Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities[1]

Manuel Mendonça
Nuno Neves
Faculdade de Ciências
Universidade de Lisboa, Bloco C6,
Campo Grande 1749-016 Lisboa - Portugal
manuelmendonca@msn.com
nuno@di.fc.ul.pt

Abstract

Wireless LANs (WLAN) are becoming ubiquitous, as more and more consumer electronic equipments start to support them. This creates new security concerns, since hackers no longer need physical connection to the networks linking the devices, but only need to be in their proximity, to send malicious data to exploit some vulnerability. In this paper we present a fuzzer, called Wdev-Fuzzer, which can be utilized to locate security vulnerabilities in Wi-Fi device drivers. Our experiments with a Windows Mobile 5 device indicate that Wdev-Fuzzer can be quite effective in confirming known issues and discovering previously unknown problems.

1. Introduction

Wireless LANs give individuals the freedom to stay connected, while moving from one coverage area to another. They can be used to extend a wired infrastructure or to replace existing ones, saving costs not only due to the declining prices of the wireless components, but also because they require no (or simpler) data cable installations. Nowadays WLAN technologies are becoming ubiquitous, and most consumer electronic products (such as laptops, PADs, cellular phones, video game consoles, digital cameras, printers and video projectors) are equipped with them.

WLAN however introduce newer problems, since they weaken the security perimeter. In many places, like airports and shopping malls, there are dozens of rogue networks just waiting to entrap unsuspecting travelers to capture private information (e.g., usernames and passwords) [1]. Additionally, it is much easier to compromise WLAN equipments because hackers only need to be in proximity of the devices to perform the attack (physical connection to the network is no longer necessary).

Although some security failures are due to wrong system configurations, many result from the exploitation of implementation bugs in the software components. In this paper, we are particularly interested in locating this sort of bugs (or vulnerabilities) in device drivers (DD) of WLAN, to allow their removal. These DD are the entry point of any device, and therefore they are the first software to process the potentially malicious traffic coming from an attacker. Moreover, almost any vulnerability in these DD has a catastrophic impact since they run in the operating system (OS) kernel.

In general, DD provide an abstraction layer between the physical details of equipments and the kernel. Currently, they are becoming the most dynamic and largest part of an OS. Their design involves knowledge from several disparate areas, like OS internals, chipset details, and synchronization that are not simultaneously mastered by programmers or designers. Therefore, they are hard to implement and to maintain. Nowadays, even tough several programs exist to assist developers in increasing the quality and reliability of their driver implementations [6][7][8][9][10], many DD still end up being deployed with bugs.

Methods for discovering vulnerabilities in DD depend on the availability of the driver code. If the code is public, then source code auditing may lead to

[1] This work was partially supported by the EU through project IST-4-027513-STP (CRUTIAL) and NoE IST-4- 026764-NOE (RESIST), and by the FCT through the Large-Scale Informatic Systems Laboratory (LASIGE), project POSC/EIA/61643/2004 (AJECT) and the FCT Multiannual Funding Programme.

good results, as one can read and check for implementation flaws. However, in the majority of situations, the code is closed. In this case, black box testing may be performed, where the functional behavior of the unit under test (UUT) is verified (output results) against the input values that are provided. Reverse engineering may also be employed to discover vulnerabilities, but it is costly, time consuming and demands profound knowledge on system architecture and machine code.

Vulnerabilities can also be discovered by another black box testing methodology, sometimes called fuzzing [26][2][11]. Fuzzing consists on presenting malformed data to the interface of the software component and on observing the outcomes. This technique may require further refinements to catch more complex bugs, due to protocol specificities, but it can be very effective in locating several kinds of vulnerabilities (like TCP-IP stack problems and OS hangs).

In our work, we have designed a new fuzzer architecture that is able to build malformed packets and perform attacks against a target system, independently of its communication media. The current implementation of the architecture, called Wdev-Fuzzer, supports the Wi-Fi protocol. In the future we intend to extend the tool to other communication protocols, such as IrDA and Bluetooth.

The tool was utilized to study the behavior of a Wi-Fi device driver, of a handheld device running Windows Mobile 5. The tested scenarios simulate an attack against the Wi-Fi device, either when it is just looking for an Access Point (AP) to connect or is already connected. Experimental results demonstrated that in most cases Windows is capable of handling correctly the malicious packets. However, in one situation, a specific Beacon packet always caused a system hang. This implies that the DD has a critical vulnerability which was previously unknown. Wdev-Fuzzer was also successfully applied to uncover other potential problems. For example, it was used to reproduce denial of service attacks with Disassociation and Deauthentication frames. Additionally the tests revealed that there might be a problem in the implementation of the TCP-IP stack.

