Beginner's Guide to Wireless Auditing

David Maynor 2006-09-19

Introduction

Since our talks at Black Hat Vegas and DEFCON, Jon Ellch and I have been peppered with questions regarding how to find vulnerabilities in wireless device drivers and the specific techniques that were employed. Rather than answer these questions one at a time, an article seemed a better course of action. In this first article, we will discuss how to build an auditing environment, how to construct fuzzing tools and, finally, how to interpret the results.

Although our previous talks have focused primarily on 802.11-based protocols, these same auditing methods can be applied to almost any type of device, including Bluetooth and infrared, with successful results. This article is designed as a beginner's guide to fuzzing wireless device drivers. To get the most out of it you should already be familiar with exploit development and debugging, as the article does not cover either of those topics in depth.

Figure 1. Like poker, but with a different kind of chips!

Building an environment for Wifi auditing

Our Black Hat presentation [ref 1] was entitled "Device Drivers: Don't Build a House on a Shaky Foundation." This concept is true for more than just device drivers, it is true for wireless auditing platforms as well. The most important part of auditing is in first building a good, robust platform to launch attacks from. The underlying operating system is up to you, but I chose to use Fedora Core 3 (while FC5 is out now, I really don't need to do more than wifi auditing). I installed a stock FC3 image. The only additional packages that were installed were done using `yum`. These involved upgrading the kernel to the latest version and

installing a package called `sharutils`. This was achieved by issuing the following commands.

```
[root:~]$yum upgrade kernel

[root:~]$yum install sharutils
```

Although this was done on a Dell Latitude D610, the internal wireless card of the machine was not used. In order to do the raw WiFi packet injection needed for fuzzing, a combination of third-party code and hardware was used. The main component of this is a library called LORCON (Loss Of Radio CONnectivity).

LORCON [ref 2] is a library that gives a programmer the ability craft a WiFi packet from scratch. LORCON is built by patching the third-party madwifi driver [ref 3] for cards based on the Atheros chipset. In order to have the best results, you should pick a card that is well supported by madwifi. For the purpose of this article, I chose the Netgear WPN511. It's a good card that supports almost every feature needed and is well supported by madwifi. It's also not hard to find.

Figure 2. Netgreat WPN511 card used for this article.

Once you have a good environment with all the necessary packages, patch madwifi with LORCON and install it. After the patch process, it should be as simple as issuing the "make" command for most systems. If there is a problem here, refer to the madwifi documentation available on the project site [ref 3]. After the build is complete you need to install the drivers with the "make install" command. You can verify the components are installed by looking in the `/lib/modules//net` directory for the existence of the wlan and ath kernel modules, as shown below in Figure 3.

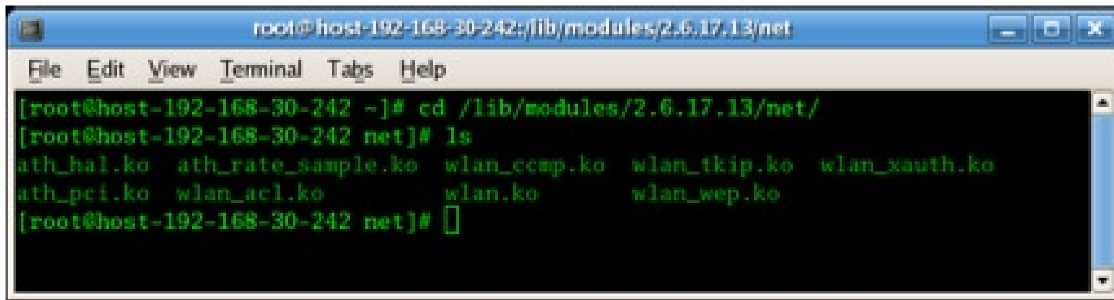

Figure 3. Verifying required components are installed.

