bitsec

KERNEL WARS: KERNEL-EXPLOITATION DEMYSTIFIED

- Why exploit kernel level vulnerabilities?
 - It's fun!
 - Relatively few are doing it
 - Bypasses defense mechanisms and restrictions



- Why exploit kernel level vulnerabilities?
 - 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

- 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



- 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)

- Payload strategy
 - Elevating privileges
 - Altering the UID-field (Unix)
 - Stealing access tokens (Windows)
 - Injecting backdoors
 - Stealth! Do everything in kernel-mode



- Payload strategy
 - Breaking chroot / jail / other restrictions
 - Everything can be bypassed in kernel-mode
 - Ring 0: One ring to rule them all..

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

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



• 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



- Kernel vulnerability in the IEEE 802.11 subsystem of FreeBSD
 - Auditing the code
 - Implementing exploit
 - Exploit developed and vulnerability found by Karl Janmar



- Auditing the IEEE 802.11 stack of FreeBSD
 - IEEE 802.11 code in its current shape is relatively new in FreeBSD
- Problems faced auditing the code
 - IEEE 802.11 has a complex link-layer protocol
 - Rough metric, source-code of input functions
 - ieee80211_input() 437 lines
 - ether_input() 107 lines
 - ip_input() 469 lines

- (cont.) Problems facing auditing the code
 - Code is not written to be easily read (not by me at least)
 - Huge recursive switch-statements
 - Example: a 274-line recursive switch-statement in ieee80211_input()
 - Macros that include return statements etc.
- Lots of user-controlled data
 - Link-layer management unauthenticated, unencrypted
- Found a local issue
 - loctl which had a logic error, only kernel-memory disclosure
- Found another interesting issue:

Code of function called by ioctl[SCAN_RESULTS]:

static int

{

```
ieee80211_ioctl_getscanresults(struct ieee80211com *ic, struct ieee80211req *ireq)
```

```
union {
    struct ieee80211req_scan_result res;
    char data[512];    /* XXX shrink? */
} u;
struct ieee80211req_scan_result *sr = &u.res;
struct ieee80211_node_table *nt;
struct ieee80211_node *ni;
int error, space;
u_int8_t *p, *cp;

p = ireq->i_data;
space = ireq->i_len;
error = 0;
```

(continue from previous)

(continue from previous)

```
if (ni->ni wpa ie != NULL) {
           memcpy(cp, ni->ni_wpa_ie, 2+ni->ni_wpa_ie[1]); <---- copy to union u</pre>
           cp += 2+ni->ni_wpa_ie[1];
     if (ni->ni_wme_ie != NULL) {
           memcpy(cp, ni->ni wme ie, 2+ni->ni wme ie[1]); 
           cp += 2+ni->ni wme ie[1];
     error = copyout(sr, p, sr->isr_len);
     if (error)
           break;
     p += sr->isr len;
     space -= sr->isr len;
ireq->i len -= space;
return error;
```

sec

}

static void

{

get_scan_result(struct ieee80211req_scan_result *sr, const struct ieee80211_node *ni)

```
struct ieee80211com *ic = ni->ni_ic;
```

```
memset(sr, 0, sizeof(*sr));
```

```
sr->isr_ssid_len = ni->ni_esslen;
```

```
if (ni->ni_wpa_ie != NULL)
```

sr->isr_ie_len += 2+ni->ni_wpa_ie[1];

```
if (ni->ni_wme_ie != NULL)
```

sr->isr_ie_len += 2+ni->ni_wme_ie[1];

sr->isr_len = sizeof(*sr) + sr->isr_ssid_len + sr->isr_ie_len;

```
sr->isr_len = roundup(sr->isr_len, sizeof(u_int32_t));
```

```
if (ni->ni_chan != IEEE80211_CHAN_ANYC) {
```

sr->isr_freq = ni->ni_chan->ic_freq;

```
sr->isr_flags = ni->ni_chan->ic_flags;
```

bitsec

}

.....

static void

}

.....

