bitsec

KERNEL WARS:

KERNEL-EXPLOITATION DEMYSTIFIED

Introduction to kernel-mode vulnerabilities and exploitation

- Why exploit kernel level vulnerabilities?
  - It's fun!
  - Relatively few are doing it
  - Bypasses defense mechanisms and restrictions
  - Attacks at the lowest level
    - Does not rely on any particular application being installed
    - Does not rely on how applications are configured
    - Does not rely on file / registry permissions

Introduction to kernel-mode vulnerabilities and exploitation

- Reasons not to exploit kernel level vulnerabilities
  - Usually one-shot, exploit needs to be very reliable
  - Kernel debugging can be tedious setting up
  - Need some knowledge about kernel internals

Introduction to kernel-mode vulnerabilities and exploitation

- Common targets for attack in a kernel
  - Systemcalls
  - I/O and IOCTL-messages through devicefiles
  - Handling of files in pseudofilesystems (like procfs)
  - Handling of data from the network (wireless/wired)
  - Interaction with hardware (USB, Firewire, etc)
  - Executable file format loaders (ELF, PE, etc)

Introduction to kernel-mode vulnerabilities and exploitation

- Payload strategy
  - Elevating privileges
    - Altering the UID-field (Unix)
    - Stealing access tokens (Windows)
  - Breaking chroot / jail / SELinux / other restrictions
    - Everything can be bypassed in kernel-mode
    - Ring 0: One ring to rule them all..
  - Injecting backdoors
    - Stealth! Do everything in kernel-mode

Introduction to kernel-mode vulnerabilities and exploitation

- Payload techniques
  - Determining addresses and offsets
    - Resolving symbols
    - Pattern matching
    - Hardcoding (last resort)

Introduction to kernel-mode vulnerabilities and exploitation

- Payload techniques
  - OS/architecture-specific techniques
    - Windows/x86: **`ETHREAD`**-pointer at **`0xFFDFF124`** (**`fs:0x124`**)
    - FreeBSD/x86: **`proc`**-pointer at **`[fs:0]`**
    - Linux/x86: **`task_struct`**-pointer at **`esp & 0xffffe000`**
    - NetBSD/x86: **`proc`**-pointer **`[[fs:4]+20]+16`**
    - Solaris/AMD64: **`_kthread`**-pointer at **`[gs:0x18]`**
    - Solaris/i386: **`_kthread`**-pointer at **`[gs:0x10]`**
    - Solaris/SPARC: **`_kthread`**-pointer in **`g7`**

Introduction to kernel-mode vulnerabilities and exploitation

- Exploitation
  - Don't overwrite/trash more than necessary!
  - Cleaning up
    - May need to rewind the stack
    - May need to repair the heap
    - May need to restore overwritten data

Windows Local GDI Kernel Memory Overwrite

- About the bug
  - GDI Shared Handle Table = Memory section with GDI handle data
  - Shared between usermode/kernelmode
  - Mapped (read-only) into every GUI-process
  - Turns out it can be remapped read-write, after bruteforcing the shared memory section handle!
  - BSOD is trivial, but can it be exploited?

Windows Local GDI Kernel Memory Overwrite

- Finding the vulnerability
  - I didn't, Cesar Cerrudo from Argeniss found it
  - The bug was made public 2006-11-06 (MoKB)
  - Microsoft was notified of the bug 2004-10-22...
  - Affected all W2K/WXP systems
  - Patched a few weeks after our talk at Blackhat Europe… ;-)

Windows Local GDI Kernel Memory Overwrite

- Reliably determining the GDI section handle
  - The GDI section = Array of structs with these fields:

`pKernelInfo`	Pointer to kernelspace GDI object data
`ProcessID`	Process ID
`_nCount`	Reference count?
`nUpper`	Upper 16 bits of GDI object handle
`nType`	GDI object type ID
`pUserInfo`	Pointer to userspace GDI object data

  - Each entry = 16 bytes

Windows Local GDI Kernel Memory Overwrite

- Reliably determining the GDI section handle
  - In Windows 2000, 0x4000 entries
  - So GDI section size >= 0x40000 bytes
  - In Windows XP, 0x10000 entries
  - So GDI section size >= 0x100000 bytes

Windows Local GDI Kernel Memory Overwrite

- Reliably determining the GDI section handle
  - Lower 16 bits of a GDI object handle
    = Index into the array in the GDI section
  - Upper 16 bits of a GDI object handle
    = Value of the nUpper-field in the struct