Now that you have the drivers, place the card in the PCMCIA slot and you should get a message similar to the one below in Figure 4:

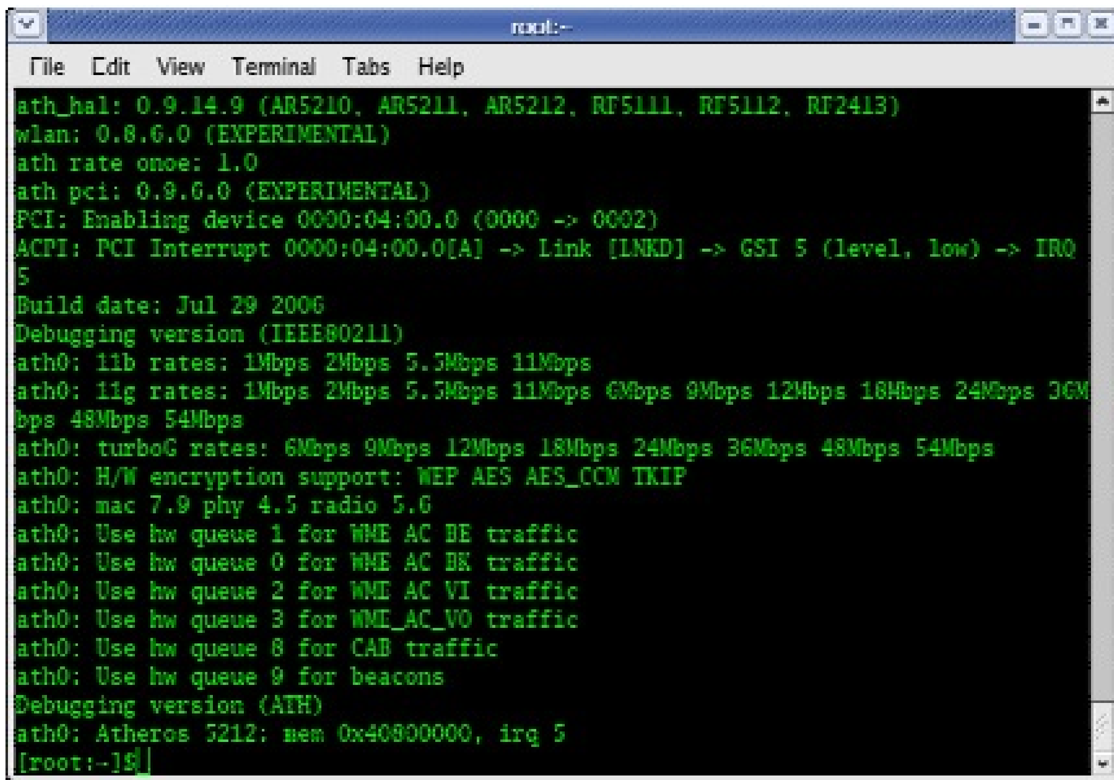

Figure 4. Verifying patched driver is working.

The first step is to bring the card up to a working state. You do this with '`ifconfig ath0 up`'. The usability can be checked by running the ifconfig command again. The card can be tested by issuing a few commands like '`iwlist ath0 scan`' and so on.

