FFB,p6B2F8,n0,n0)1C495C2244759C0,634,27`
  - `3201:s0=NtDeviceIoControlFile(!1F0.2D8="\Device\LanmanDatagramReceiver",p2D0,p,p,i0.50,n130023,pD5FD24,n50,pA4FF8,n1000)1C4964E8570CB16,584,47`

Exploiting DDCV

- Norton A/V Enterprise
- Contains NAVAP.sys device driver
- Allows communication from user program via **DeviceIoControl**()
- The following CTL_CODE supported:
  - PAGE:0001649D cmp ecx, 222A83h
  - PAGE:000164A5 cmp ecx, 222A87h
  - PAGE:000164AD cmp ecx, 222A8Bh
  - PAGE:000164B5 cmp ecx, 222A8Fh
  - PAGE:000164BD cmp ecx, 222A93h
  - PAGE:000164C5 cmp ecx, 222A97h
  - PAGE:000164CD cmp ecx, 222A9Bh
- Uses **Neither I/O** heavily (for performance?)

Overwrite Kernel memory

- With the ability to write to kernel we can:
  - Overwrite return address
  - Overwrite function pointer
  - Overwrite switch jump table
  - Overwrite Service Descriptor Table
  - etc
- Once overwritten, kernel will jump to us when it reach that code

Pseudo exploitation

- Determine output value of the vulnerable **DeviceIoControl**()
- Allocate memory which Device will jump to
  - **hMem = VirtualAlloc(myAddress, 0xf000, MEM_COMMIT, PAGE_EXECUTE_READWRITE);**
- Copy the shellcode into allocated memory
- Open the driver
  - **handler = CreateFile()**
- Send first signal to overwrite jump table
  - **DeviceIoControl(handler, 0xXXXXXXX7, inBuffer, 0x20, outBuffer, 4, &n, 0))**
- Send second signal to jump to shellcode

Overwrite any memory

- Overwrite switch jump table
- Many Device Driver has switch statement to process user request in **DriverDispatcher**() that look like this:

```
NTSTATUS NPF_IoControl(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp)
{…
switch (FunctionCode){
            case BIOCGSTATS: //function to get the capture stats
               …
               EXIT_SUCCESS(26);
               break;
            case BIOCGEVNAME:
               …
               break;
            case BIOCSENDPACKETSSYNC:
            …
}
```

Switch jump table

- In Assembly:

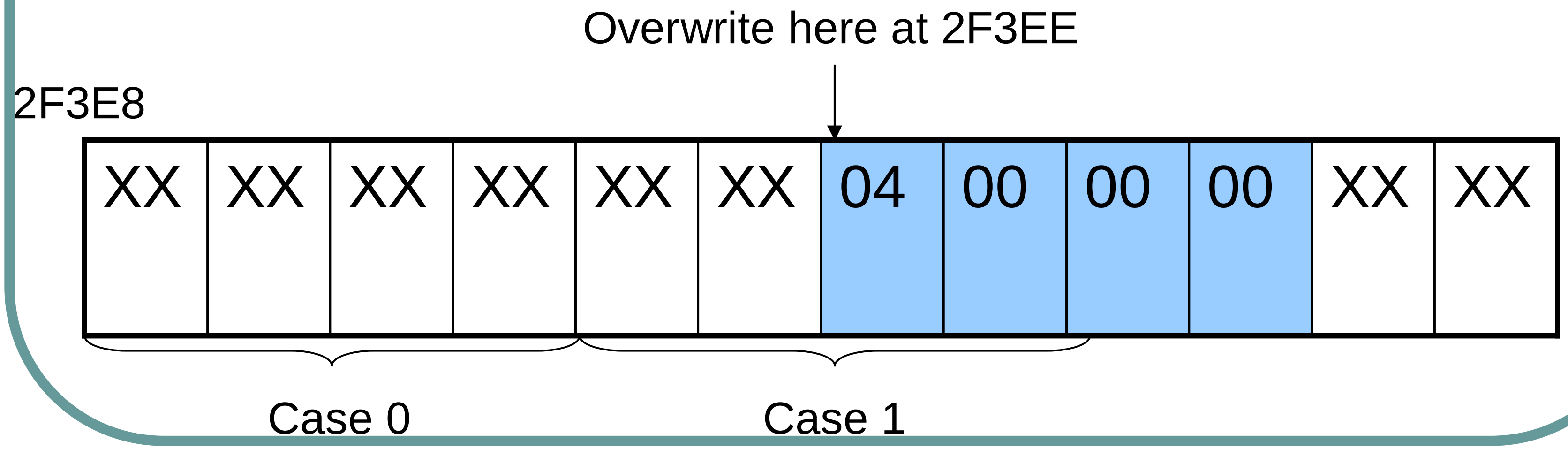
```
PAGE:0002F049 loc_2F049:                          ; CODE XREF: sub_2F038+D  j
PAGE:0002F049              mov    eax, [ebp+arg_0]
PAGE:0002F04C              dec    eax
PAGE:0002F04D              cmp    eax, 0Fh        ; switch 16 cases
PAGE:0002F050              ja     loc_2F3E1       ; default
PAGE:0002F056              jmp    ds:off_2F3E8[eax*4] ; switch jump
...
PAGE:0002F3E8 off_2F3E8    dd offset loc_2F05D    ; DATA XREF:
    sub_2F038+1E  r
PAGE:0002F3E8              dd offset loc_2F08C    ; jump table for switch statement
PAGE:0002F3E8              dd offset loc_2F0AF
PAGE:0002F3E8              dd offset loc_2F0B9
PAGE:0002F3E8              dd offset loc_2F0C3
PAGE:0002F3E8              dd offset loc_2F0F4
PAGE:0002F3E8              dd offset loc_2F125
PAGE:0002F3E8              dd offset loc_2F154
```