{

get_scan_result(struct ieee80211req_scan_result *sr, const struct ieee80211_node *ni)

```
struct ieee80211com *ic = ni->ni_ic;
```

```
memset(sr, 0, sizeof(*sr));
```

```
sr->isr_ssid_len = ni->ni_esslen;
```

```
if (ni->ni_wpa_ie != NULL)
```

sr->isr_ie_len += 2+ni->ni_wpa_ie[1];

```
if (ni->ni_wme_ie != NULL)
```

```
sr->isr_len = sizeof(*sr) + sr->isr_ssid_len + sr->isr_ie_len;
```

```
sr->isr_len = roundup(sr->isr_len, sizeof(u_int32_t));
```

```
if (ni->ni_chan != IEEE80211_CHAN_ANYC) {
```

sr->isr_freq = ni->ni_chan->ic_freq;

```
sr->isr_flags = ni->ni_chan->ic_flags;
```

• Test our theories

- Hardcode test-case into kernel
- Create a custom kernel with debugging facilities
- Modify kernel config:

makeoptions	DEBUG=-g
options	GDB
options	DDB # optional
options	KDB

- Recompile & reboot
- Make sure DDB is enabled
- \$ sysctl -w debug.kdb.current=ddb

- Trigger the affected code

- in this example if config will do the work

Fatal trap 12: page fat	alt while in kernel mode
fault virtual address	= 0x41414155
fault code	= supervisor write, page not present
instruction pointer	= 0x20:0xc06c405c
stack pointer	= 0x28:0xd0c5e938
frame pointer	= 0x28:0xd0c5eb4c
code segment	= base 0x0, limit 0xfffff, type 0x1b
	= DPL 0, pres 1, def32 1, gran 1
processor eflags	= interrupt enabled, resume, IOPL = 0
current process	= 203 (ifconfig)
[thread pid 203 tid 10	0058]
Stopped at ieee802	211_ioctl_getscanresults+0x120: subw %dx,0x14(%eax)

- Can it be triggered remotely?
 - Who is calling this ioctl?
 - Yes! wpa_supplicant regularly calls this ioctl
 - wpa_supplicant is supplied in the base distribution
 - Is needed for 802.1X authentication (WPA-PSK) etc.
- We need to send raw frames
 - BPF in NetBSD was extended to be able to send arbitrary IEEE 802.11 frames

• Switch to better debugging environment – GDB

- Configure kernel to allow kernel-debugging:

In /boot/device.hints:

hint.sio.0.flags="0x80"

- Then switch default debugger:
- \$ sysctl -w debug.kdb.current=gdb

http://www.freebsd.org/doc/en_US.ISO8859-1/books/developershandbook/kerneldebug.html

• We prepare a beacon frame with large SSID,WPA and WME fields

16:32:33.155795 Ous BSSID:cc:cc:cc:cc:cc:cc DA:ff:ff:ff:ff:ff SA:cc:cc:cc:cc:cc:cc Beacon 0x0000: ceef f382 c40b 0000 6400 0100 0020 5858d....XX 0x0020: 5858 5858 5858 5858 5858 5858 5858 0104 XXXXXXXXXXXX.. 0x0030: 8284 0b16 0301 01dd fc00 50f2 0141 4141 . . . 0x0130: 4141 4141 41dd fd00 50f2 0201 4141 4141 AAAAA...P...AAAA . . . 0x0230: 4141 4141 AAAA

sec

• On target when frame is sent:

[New Thread 100058]

