Attacking the Windows Kernel

Below The Root

Jonathan Lindsay, Reverse Engineer in extremis

NGS Consulting
Introduction

Limited to Windows, and aimed at IA32:
• Outline of protected mode and the kernel
• Attack vectors
• Useful tools
• Examples
• Defensive measures
• Future directions
Architecture Overview
A long time ago in a galaxy far, far away…

The progression from Intel’s 8088 to 80386, via the 80286, added:

- Page and segment level protection
- Call, interrupt and task gates
- Privileged and sensitive instructions
- Four privilege levels underlying the protection mechanisms above
- 32bit support
The supervisor

The NT kernel provides:

• Segregation of user mode processes
• Protection of the kernel from user mode
• Provide services to user mode and other kernel mode code
• Session management and the Windows graphics subsystem
The NT kernel

- System call and DeviceIoControl covered
- Graphics drivers
  - Display driver
  - Miniport driver
- NDIS and TDI
- Port objects
- Windows Driver Framework
- Kernel mode callbacks
- Hardware interfaces
  - Talking to hardware
  - Listening to hardware
A plan of attack

• Directly from user mode?
  – CPU bugs
  – Operating system design
• Public APIs
  – StartService, DeviceIoControl, ExtEscape
• Undocumented APIs
  – ZwSystemDebugControl, ZwSetSystemInformation
• Architectural flaws
• Bugs in code
• Subverting operating system initialization
• Modifying kernel modules on disk
  – Viruses
  – DLL (export driver) injection
Tools of the trade
Two different approaches

• Dynamic analysis
  – Will not guarantee results
  – Fuzzing awkward to automate

• Static analysis
  – Can be complicated and time consuming
  – Source code very helpful

• Best results achieved by combining both
Static analysis

• Static driver verifier
• PREFast
• Disassembler
• Windows Driver Kit
  – Documentation and header files
Dynamic analysis

- WinDbg
- Driver verifier
- Miscellaneous
  - WinObj
  - NtDispatchPoints
  - Rootkit Hook Analyzer
Getting our hands dirty
I have the tools, now what?

- Poor access control
- Trusting user supplied data
  - Pointers and lengths
- Typical coding bugs
  - Boundary conditions
  - Off-by-one errors
- Design flaws
  - Expose kernel functionality or data
Reverse engineering

- Knowing the correct entry points means code coverage can be guaranteed
- Subtle bugs are easier to find - signedness
- Memory overwrites are very easy to find
- Highlight areas of code more suited to fuzzing
- No need to analyze a crash dump
- Lack of symbolic information may prove awkward
CDFS DispatchDeviceControl

```
mov    ebx, [ebp+IRP]
push   esi
mov    esi, [ebx+60h]
push   edi
mov    edi, [ebp+Context]
lea    eax, [ebp+var_4]
push   eax
lea    eax, [ebp+IRP]
push   eax
push   dword ptr [esi+18h]          ; Get and decode the FileObject
push   edi
call   C_decodeFileObject
cmp    eax, 2
jz     short loc_15745
mov    esi, 0C000000Dh
                ; Check it’s a valid request

loc_15739:
push   esi
push   ebx
push   edi
call   C_decodeCompleteRequest
mov    eax, esi
jmp     short loc_15799

; ---------------------------------------------------------------------------

loc_15745:          ; Get the IoControlCode from
                    ; IRP.Tail.CurrentStackLocation
mov    eax, [esi+0Ch]
cmp    eax, 24000h
jnz    short loc_157A0
mov    eax, [ebp+IRP]
push   dword ptr [eax+40h]
push   edi
call   C_decodeVerifyVcb

loc_1575B:          ; Verify the Volume Control
                    ; and proceed with the request
```
Source code analysis