Where to Overwrite ?

- We can overwrite the first switch case at 0x2F3E8 with address of our shellcode
- Then, we call the **DeviceIoControl**() again
- When it reach the first switch case again, it will jump to our shellcode
- However, the output from **DeviceIoControl**() is always fixed at 0x4

Overwrite

- Address always overwritten with 0x4
- If we overwrite case 0 with 0x4, the next call to it will jump to 0x00000004
- We cant allocate memory at 0x00000004
- So, we overwrite the first two bytes of the second case



Overwrite here at 2F3EE

2F3E8

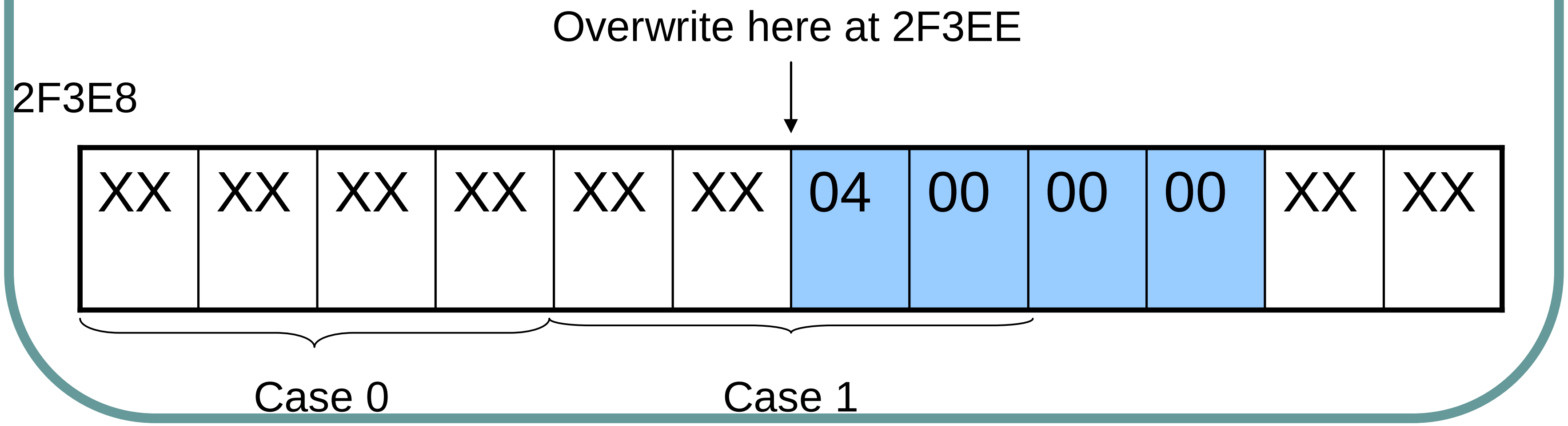
XX	XX	XX	XX	XX	XX	04	00	00	00	XX	XX

Case 0

Case 1

… Overwrite

- Now, if we trigger Case 1, it will jump to:
  - 0x0004XXXX
- We can allocate memory 0x00040000 before calling Case 1