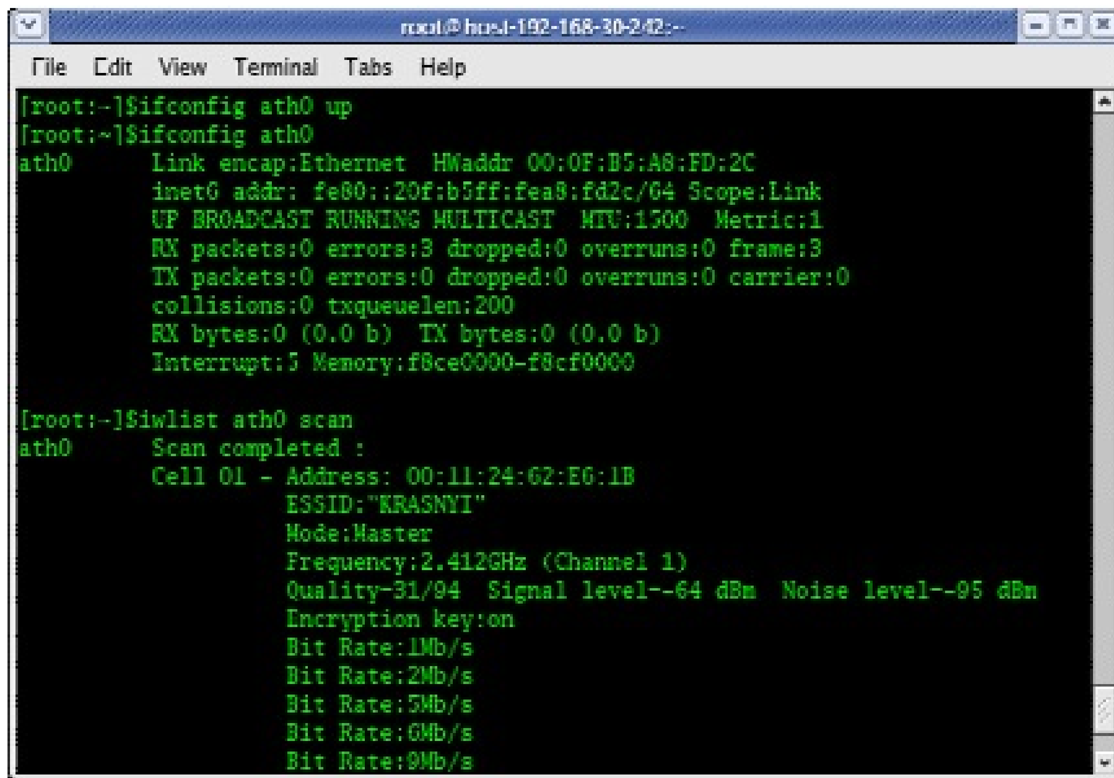

Figure 5. Using ifconfig and iwlist to test the card.

I wrote a shellscript to automate this task. It's useful for not having to repeat the same command over and over again. Mine looks like this:

```
#!/bin/bash
ifconfig ath0 up
ifconfig ath0 192.168.1.1
iwconfig essid "wifiaudit"
iwconfig ath0 mode Master
iwpriv ath0 mode 2
iwconfig ath0 channel 1
```

Now that the environment is set up, it's time to actually build packets to inject - which means you have to write code. All the fuzzers I have developed are written in C and use the LORCON API to develop and the inject packets. If you don't know C or you don't want to spend a lot of time hand developing packet structures, I strongly suggest taking a look at an excellent tool called scapy [ref 4].

Scapy is a packet creation tool written in Python by a programmer named Philippe Biondi. The combination of Python with the way the tool is designed means that, with very little knowledge of networking, you can write a pretty powerful fuzzer quickly. Fortunately, scapy is WiFi aware. Download a copy of it a run it. Don't worry about seeing any errors as they won't affect the basic sending and receiving of packets. Run scapy and do a ls(). This will show you all the different layers available to you.

