CS 290
Host-based Security and Malware

Christopher Kruegel

chris@cs.ucsb.edu

Advanced Memory Corruption Exploits

Advanced Memory Corruption Exploits

- Windows shellcode
- Kernel exploits and shellcode

Windows Shellcode

- System calls are not the answer
  - Native API implemented in ntoskrnl.exe and exposed via ntdll.dll
  - Windows system call interface (int 0x2e or sysenter) changes between versions
  - Windows system call interface is limited and poorly documented (no standard network calls such as open, connect, …)

- Using system calls in Windows shellcode is “bad practice”
  - instead, use library functions (Windows API)
  - first, decide which functions you need
  - then, find their (absolute) addresses

Library Functions

- Which library functions can be used?

- All Windows programs link against two libraries
  - `ntdll.dll` (Native API exports)
  - `kernel32.dll` (base services – processes, files, …)

- `kernel32.dll` contains two important functions
  - `LoadLibraryA(libraryname)`
  - `GetProcAddress(hmodule, functionname)`

- Enough to execute any function we need, but ...
  we have to find their correct addresses first

Finding Function Addresses

- Addresses of library functions can be found with `dumpbin`
  - easy to do, but inflexible (non-portable)
  - problem is that function addresses can differ between Windows versions and service packs

Finding Function Addresses

Visual Studio 2008 Command Prompt

```
C:\WINDOWS\system32>dumpbin /headers kernel32.dll
Microsoft (R) COFF/PE Dumper Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file kernel32.dll

PE signature found

File Type: DLL

FILE HEADER VALUES
             14C machine (x86)
               4 number of sections
        4802A12C time date stamp Mon Apr 14 01:11:24 2008
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
            210E characteristics
                   Executable
                   Line numbers stripped
                   Symbols stripped
                   32 bit word machine
                   DLL

OPTIONAL HEADER VALUES
             10B magic # (PE32)
            7.10 linker version
           83200 size of code
           70200 size of initialized data
               0 size of uninitialized data
            B63E entry point (7C80B63E)
            1000 base of code
           80000 base of data
        7C800000 image base (7C800000 to 7C8F5FFF)
```

7C800000 image base

Finding Function Addresses

Visual Studio 2008 Command Prompt

```
C:\WINDOWS\system32>dumpbin /exports kernel32.dll | more
Microsoft (R) COFF/PE Dumper Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file kernel32.dll

File Type: DLL

  Section contains the following exports for KERNEL32.dll

    00000000 characteristics
    48025BE1 time date stamp Sun Apr 13 20:15:45 2008
        0.00 version
           1 ordinal base
         953 number of functions
         953 number of names

    ordinal hint RVA      name

          1    0 0000A6D4 ActivateActCtx
```

```
        830  33D 00009689 SetWaitableTimer
        831  33E 000666AA SetupComm
        832  33F 00072F84 ShowConsoleCursor
        833  340 000366AE SignalObjectAndWait
[illegible]
        835  342 00002446 Sleep
[illegible]
        837  344 0003974A SuspendThread
        838  345 00010702 SwitchToFiber
        839  346 000329AA SwitchToThread
        840  347 00010BAC SystemTimeToFileTime
        841  348 0002E991 SystemTimeToTzSpecificLocalTime
```

```
  7C800000 image base
+ 00002446 offset
-------------------------
  7C802446 fct address
```

Finding Function Addresses

```
#include "stdafx.h"
#include "Windows.h"

int _tmain(int argc, _TCHAR* argv[])
{
    _asm {
        push 5000
        mov eax, 0x7C802446
        call eax
    }
    return 0;
}
```

Program sleeps for 5 seconds and then exits

Dynamic Addressing

Now, we want to find function addresses dynamically

- two problems need to be solved

1. Kernel32.dll is not always loaded at the same address
   - locate start address of kernel32.dll

2. Addresses of functions inside kernel32.dll may vary
   - locate our two important functions in kernel32.dll

Locating kernel32

- The operating system allocates a Process Environment Block (PEB) structure for every running process
  - The PEB can always be found at fs:[0x30] in the process memory
- The PEB structure contains three linked lists with info about loaded modules that have been mapped into process space
  - One list is ordered by the initialization time
  - kernel32.dll is always the second module to be initialized
- It is possible to extract the base address for kernel32.dll from PEB

Locating kernel32

```
unsigned int find_kernel32()
{
    _asm {
        xor eax, eax
        mov eax, fs:[0x30]      // start of PEB
        mov eax, [eax + 0x0c]   // start of PEB_LDR_DATA
        mov eax, [eax + 0x1c]   // start of first element (ntdll.dll)
        mov eax, [eax]          // start of second element (kernel32.dll)
        mov eax, [eax + 0x8]    // base address of kernel32.dll
    }
}
```

Locating kernel32

- Alternative ways (smaller in size)
  - find a pointer that points into kernel32
  - possible pointers
    - Unhandled Exception Handler default entry (top entry located at fs:[0])
    - via top of stack, referenced via Thread Control Block (TCB – fs:[0x18])
  - search pages backwards in memory until you find one that starts with 'MZ' (actually, 64KB steps sufficient)