Windows Local GDI Kernel Memory Overwrite

- Reliably determining the GDI section handle
  - Final method:
    - Create a GDI object, handle value = **`H`**
    - Index into table = **`H & 0xFFFF`** (lower 16 bits of **`H`**)
    - nUpper = **`H >> 16`** (upper 16 bits of **`H`**)
    - For each valid shared memory section handle, check if:
      - Section size >= 0x40000 (W2K) / 0x100000 (WXP)
      - **`pGDI[(H & 0xffff)].ProcessID == ExploitPID`**
      - **`pGDI[(H & 0xffff)].nUpper == H >> 16`**
      - **`pGDI[(H & 0xffff)].nType == <TypeID>`**

bitsec

Windows Local GDI Kernel Memory Overwrite

- Finding a way to exploit the bug
  - Two main points of attack:
    - **`pKernelInfo`** : Used in kernel context
    - **`pUserInfo`** : Used in a privileged process
  - Pointers are always interesting targets...
  - Goal: Being able to write to an arbitrary memory address, once that is achieved turning it into arbitrary code execution should be trivial

Windows Local GDI Kernel Memory Overwrite

- Finding a way to exploit the bug
  - Exploiting through a privileged process would most likely be very hard to do reliably, even harder to do generically and chances are quite slim it would be portable across both Windows 2000 and XP
  - Attacking the kernel directly would bypass any hardening measures
  - And of course.. Kernelmode = More fun! ;-)

Windows Local GDI Kernel Memory Overwrite

- Attacking the `pKernelInfo` pointer
  - The naive approach:
    - Overwrite it with trash and hope it ends up in EIP o_O
  - A more realistic approach:
    - Try different kinds of GDI objects (windows, fonts, brushes, etc)
    - Point the `pKernelInfo` into valid usermode memory
    - Fill that memory with an easily recognizable pattern
    - Call GDI related system calls and see if they end up crashing
    - Analyze the crash in WinDBG, analyze the code with IDA Pro
    - Look for dereferences of data in our fake struct

Windows Local GDI Kernel Memory Overwrite

- My final attack
  - Create a **`BRUSH`**-object
  - Point the **`pKernelInfo`** pointer into usermode data with:
    - **`FakeKernelObj[0] = <Evil GDI Object Handle>`**
    - **`FakeKernelObj[2] = 1`**
    - **`FakeKernelObj[9] = <Target Address>`**
  - Call **`NtGdiDeleteObjectApp(<Evil GDI Object Handle>)`**
  - Boom! **`0x00000002`** is written to **`<Target Address>`**
  - Turns out to be a reliable method for all the vulnerable systems

Windows Local GDI Kernel Memory Overwrite

- Now what?
  - Need to find a suitable function pointer to overwrite and a method for determining its address
  - Can only write the fixed value 2 (byte sequence: `02 00 00 00`)
  - We can use two partial overwrites to construct a high address that can be mapped with `VirtualAlloc()`
  - Or we can use `NtAllocateVirtualMemory()` directly and "fool" it into mapping the `NULL` page, where we place our code

Windows Local GDI Kernel Memory Overwrite

- Determining where to write
  - There are probably many function pointers in the kernel that can be used, we need to make sure we use one that these conditions holds for though:
    - Should be possible to reliably determine its address
    - Should be called in the context of our exploit process
    - Should be rarely used, specifically it must not be used during the time between us overwriting it and us triggering a call to it within the context of our exploit
  - An obvious choice is a rarely used system call

Windows Local GDI Kernel Memory Overwrite

- Determining where to write
  - The system call pointers are stored in two tables:
    - **`KiServiceTable`**
    - **`W32pServiceTable`**
  - **`KiServiceTable`** contains the native NT API
  - **`W32pServiceTable`** contains the system calls for the Win32 subsystem (which includes GDI)

Windows Local GDI Kernel Memory Overwrite

- Determining where to write
  - My first choice was a pointer in **`KiServiceTable`**
  - There are documented ways to determine its address, specifically I used a popular method posted to the rootkit.com message board under the pseudonym 90210
  - Worked great!
  - Except under Windows XP SP1...

Windows Local GDI Kernel Memory Overwrite

- Determining where to write
  - So why exactly didn't it work?
  - Turns out that **`KiServiceTable`** actually resides in the read-only text segment of ntoskrnl.exe
  - Read-only kernel pages are usually not enforced
  - I wanted a solution that worked reliably for every Windows 2000 and Windows XP release

Windows Local GDI Kernel Memory Overwrite

- Determining where to write
  - What about **`W32pServiceTable`**?
  - Resides in the data segment of WIN32K.SYS
  - Data segment = writable = perfect!
  - Now the only problem that remains is determining its address, since **`W32pServiceTable`** is not an exported symbol

Windows Local GDI Kernel Memory Overwrite

- Determining where to write
  - Need to come up with my own method
  - One idea was searching for 600 consecutive pointers to the WIN32K.SYS text segment from within the data segment (600+ Win32-syscalls)
  - Not entirely reliable, since there may be unrelated pointers to the text segment right before the start of **`W32pServiceTable`**

Windows Local GDI Kernel Memory Overwrite

- Determining where to write
  - Second and final idea was searching for the call to **`KeAddSystemServiceTable()`** within the ”INIT” section of WIN32K.SYS and searching backwards for the push of the **`W32pServiceTable`** argument
  - Works great!