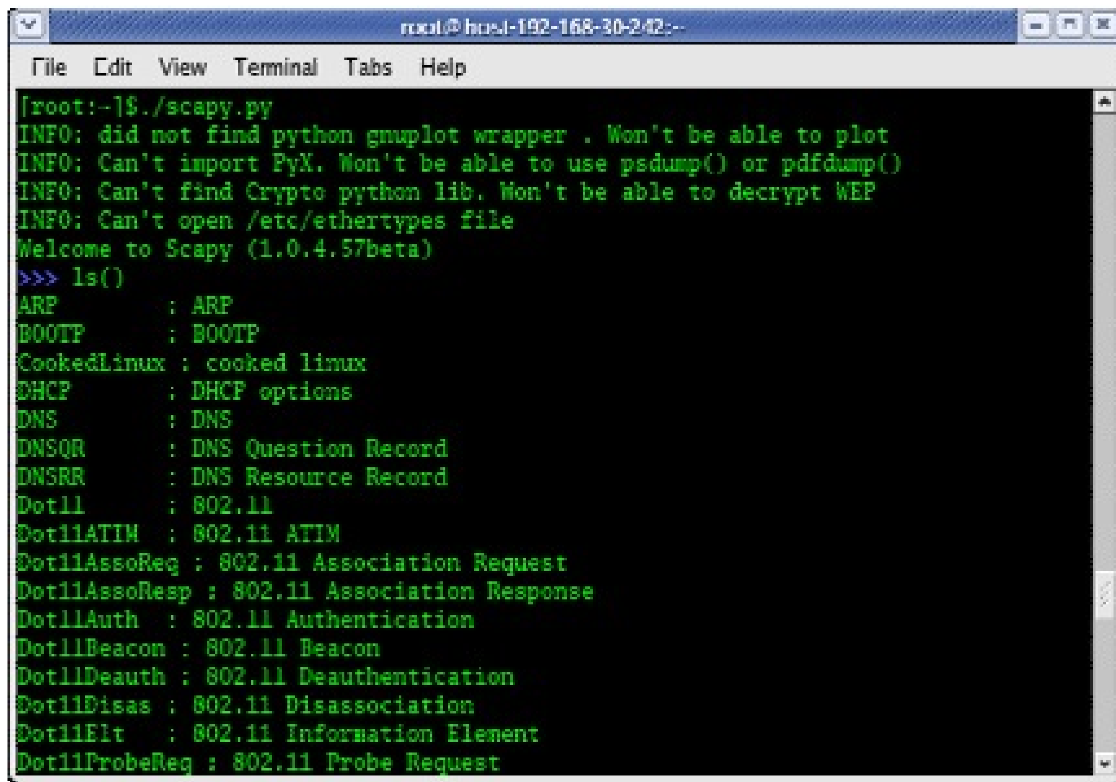

Figure 6. Loading scapy for packet manipulation.

The types of packets that will be most interesting with WiFi fuzzing will be the Dot11 series for packet construction. It is pretty easy to create a simple Python script that will inject anything you want. A test script to get started could be something as simple as what is shown below. All this little script will do is generate a simple frame and inject it. The script is as follows:

```
#!/bin/env python

import sys
from scapy import *

victim=sys.argv[1]
attacker=sys.argv[2]

conf.iface="ath0raw"

frame=Dot11(subtype=1, type=0, addr1=victim, addr2=attacker, addr3=attacker)
sendp(frame)
```

If you run Wireshark (formerly known as Ethereal) on the box and sniff `ath0raw` you will see the packets injected. The subtype of 1 sets the packet to be an association response. The command line used while running the test script is very simple:

```
./wifi.py 11:22:33:44:55:66 66:55:44:33:22:11
```

The result of several runs of the script can bee seen in Wireshark. Wireshark is useful in constructing and debugging a fuzzer, as it helps when fine tuning exactly what fields you want to exercise.