Overwrite here at 2F3EE

2F3E8

XX	XX	XX	XX	XX	XX	04	00	00	00	XX	XX

Case 0

Case 1

Jump to shellcode

- Device driver will jump in to 0x0004XXXX after the second signal
- We need to allocate specific memory region:
  - **VirtualAlloc(0x00040000, 0xf000, MEM_COMMIT, PAGE_EXECUTE_READWRITE);**
- Copy our shellcode into the region

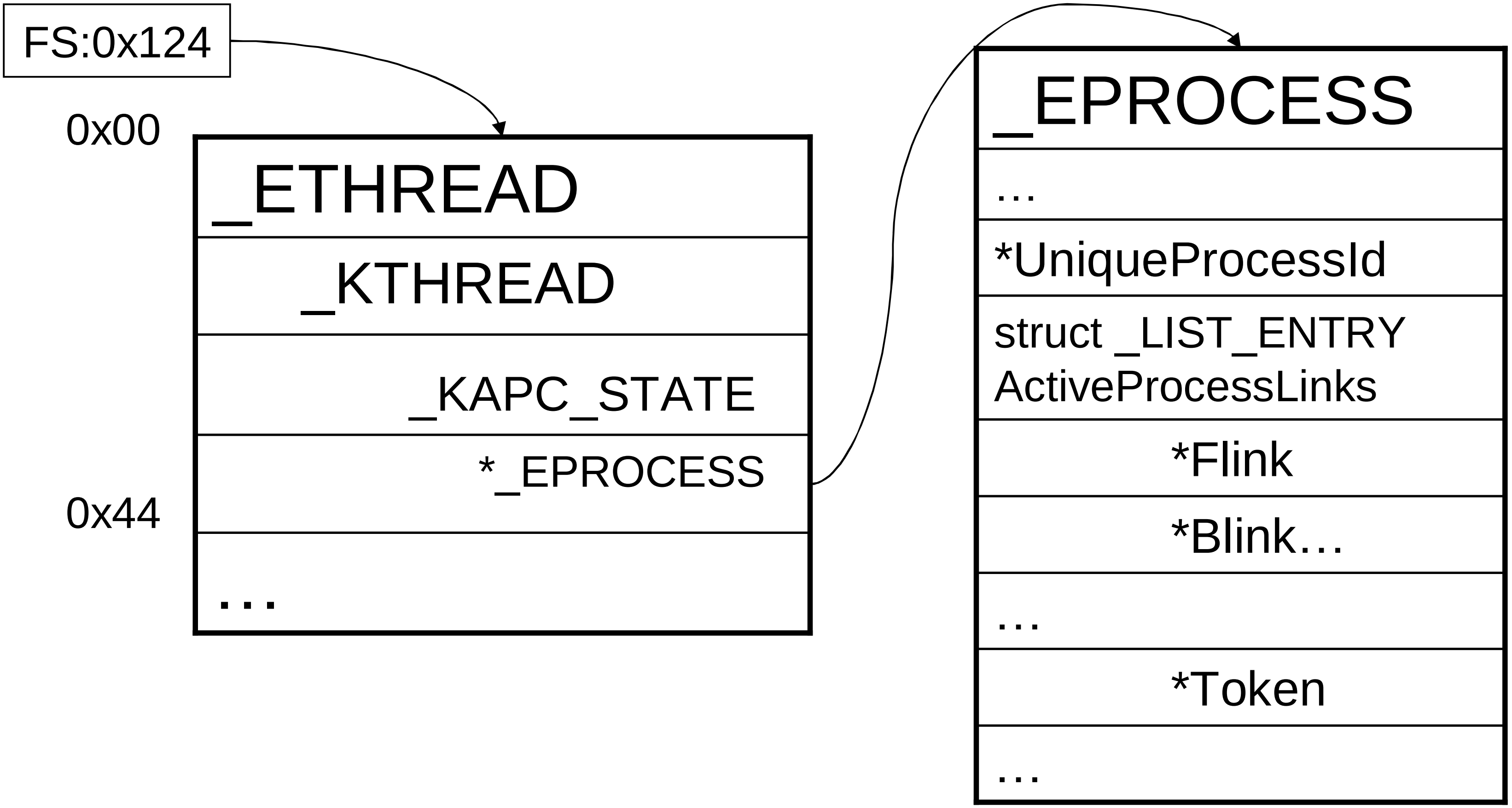
Kernel Shellcode (Eyas' style)