• Access to source is not common
• Source code and a suitable IDE will greatly improve auditing speed
• Assumptions made by the coder may help hide subtle bugs
• Tools are available to help speed up the process even further
• grep FIXME –r *.*
if (TypeOfOpen != UserVolumeOpen) {
    CdCompleteRequest(IrpContext, Irp, STATUS_INVALID_PARAMETER);
    return STATUS_INVALID_PARAMETER;
}

if (IrpSp->Parameters.DeviceIoControl.IoControlCode == IOCTL_CDROM_READ_TOC) {
    //
    //  Verify the Vcb in this case to detect if the volume has changed.
    //
    CdVerifyVcb(IrpContext, Fcb->Vcb);
    //
    //  Handle the case of the disk type ourselves.
    //
    } else if (IrpSp->Parameters.DeviceIoControl.IoControlCode == IOCTL_CDROM_DISK_TYPE) {
        //
        //  Verify the Vcb in this case to detect if the volume has changed.
        //
        CdVerifyVcb(IrpContext, Fcb->Vcb);
        //
        //  Check the size of the output buffer.
        //
        if (IrpSp->Parameters.DeviceIoControl.OutputBufferLength < sizeof(CDROM_DISK_DATA)) {
            CdCompleteRequest(IrpContext, Irp, STATUS_BUFFER_TOO_SMALL);
            return STATUS_BUFFER_TOO_SMALL;
        }
Getting a foot in the door

Kernel targets we are interested in:

• Static or object function pointers
• Kernel variables - MmUserProbeAddress
• Descriptor tables
• Return address
• Code from a kernel module
• I/O access map from TSS
• Kernel structures – process token, loaded module list, privilege LUIDs
Real world examples
NT kernel compression support

• Kernel runtime library exports functions to support compression
  – Used by SMB and NTFS

• Support routines take a parameter indicating what algorithm to use
  – Used as an index into a function table

• The table only has 8 entries, whereas the maximum index allowed is 15
  – We can treat code or data as a function pointer, potentially to a user mode address
RtlGetCompressionWorkSpaceSize proc near
sub rsi, 28h
 test cl, cl
movzx r9d, cl
jz short loc_140200E76 ; Check the index is not zero
cmp r9w, 1
jz short loc_140200E76 ; Check the index is not one
test r9b, 0F0h
jz short loc_140200E60 ; Check the index is less than 0x10
mov eax, 0C000025Fh
jmp short loc_140200E7B

loc_140200E60:

movzx eax, r9w
lea r9, RtlWorkSpaceProcs
and cx, 0FF00h ; Mask off the format, and leave only the compression level
call qword ptr [r9+rax*8] ; Call the relevant function from the table
jmp short loc_140200E7B

loc_140200E76:
mov eax, 0C000000Dh
loc_140200E7B.
add rsi, 28h
ret

RtlGetCompressionWorkSpaceSize endp

RtlWorkSpaceProcs dq 0
dq 0

dq offset RtlCompressWorkSpaceSizeLZNT1

dq offset RtlReserveChunkNS

dq offset RtlReserveChunkNS

dq offset RtlReserveChunkNS

dq offset RtlReserveChunkNS

dq offset RtlReserveChunkNS

LZNT1Formats dq 0F0000FFFFh ; With the above code, all the following quadwords
dq 1000001002h ; can be treated as function pointers
dq 7FF0000000Ch
dq 8020000001Fh
dq 0800000020h
dq 3F000003F0h
dq 4000000402h
dq 1FF0000000Ah
Trusting user input

• The following code takes a pointer from a buffer supplied by the user and trusts it
  – Either a sign-extended kernel stack address or an internal handle will be written there
• This can be used to overwrite other code or data, allowing arbitrary code execution
• User supplied pointers into:
  – user mode should be validated
  – kernel mode should be opaque, e.g. a handle
SubFunction:

test esi, esi                      ; Check it is a valid handle
jz InvalidParameter

test ebp, ebp                      ; Check we have a non-NULL input buffer pointer
jz InvalidParameter

mov edi, [esp+9Ch+OutBuffer]
test edi, edi                      ; Check we have a non-NULL output buffer pointer
jz InvalidParameter
cmp edx, 20h                       ; Check the size of the input buffer is 0x20
jnz InvalidParameter
cmp edx, ecx                       ; Check the output buffer is the same size
jnz InvalidParameter
mov eax, [ebp+0Ch]
test eax, eax                       ; Verify the user controlled function index
jz short DefaultOp

cmp eax, 7Fh                       ; Get a user controlled pointer from the input buffer
jbe short ValidOp

jmp short ValidOp                   ; Address part of the thread's kernel mode stack

mov ecx, [ebp+10h]                  ; This will set edx to 0xffffffff
lea eax, [esp+9Ch+var_80]

mov dw ptr [ebp+0Ch], 0FFh          ; Write the sign-extended stack address to the user
mov [ecx], eax
mov [ecx+4], edx                    ; specified buffer

jmp short ValidOp                   ;---------------------------------------------