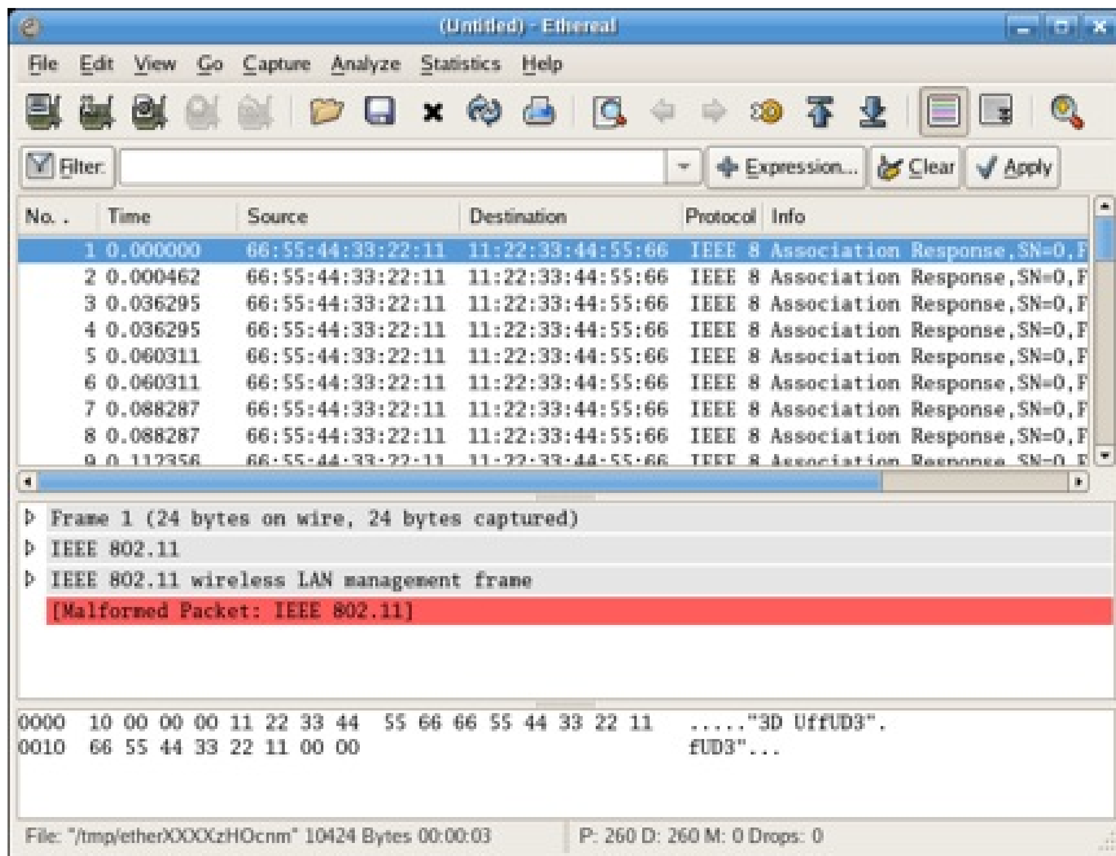

Figure 7. Using Wireshark after several iterations of our test script.

Looking at the other fields that scapy supports, it is now as easy as stacking them together. If you are unsure of what arguments are passed to a field, you can just do an ls() for it. For instance doing a ls(Dot11) will yield the following result:

```
>>> ls(Dot11)
subtype     : BitField               = (0)
type        : BitEnumField           = (0)
proto       : BitField               = (0)
FCfield     : FlagsField             = (0)
ID          : ShortField             = (0)
addr1       : MACField               = ('00:00:00:00:00:00')
addr2       : Dot11Addr2MACField     = ('00:00:00:00:00:00')
addr3       : Dot11Addr3MACField     = ('00:00:00:00:00:00')
SC          : Dot11SCField           = (0)
addr4       : Dot11Addr4MACField     = ('00:00:00:00:00:00')
>>>
```

In order to stack the fields, they are separated by a slash. You will create a general control frame followed by a field of a certain subtype. This would look as follows:

```
frame=Dot11()/Dot11AssoResp()
```

One of the nicest features about scapy is its fuzz function. You can wrap any of these elements in fuzz() and in a loop it will generate values for anything you didn't supply. You can see the results of this with a simple modification to the test script used earlier:

```
#!/bin/env python

import sys
from scapy import *

victim=sys.argv[1]
attacker=sys.argv[2]

conf.iface="ath0raw"

frame=fuzz(Dot11(addr1=victim, addr2=attacker, addr3=attacker))
sendp(frame, loop=1)
```

A run with the same command line options as previously used will produce a different packet for each injection. The only thing that will stay static across the packets is addr1, addr2, and addr3. This is a quick and simple way to generate fuzzing packets. There are a couple of different ways to go about fuzzing for best results.