- What do we need to execute?
- Written by Eyas
- http://www.xfocus.net/articles/200306/545.html
- Technique:
  - Find System's token
  - Replace process's token pointer with System's token

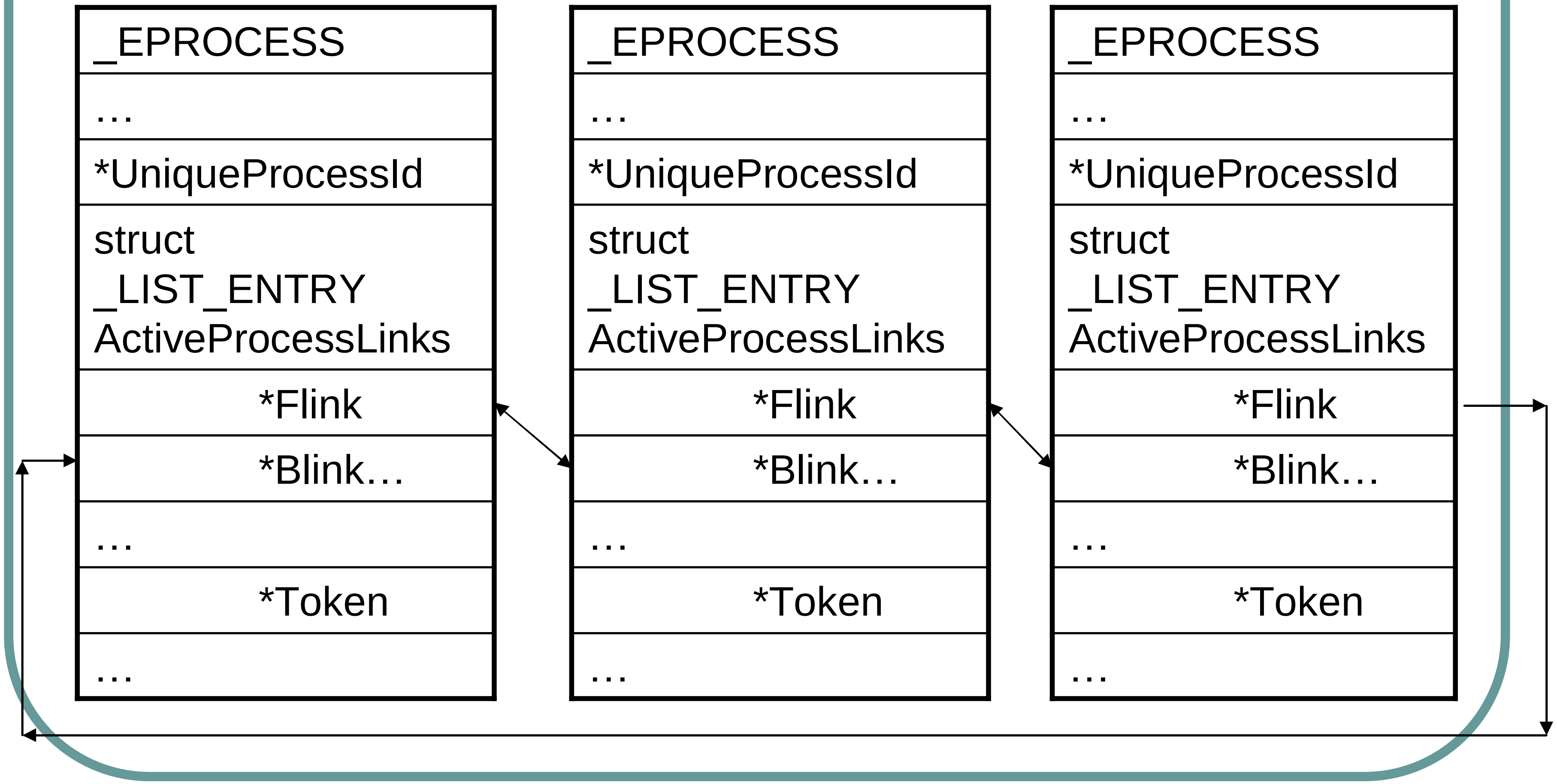
Find SYSTEM process

- Locate the ETHREAD
  - **fs:[0x124] or 0xffdff124**
- From ETHREAD, we jump to EPROCESS
- Within EPROCESS, use **ActiveProcessLinks** to loop into all active process
- For each process, check the UniqueProcessId
- SYSTEM Pid is:
  - Win2k = 8
  - WinXP = 4
- Can use similar technique to find other PID