DefaultOp:

mov dw ptr [ebp+0Ch], 41h

ValidOp:

mov edx, [ebp+10h]
mov eax, [ebp+0Ch]
An architectural flaw

- A function designed to allow the modification of arbitrary memory
- Exposed to unprivileged users
- Provided the internal data structure can be figured out, it is then easy to exploit
- Either access control to the driver, or a different architecture is needed
push    ebx
mov     ebx, [esp+Function]         ; Check if it is a memory operation
cmp     ebx, MEMORY_OPERATION
push    ebp
mov     ebp, [esp+4+SourceDescriptor] ; Get a pointer to the source buffer descriptor
jnz     short NoAddress
mov     ebx, [ebp+4]                ; Get the source start address

NoAddress:
mov     eax, [ebp+8]
mov     edx, [eax]
test    edx, edx                    ; Check that the buffer offset is non-zero
jz      short InvalidParameter
test    ebx, ebx
jl      short InvalidParameter
mov     eax, [eax+4]                ; Get the source end address
cmp     eax, ebx
jb      short InvalidParameter
mov     ecx, [esp+4+DestinationSize]
sub     eax, ecx
jmp     short SizeOk

SizeOk:
test    eax, eax
jz      short RequestProcessed      ; Make sure we are copying some bytes
push    esi
push    edi
mov     edi, [esp+0Ch+Destination]  ; Destination address is an arbitrary address passed in
mov     ecx, eax
lea     esi, [edx+ebx]
shr     ecx, 2
rep movsd
mov     ecx, eax
and     ecx, 3
rep movsb
pop     edi
pop     esi
jmp     short RequestProcessed      ; And we’re done
Defensive measures
Current architecture

- Parameter validation
- Code signing – quality control?
- PatchGuard
- Moving functionality into user mode – UMDF, display drivers in Vista
- Restricting access to APIs
  - User restrictions
  - Privilege restrictions
  - Process restrictions
Alternative approaches

• Hypervisor
  – Designed to help virtualization
  – Provides a layer beneath the supervisor
  – It could be used to provide a microkernel architecture

• Microkernel
  – Does not require virtualization hardware
  – Minimizes the attack surface provided by the kernel
  – Increases flexibility with respect to service implementation
  – Microsoft’s Singularity microkernel is strongly typed and uses software based protection
Future work
A problem has been detected and Windows has been shut down to prevent damage to your computer.

The end-user manually generated the crashdump.

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:

*** STOP: 0x000000E2 (0x00000000,0x00000000,0x00000000,0x00000000)
Fuzzing

• Application fuzzing unlikely to crash the OS
• We need to automate crash recovery and analysis:
  – Run in a VM, but what about real hardware?
  – Have bugcheck callbacks
  – Modify the kernel itself
• Fuzzing interfaces is greatly aided by some form of static analysis
Virtualizing the kernel

- Provide a user mode environment that looks the same as the kernel
- Implement user mode compatible APIs where necessary
- Provide basic I/O, PnP, Process Support and executive functionality
- Trap and handle protected and privileged code execution
- Add instrumentation for analysis and logging
Automated binary analysis

- Model basic CPU functionality
  - Instead of processing a specific value, instructions work on a defined range
  - Instructions can modify the range stored in a register
- Allows all code paths to be assessed
  - Large state space
- Determine ranges of values that will hit certain pieces of code
- Heuristic bug detection
In conclusion ...
Summary

• Current NT kernel architecture increases the likelihood of security issues
• Debatable how much effort has gone into securing kernel code
• Some areas of the kernel have not received much attention
• There is plenty of scope for further research and tool development
Questions?

Thanks