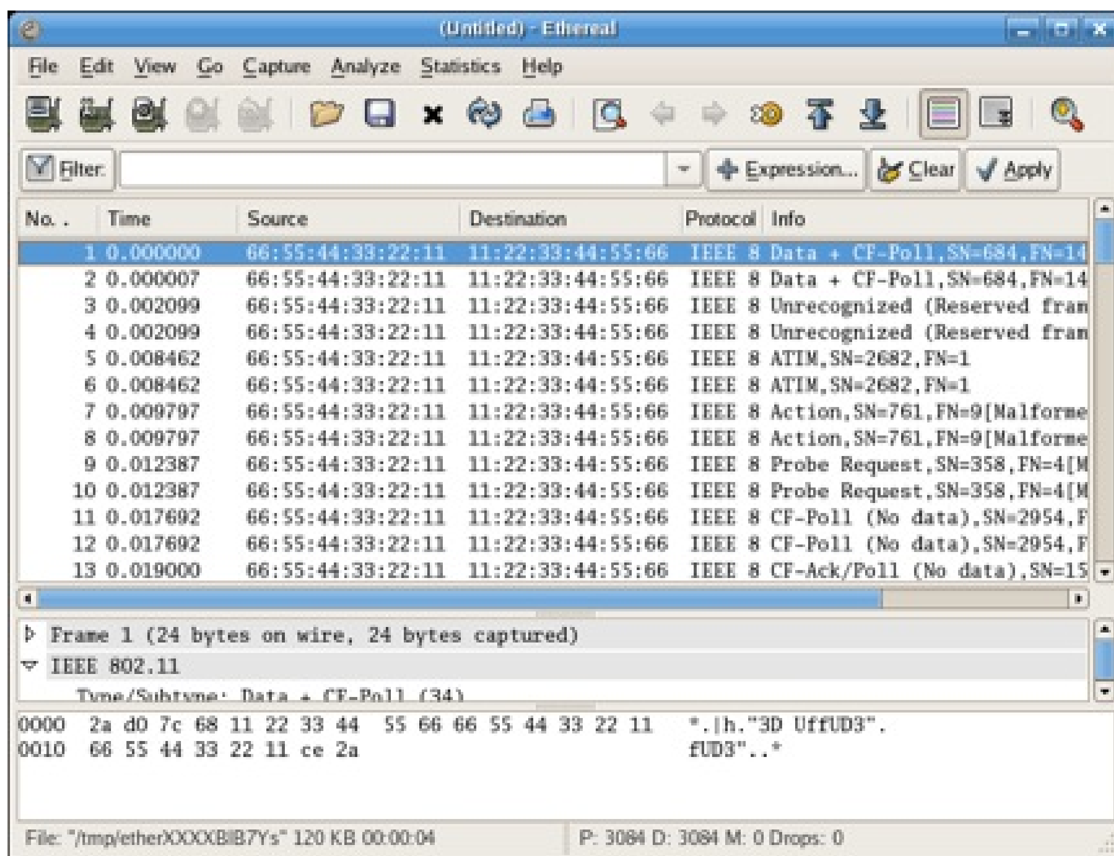

Figure 8. Using Wireshark to look at fuzzing packets.

A beginner's guide to effective fuzzing

Now that your environment is setup and working, the first step is to fuzz the target cards in different states. The state can be Associated, Unassociated, Ad-Hoc, scanning, and so on. The different states are important because many of the code paths you want to exercise can only be reached in certain states. Most drivers are intelligent enough to reject packets for a state they are not currently in. An example of this would be a laptop in an associated state with an access point and the fuzzer generating ad-hoc packets. In most drivers these packets will just be silently ignored.

The second thing to increase your chances of a successful fuzzing run is to use a kernel debugger. When a machine becomes unresponsive, you should be able to save the recent packets sent. Tracking down a vulnerability can be hard, especially since the crash can occur in a variety of different places. A minor overwrite in memory can lead to a mild memory corruption that may not be evident until a different driver attempts to access the same corrupted memory. Tracking down the exact cause of the vulnerability can be a difficult task.

On Windows, tools like Softice [ref 5] or Windbg [ref 6] can be used to set breakpoints on certain calls that would be beneficial in tracking down the corruption. On Apple's OS X it's a little more difficult to do this as kernel debugging requires two machines.

If Windows is your target, using the Windows DDK [ref 7] will be most helpful as a tool called DriverVerfier can help you quickly track down any memory corruption. Either way, it

will become very important that you become familiar with Windbg for its analysis of crash dumps. After a crash dump is loaded, the command '`!analyze â€"v`' is useful for generating a detailed analysis. The stack backtrace may not be that reliable as there is a good chance you have overwritten parts of it.