Locating SYSTEM process



Loop between processes

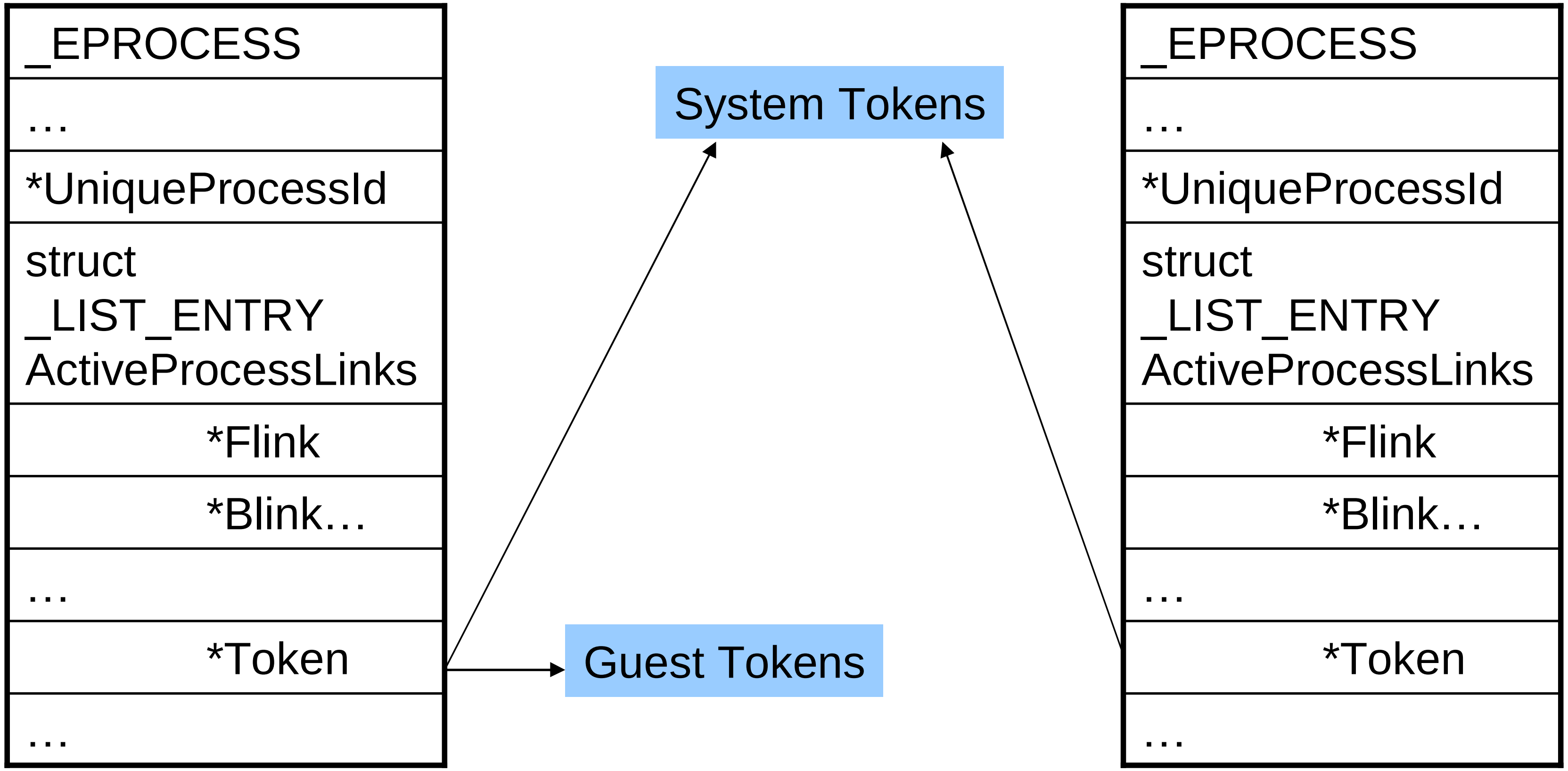


_EPROCESS	_EPROCESS	_EPROCESS
…	…	…
*UniqueProcessId	*UniqueProcessId	*UniqueProcessId
struct _LIST_ENTRY ActiveProcessLinks	struct _LIST_ENTRY ActiveProcessLinks	struct _LIST_ENTRY ActiveProcessLinks
*Flink	*Flink	*Flink
*Blink…	*Blink…	*Blink…
…	…	…
*Token	*Token	*Token
…	…	…