Windows Local GDI Kernel Memory Overwrite

- Payload
  - Want to elevate the privileges of the exploit process
  - Not as easy as in Unix, need to "steal" an existing access token from a privileged process
  - This method has been used in several of the few other kernelmode exploits for Windows that exists
  - But caused occasional BSOD:s for me, seemingly related to the reference counting of tokens
  - Usually only if the exploit is executed several times on the same box without rebooting it in between

Windows Local GDI Kernel Memory Overwrite

- Payload
  - Solution: Restore the original access token after executing a new privileged process, or whatever it is we wanted to do with our elevated privileges
  - Also restores the overwritten system call pointer
  - Done! Reliable exploitation of the GDI bug across all the vulnerable Windows 2000 and Windows XP systems has been achieved

Windows Local GDI Kernel Memory Overwrite

Demonstration

NetBSD mbuf Overflow

- Finding the vulnerability
  - Fuzzing it
    - Almost instant crash
    - Very similar to NetBSD-SA2007-004, which was demonstrated at our BlackHat Europe talk
  - Tracking it down
    - DDB / GDB
    - Source code
  - Introduction to the bug
    - Mbuf pointer overflow / arbitrary **`MFREE()`**

NetBSD mbuf Overflow

- mbufs
  - Basic kernel memory unit
  - Stores socket buffers and packet data
  - Data can span several mbufs (linked list)

NetBSD mbuf Overflow

- Developing the exploit
  - **`MFREE()`** allow for an arbitrary 32-bit value to be written to an arbitrary address (Normal unlinking stuff)
  - mbuf can have external storage
    - And their own free routine!
      - This is what I'm using in my exploit
      - Exploited mbuf is freed in **`sbdrop()`**

NetBSD mbuf Overflow

```
sbdrop(struct sockbuf *sb, int len)
{
   struct mbuf     *m, *mn, *next;
   next = (m = sb->sb_mb) ? m->m_nextpkt : 0;
   while (len > 0) {
    if (m == 0) {
        if (next == 0)
            panic("sbdrop");
        m = next;
        next = m->m_nextpkt;
        continue;
    }
    if (m->m_len > len) {
        m->m_len -= len;
        m->m_data += len;
        sb->sb_cc -= len;
        break;
    }
    len -= m->m_len;
    sbfree(sb, m);
    MFREE(m, mn);
```

NetBSD mbuf Overflow

- Unlink technique
  - Remove mbuf from chain and link remaining neighboring mbufs together
  - “Arbitrary” write operations takes place

```
#define _MCLDEREFERENCE(m)   \
do {
    (m)->m_ext.ext_nextref->m_ext.ext_prevref =(m)->m_ext.ext_prevref;
    (m)->m_ext.ext_prevref->m_ext.ext_nextref =(m)->m_ext.ext_nextref;
} while (/* CONSTCOND */ 0)
```

NetBSD mbuf Overflow

- Unlink technique example
  - Unlinking an mbuf with these values
    - `m_ext.ext_nextref == 0xdeadbeef`
    - `m_ext.ext_prevref == 0xbadc0ded`
  - Can be expressed as
    - `*(unsigned *)(0xbadc0ded+NN) = 0xdeadbeef;`
    - `*(unsigned *)(0xdeadbeef+PP) = 0xbadc0ded;`
  - Where NN and PP are the offsets to ext_nextref and ext_prevref in the mbuf structure respectively.

NetBSD mbuf Overflow

- Targets to overwrite
  - Return address
  - “Random” function pointer
  - **`sysent`** – function pointers to syscalls
- Cleaning up
  - Memory pools, messy and changes between releases
  - **`mbinit()`**

NetBSD mbuf Overflow

- External free() technique
  - Some mbufs holds a reference to their own free() routine
  - No unlinking is done if **`ext_nextref`** references its own mbuf
  - Point **`ext_free`** to your payload – Job done!
  - Bonus – No mess to clean up

NetBSD mbuf Overflow

- Payload
  - How to find your process
    - I have used allproc and %fs
  - Changing credentials
    - Credential structure pointer found in proc structure
    - Change UID 0
  - “Cheating” my way out of the loop
    - I'm lazy – Return from payload with an extra leave
  - Placing the payload
    - Return to userland