For best success, you should automate the process of status checking on the victim. For instance, on OS X there is an airport command which can be used to manipulate most of the wireless options without the need for going through the GUI. As malformed traffic is generated and spewed at the target, the machine may disassociate from the network and search for a better network. You can script the airport command to check the current state so that if it's not what is desired, it can be changed automatically. The â€"I argument passed to airport will give the current status. You can disassociate from a network, join a network or even force the airport to do any action repeatedly with the â€"r option. The same type of action can be done in Linux and in Windows with their respective tools. Normally a fuzzer run may take a long time to complete and it can be a horrible feeling when you discover that your target was in an incorrect state for the majority of that time.

Fuzzer runs are generally much more useful if they are directed toward areas a researcher thinks may be weak. This is best done through reverse engineering. Drivers generally aren't very large and will not take long to disassemble. The interesting thing about drivers is that you will find code normally extinct in other areas of the operating system that are still in abundance in drivers. This includes unchecked memcpy, string operations and loops that write to arrays without good terminating conditions. Looking through the disassembled code, you can generally tell what a code segment is responsible for by the debug messages they generate. If you find heavy use of memcpy, sprintf or strcpy, concentrate your fuzzer on those areas.

Going forward and next steps

Although this article was designed as a beginner's guide to auditing WiFi, these same types of techniques can be ported to other wireless protocols like Bluetooth. Although the fuzzers we used for our Black Hat presentation were written in C, it's hard to ignore how useful something like the Python based scapy is for quick and relatively easy fuzzing. In addition to the Dot11 packets it can generate, scapy can also generate L2 packets for Bluetooth use as well. The fuzz() function applies to these just like the Dot11 interface.

Bluetooth is a target-rich environment, even more so than WiFi. The range of Bluetooth is much less, but it's designed to be more open. Bluetooth supports features like SAR (Segmentation and Reassembly) and different kinds of encryption and compression, which are often ripe for auditors to pick apart. Simple things like oversized packets and requesting services, event though a device isn't paired, will cause certain Bluetooth stacks and mobile devices to crash.

The purpose of our recent talks and this introductory article was to show how easy building a wireless auditing platform is - and how these types of techniques can be incorporated into a QA testing plan. Fuzzers are useful for finding vulnerabilities, mostly the low-hanging fruit, but there is no replacement for time spent reverse engineering binaries. Taking the reverse engineering route, much more subtle bugs can be found that would take a fuzzer a very

long time to discover. A future article will look at fuzzing WiFi drivers in more detail.

References

[**ref 1**] "Device Drivers: Don't Build a House on a Shaky Foundation" by David Maynor and Jon Ellch. Black Hat 2006, Las Vegas. http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Cache.pdf (1.8 Mbyte PDF)

[**ref 2**] LORCON (Loss Of Radio CONnectivity) library. http://802.11ninja.net/code/lorcon-current.tgz

[**ref 3**] Madwifi multiband Atheros driver for Wifi. http://madwifi.org/

[**ref 4**] Scapy packet manipulation program. http://www.secdev.org/projects/scapy/

[**ref 5**] Softice application debugger, acquired by CompuWare. download from Softpedia.com

[**ref 6**] Microsoft Windbg application debugger. http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[**ref 7**] Windows DDK (Driver Development Kit). http://www.microsoft.com/whdc/devtools/ddk/default.mspx

About the author

David Maynor is a Senior Researcher with SecureWorks. His previous roles include reverse engineering and researching new evasion techniques with the ISS Xforce R&D team, application development at the Georgia Institute of Technology, as well as security consulting, penetration testing and contracting with a wide range of organizations.

Reprints or translations

Reprint or translation requests require prior approval from SecurityFocus.

© 2006 SecurityFocus

Comments?

Public comments for Infocus articles require technical merit to be published. General comments, article suggestions and feedback are encouraged but should be sent to the editorial team instead.

Privacy Statement
Copyright 2006, SecurityFocus