Locating kernel32

```
unsigned int find_kernel32_alt()
{
    _asm {
        push esi
        push ecx
        xor ecx, ecx
        mov esi, fs:[ecx]
        not ecx

    find_kernel32_seh_loop:
        lodsd
        mov esi, eax
        cmp [eax], ecx
        jne find_kernel32_seh_loop
        mov eax, [eax + 0x04]

    find_kernel32_base:
        dec eax
        xor ax, ax
        cmp word ptr [eax], 0x5a4d
        jne find_kernel32_base

        pop ecx
        pop esi
    }
}
```

Locating GetProcAddress

- Use the *image export directory* of the DLL (.edata)
  - declares exported functions, using the following four tables:

    address table (relative virtual addresses – indexed by ordinal)

    name pointer table (pointer to strings)

    ordinal table (same order as name pointer table)

    name table (actual string data)

- Algorithm to obtain address (RVA) for symbol “ExportName”

```
i = Search_ExportNamePointerTable(ExportName);
ordinal = ExportOrdinalTable [i];
SymbolRVA = ExportAddressTable [ordinal - OrdinalBase];
```

Locating GetProcAddress

- To resolve a symbol one must
  - search it in the name table (via name pointer table)
  - the corresponding entry in the ordinal table is function index
  - use index to retrieve the function virtual address from address table

- Storing function names as strings in the shellcode is bad
  - takes too much space
  - solution:

    hash function names (and only store hashes in shellcode)
  - requires that shellcode comes with a hash function

Payloads

- Once functions can be located …
  - (Reverse) Bindshell
    kernel32.dll: CreateProcessA
    ws2_32.dll: WSASocketA, connect, bind, listen, accept
  - Download / Execute
    kernel32.dll: CreateFile, CreateProcessA
    wininet.dll: InternetOpenUrlA and InternetReadFile

Advanced Memory Corruption Exploits

- Windows shellcode

- Kernel exploits and shellcode

Kernel Exploits

- What types of kernel space vulnerabilities are there?
  - invalid (user) pointer dereference
  - kernel stack buffer overflows
  - heap (slab) overflows

  …

- What is special about the payload?
  - *locate* other functions (making a system call is not an option)
  - *stage* standard (user mode) payload
  - *recover* to prevent kernel crash

- In general, most kernel exploits require some special twist

Locating Functions

- Quite similar to what we have just seen
  - need to find exported kernel functions
  - typically, functions are used by kernel modules / device drivers
  - scan memory for known byte signature

    'MZ' at beginning of ntoskrnl.exe

    system call table signature (and known offsets into table)

Stager

- Copy the ring0 or ring3 to a suitable location
  - currently loaded pages of a process
  - Windows `SharedUserData`
  - space between kernel stack and thread_info
  - unused entries in the IDT
  - Asynchronous Procedure Calls (APCs)

Stager

- Problem
  - sometimes, exploit happens in interrupt context
  - no process associated with kernel code, cannot block or sleep

- Install a hook that executes payload later (in desired context)
  - interrupt handler
  - system call handler
  - MSR (mode specific register) – used with `sysenter`
  - saved process return address
  - system call gate (in Windows: `SharedUserData`)

Recovery

- If the system crashes after the stager has finished, we have not accomplished anything
  - need to recover from the exploit and leave system in a safe state
- Recovery depends on the situation
  - restore registers (but we smashed the stack...)
  - enable interrupts or preemption
  - release spinlocks
- Standard tricks
  - spin thread
  - throw exception (rarely possible)
  - restart thread
  - walk stack until valid frame is detected

Kernel NULL dereference

- Kernel developers make mistakes too …
  - kernel code can access a NULL pointer, or it can
  - call a function through a NULL pointer
    (function pointers are quite common in kernel code)

- Normally, this just “crashes” the kernel (oops)
  - can be viewed on console or with `dmesg`

- However, a NULL pointer really points to address 0, which lies in lower (user) part of the address space

- The reason is that the kernel doesn’t switch address spaces but “reuses” the one of the process that invoked system call

Kernel NULL dereference

- Exploit
  - map valid code to address 0 (first page)
  - trigger NULL pointer dereference
  - kernel will happily execute our code with kernel privileges
- Payload
  - simply set privileges of current process to root

Kernel NULL dereference

- CVE 2009-2692

```
static ssize_t sock_sendpage(struct file *file, struct p
                                                  int offset, size_t size, lc
more)
{
        struct socket *sock;
        int flags;

        sock = file->private_data;

        flags = !(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
        if (more)
                flags |= MSG_MORE;

        return sock->ops->sendpage(sock, page, offset, size, flags);
}
```

Kernel NULL dereference

- Possible defense
  - disallow mapping page to address 0

    `/proc/sys/vm/mmap_min_addr`

- Can be bypassed

  `http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html`