NetBSD mbuf Overflow

Demonstration

OpenBSD IPv6 Remote mbuf Overflow

- Bug found and researched by Alfredo Ortega
- PoC code was released to execute a breakpoint
- I successfully tested the vulnerability against OpenBSD 4.0, 3.9, 3.8, 3.7, 3.6, 3.5, 3.4, 3.3, 3.2, 3.1 (older releases with support for IPv6 might be vulnerable too)
- Code is different in OpenBSD <= 3.6, I focused on 3.7 - 4.0

OpenBSD IPv6 Remote mbuf Overflow

The Bug

- Sending specially crafted fragmented ICMPv6 packets causes an mbuf structure to be overwritten
  - PoC code overwrites **`ext_free`** function pointer
    - **`ecx`**, **`ebx`** and **`esi`** points to start of overwritten mbuf
    - **`jmp <reg>`** and then **`jmp`** backwards to reach stage 1

OpenBSD IPv6 Remote mbuf Overflow

Payload in 3 stages

- Stage 1 – Backdoor installation, **`icmp6_input`** wrapper
- Stage 2 – Backdoor
- Stage 3 – Command(s)

OpenBSD IPv6 Remote mbuf Overflow

Stage 1 (1/2)

- Find stage 2
  - Last mbuf in chain for previous packet (**`%esp+108`**)
- Calculate address of symbol resolver routine (offset from start of stage 2)
- Resolve **`inet6sw`** and fetch **`inet6sw[4].pr_input`**, the address of the current **`icmp6_input`** function.

OpenBSD IPv6 Remote mbuf Overflow

Stage 1 (2/2)

- Make sure backdoor is not already installed
  - Compare the first four bytes of the backdoor with the corresponding bytes in the input function (starts with a **`call`** instead of **`push %ebp`**).
- Allocate kernel memory for stage 2 and copy the code.
- Wrap **`icmp6_input`** with stage 2
  - Replace **`inet6sw[4].pr_input`** with pointer to stage 2
- Clean up stack and return

```
addl    $0x20, %esp
popl    %ebx
popl    %esi
popl    %edi
leave
ret
```

OpenBSD IPv6
Remote mbuf Overflow

Stage 2 (1/4) – ELF symbol resolver

- Find ELF header, which is mapped after .bss
  - Scan for `"\x7FELF"` from Interrupt Descriptor Table
- Search for hash of symbol string in the .dynsym section

OpenBSD IPv6 Remote mbuf Overflow

Stage 2 (2/4)

- Listen for ICMPv6 packets with magic bytes
  - Copy stage 3 code to allocated memory
  - Wrap system call with stage 3 command
- Call the real **`icmp6_input`** routine and return

OpenBSD IPv6 Remote mbuf Overflow

Stage 2 (3/4) – Syscall wrapper

- Exploit will be more portable if system calls are used
- Need process context to use system calls
- **`fork1()`** from **`initproc`** inside an interrupt does not work (anymore)
- Wrap a system call, wait for it to be called, **`fork1()`** from that process

OpenBSD IPv6 Remote mbuf Overflow

Stage 2 (4/4) – Syscall wrapper

- Wrap **`gettimeofday()`** with stage 3, since it is called quite frequently
- Store address of the real **`gettimeofday()`** and its index (116) at the beginning of the stage 3 command

```
# Set syscall address in table
.macro set_syscall sysent, idx, addr
                        movl    \sysent, %ecx
                        movl    \idx, %eax # Index
                        movl    \addr, 4(%ecx, %eax, 8)
.endm
```

OpenBSD IPv6 Remote mbuf Overflow

Stage 3 (1/3) – Commands

- Connect-back
- Set secure level
- Shell commands (**`/bin/sh -c`**)
- Uninstall backdoor

There is no built-in command for transferring files.

Use a connect-back shell and:

- **`uuencode(1)`** + **`cat(1)`** to send (binary) files
- **`script(1)`** + **`uuencode(1)`** to fetch them.

OpenBSD IPv6 Remote mbuf Overflow

Stage 3 (2/3) – Initialization

- Use stage 2 resolver to resolve symbols
- Reset the wrapped system call to the original function pointer
- Call the real system call and save the return value
- **`fork1()`** from the calling process

OpenBSD IPv6 Remote mbuf Overflow

Stage 3 (3/3) – Command process

- Make sure we run as root

```
.macro setuid_root proc
    movl 16(\proc), %eax   # struct pcred pointer
    movl $0, 4(%eax)       # Real User ID
.endm
```

- Terminate the process on failure

OpenBSD IPv6 Remote mbuf Overflow

Demonstration

FreeBSD 802.11 Remote Integer Overflow

- Kernel vulnerability in the IEEE 802.11 subsystem of FreeBSD
  - Vulnerability found and exploited by Karl Janmar
  - IOCTL
  - Integer-overflow
  - Stackbased buffer overflow
  - wpa_supplicant
  - WPA-PSK

FreeBSD 802.11 Remote Integer Overflow

Demonstration

bitsec