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 100058]
0xc06c405c in ieee80211_ioctl_getscanresults
  (ic=0x41414141, ireq=0x41414141)
    at ../../.net80211/ieee80211_ioctl.c:1047
1047 ireq->i len -= space;
```

(gdb) print ireq

\$1 = (struct ieee80211req *) 0x41414141

(gdb) bt

at ../../net80211/ieee80211_ioctl.c:1047

- #1 0x41414141 in ?? ()
- #2 0x41414141 in ?? ()
- #3 0x41414141 in ?? ()
- #4 0x41414141 in ?? ()
- #5 0x41414141 in ?? ()
- #6 0x41414141 in ?? ()

(9	gdb) l	ist ieee80211_ioctl_getscanresults
1(003	static int
1(004	<pre>ieee80211_ioctl_getscanresults(struct ieee80211com *ic, struct ieee80211req *ireq)</pre>
1(005	{
1(006	union {
1(007	<pre>struct ieee80211req_scan_result res;</pre>
1(008	char data[512]; /* XXX shrink? */
1(009	} u;
1(010	<pre>struct ieee80211req_scan_result *sr = &u.res</pre>
1(011	<pre>struct ieee80211_node_table *nt;</pre>
1(012	<pre>struct ieee80211_node *ni;</pre>
1(013	int error, space;
1(014	u_int8_t *p, *cp;
1(015	
1(016	<pre>p = ireq->i_data;</pre>
1(017	<pre>space = ireq->i len;</pre>

(gdb) x/150xw &	2u			
0xd0c5e940:	0x096c0148	0x370000e0	0x8d000164	0x89905342
0xd0c5e950:	0x8482048e	0x0000160b	$0 \times 0000000000000000000000000000000000$	0x00000000
0xd0c5e960:	0x00fd2000	$0 \times 0000000000000000000000000000000000$	0x58585858	0x58585858
0xd0c5e970:	0x58585858	0x58585858	0x58585858	0x58585858
0xd0c5e980:	0x58585858	0x58585858	0x5000fcdd	0x <mark>4141</mark> 01f2
0xd0c5e990:	0x41414141	0x41414141	0x41414141	0x41414141
0xd0c5eb40:	0x41414141	0x41414141	0x41414141	0x41414141
0xd0c5eb50:	0x41414141	0x41414141	0x41414141	0x41414141
0xd0c5eb60:	0x41414141	0x41414141	0x41414141	0x41414141
0xd0c5eb70:	0x41414141	0x41414141	0x41414141	0x41414141
0xd0c5eb80:	0x41414141	0xd0c5eb <mark>41</mark>	0xc063b816	0xc1509d00
0xd0c5eb90:	0xc01c69eb	0xc16eec00		

• • •

(gdb) print \$ebp

\$8 = (void *) 0xd0c5eb4c

- We can overwrite the return-address, what to put there?
 - Return to a jmp ESP or equivalent in .text
- Search in kernel binary after wanted byte sequence

\$ search_instr.py -s 0x003d4518 -f 0x00043c30 -v 0xc0443c30
FreeBSD_GENERIC_i386_6.0
0xc0444797: 0xff 0xd7, call *%edi
0xc04486c4: 0xff 0xd7, call *%edi
...
0xc044c5dd: 0xff 0xd7, call *%edi
0xc044d3d: 0xff 0xe4, jmp *%esp
0xc0450109: 0xff 0xd1, call *%ecx

bitsec

. . .

- Initial payload
 - Can't use stack before overwritten return address
 - Resides after overwritten return address
 - Limited to 32 byte to not destroy a previous frame we want intact
 - Stage a second payload that resides in received beacon frame (in kernel list)



Second stage payload

- Allocate memory for backdoor
- Copy head of backdoor to allocated area
- Save away original management-frame handler
- Overwrite management-frame handler with pointer to our backdoor
- Return from our exploited function, restore frame 2 levels down

– Return a empty scan list and without error Ditsec

- Backdoor in place
 - Backdoor function receive all management-frame
 - Look for magic number at a fixed position, within
 WPA IE field

	offset: 88	offset: 102	offset: 103	offset: 104
ieee802.11	magic number	cmd. len	cmd. type	cmd. data
frame				

- First command initializes the backdoor
- If no magic number found pass frame to real handler

- (cont.) Backdoor in place
 - Send back response as a probe-response
 - Payload is included in the optional response field
 - Spoofed source/destination MAC addresses



- Backdoor command type
 - Ping backdoor
 - Every ping has a unique 32-bit identifier
 - Send back pong response including identifier
 - Upload backdoor-code
 - Every upload has a 16-bit offset and a 251 bytes of possible data
 - Send back ACK response with ACK'd offset
 - Execute backdoor-code
 - All commands have a variable argument data field
 - Send back execution result

- Upload and Execute command
 - The only primitives needed to implement plug-ins
 - Plug-ins doesn't need to handle the actual communication part
- Fileserver plug-in
 - Read file, in 128 byte chunks
 - Stat file, get state information of file
 - Write (and possibly create) file, in 128 byte chunks

- Do FS operations the way the kernel does it
 - Extract the essential calls needed for the operations
 - Open and read in file example:
 - 1. Initialize a *struct nameidata*, the way NDINIT() macro does

 - 3. Try lookup vnode with vn_open()
 - 4. Do the actual read with *vn_rdwr()*
 - 5. Unlock and close vnode using VOP_UNLOCK_APV() and vn_close()
 - Some vnode macros are messy in assembly, disassembling the kernel can help

- Last word
 - Net80211 framework in *BSD is a huge work and deserve credits, brought a lot of good things with it
 - ... but might need some cleaning up and security auditing
- Demonstration

• Finding the vulnerability

- Fuzzing it

sec

- In-house developed fuzzer
- Almost instant crash
- Tracking the bug down
 - DDB / GDB
 - Source code
- Introduction to the bug
 - mbuf pointer overflow / arbitrary m_free()

• Mbufs

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



- Developing the exploit(s)
 - m_free() allows 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!



- Unlink technique
 - Remove mbuf from list 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)
```