Replace Token Pointer

- Windows's Security Reference Monitor (SRM) uses token to identify process or thread
- To become SYSTEM, we just need a SYSTEM token
- A pointer to SYSTEM token is inside its EPROCESS
- Once we located SYSTEM process, we change our process token to point to SYSTEM token

Getting System Token



_EPROCESS
…
*UniqueProcessId
struct _LIST_ENTRY ActiveProcessLinks
*Flink
*Blink…
…
*Token
…

Guest process

System Tokens

Guest Tokens

_EPROCESS
…
*UniqueProcessId
struct _LIST_ENTRY ActiveProcessLinks
*Flink
*Blink…
…
*Token
…

System process

Base address of Device Driver

- Need to overwrite the exact location of switch table
- Device driver base memory change every boot
- Use **NtQuerySystemInformation**()
- Get **SystemModuleInformation** list
- Compare Module name to get based address of any device driver

Getting process name

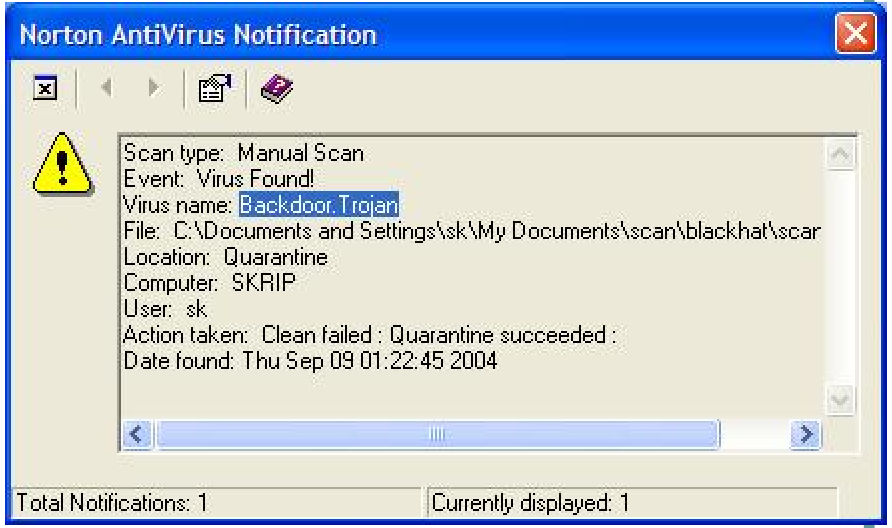
- Using **NtQuerySystemInformation**() again but getting processes list **SystemProcessesAndThreadsInformation**
- Compare **ProcessName** to get **ProcessId**
- For each **ProcessId**, escalate it to SYSTEM

Proof of Concept

- The complete exploit is available from:
  - www.scan-associates.net/papers/navx.c

Attack scenario

- Server allows us to upload *.*
- But every time we uploaded cmd.asp, it disappear
- Apparently, Norton A/V detects cmd.asp as trojan and delete it



Scan type: Manual Scan
Event: Virus Found!
Virus name: Backdoor.Trojan
File: C:\Documents and Settings\sk\My Documents\scan\blackhat\scar
Location: Quarantine
Computer: SKRIP
User: sk
Action taken: Clean failed : Quarantine succeeded :
Date found: Thu Sep 09 01:22:45 2004

Total Notifications: 1
Currently displayed: 1

Encoding script

- Encode cmd.asp using Microsoft Script Encoder
  - http://www.microsoft.com/downloads/details.aspx?FamilyId=E7877F67-C447-4873-B1B0-21F0626A6329&displaylang=en
- Upload cmdx.asp to get arbitrary command execution
- But we only get IUSR user **L**

Privilege escalation

- Upload navx.exe
- Run navx.exe
- Exploit escalate all DLLHOST into SYSTEM
- Command in cmdx.asp is now running as SYSTEM

- Thank you XFocus
- Thank you!
- Q & A