- 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 the

ext_nextref and ext_prevref pointers within

the mbuf structure respectively

- Targets to overwrite
 - Return address
 - "Random" function pointer
 - sysent function pointers to syscalls



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

Payload

- Decided to change process credentials to root.
- How to find your process
 - I used allproc and %fs
- Changing credentials
 - Pointer to credential structure in proc structure
 - Change UID to 0
- Placing the payload
 - Return to userland

• Cleaning up

- Memory pools Enough to give anyone a headache
- mbinit() saves the day
- External storage/free() Nothing to clean up!
- Demonstration

- 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?

- 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...
 - But... No exploit and still unpatched!
 - All W2K/WXP systems are vulnerable



- 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

- 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



- 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



- 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 == GetCurrentProcessId()
 - pGDI[(H & 0xffff)].nUpper == H >> 16
 - pGDI[(H & 0xffff)].nType == <TypeID-for-object-wecreated>

- Setting up a kernel debugging environment
 - No previous Windows kernel debugging experience
 - Two main options: SoftICE / WinDBG
 - SoftICE is discontinued since a while back..
 - Better learn WinDBG!



- Setting up a kernel debugging environment
 - WinDBG normally requires a two-machine setup
 - Can emulate this using VMWare, by configuring the virtual serial port to use a named pipe



- 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



- Finding a way to exploit the bug
 - Exploiting via 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 on the target
 - And most importantly.. Kernelmode = More fun! ;-)

- Finding a way to exploit the bug
 - Cannot reveal all details yet since Immunity Inc has bought the rights to the exploit, until the end of april
 - Pointing the pKernelInfo pointer into specially crafted (usermode) data is a good start though!
 - Calling various GDI related syscalls and checking out what happens is also a good idea
 - WinDBG and IDA Pro are your best friends...

- Finding a way to exploit the bug
 - Making a long story short, a reliable way to write a certain fixed value to an arbitrary address was achieved
 - Turns out the same technique could be used on all Windows 2000 and Windows XP systems



- Now what?
 - Need to find a suitable function pointer to overwrite and a method for determining its address
 - Can only write a fixed and very low value
 - We can use two partial overwrites to construct a high address that can be mapped with VirtualAlloc()
 - Or we can use NtAllocateVirtualMemory() directly and "fooling" it into mapping the NULL page

- 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

- 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)



- 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...

- 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 <u>usually</u> not enforced
 - I wanted a solution that works reliably for every
 Windows 2000 and Windows XP release



- 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



- 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



- 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!

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

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 Windows 2000 and Windows XP systems has been achieved



- Final touch: Portability
 - Changed between Windows 2000 and XP:
 - Syscall numbers
 - Token field offsets
 - Exploit automatically adjusts the payload



• Demonstration