2. Wdev-Fuzzer

2.1 The Architecture of the Wdev-Fuzzer

The Wdev-Fuzzer is divided in 8 modules (see Figure 1). Message Specification is a text file that defines packets as a group of fields. Each packet field

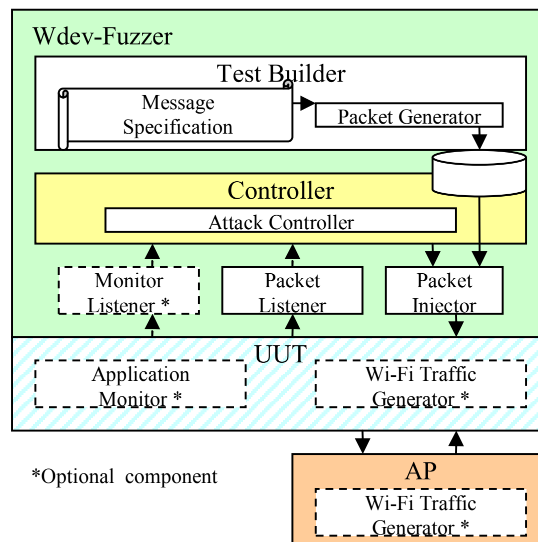

Figure 1. Wdev-Fuzzer block diagram

is also specified in the same file using basic data types that are intrinsic to the Packet Generator.

For each basic type there is a fuzz operator that assigns specific values according to some given rules. During the construction of the packets, the Packet Generator takes the packet description as input, and uses these operators to fill in the values of the fields.

The result is a ready-to-be-send potentially bogus packet. By extending the basic types and the fuzz operators, it is possible to build newer types and values, in order to meet specific protocol requirements.

The Attack Controller controls the activity of the Packet Injector. It decides which next packet (attack) should be transmitted, based on the feedback given by the Monitor Listener and Packet Listener, using a predetermined criteria. The Packet Listener receives and analyzes all responses that arrive from the UUT. The Monitor Application and corresponding Listener are optional components that exchange information about the state of the UUT. They find out if an attack was successful and contribute to the decision of which attack should be performed next. The Packet Injector sends the packets to the UUT.

The Traffic Generator is used to create and exchange good packets between the AP and the UUT. This way we can observe the system behavior when subject to an attack while correct data is being transmitted by a Real AP.

The basic architecture of Wdev-Fuzzer can be tailored to several communication protocols, still some changes will have to be performed. For example, a new Message Specification has to be carried out and the Packet Injector and Packet Listener implementations

have to be updated to use the specific functions for sending and receiving raw packets from the media.

2.2 Using Wdev-Fuzzer in 802.11

The IEEE 802.11 architecture consists of several interacting components to provide a WLAN that supports station mobility transparently to upper layers. The basic service set (BSS) is the fundamental building block of an IEEE 802.11 LAN. The BSS coverage area is where the member stations (STA) of the BSS may remain in communication. If a STA moves out of its BSS, it can no longer directly communicate with the other members.

The independent BSS (IBSS) is the most basic type of a Wi-Fi LAN, and consists of only two STA that are able to exchange data directly with each other. Since this type of network is often formed without pre-planning it is usually referred to as an ad-hoc network.

A BSS, instead of operating independently, may also be part of an extended form of network that is built with multiple BSSs and is interconnected by a distribution system (DS). In this setting, an AP gives access to the DS by providing DS services in addition to act as a STA.

Figure 2 shows the Medium Access Control (MAC) message frame format for the 802.11 protocol. These frames may be composed by fixed length (FL) and Tag Length Value (TLV) field types. To facilitate message parsing, when FL and TLV fields appear in the same message, FL fields always come first. A FL field appears at a fixed location relative to the beginning of the frame and it always has the same length. A TLV field has three elements, a Tag which uniquely identifies the field, a size element which determines the length of the data and the data itself.

The MAC frame types that may be exchanged between a pair of STAs depend on their state. The state of the sending STA, given by Figure 3, is defined with respect to the intended receiving STA. The allowed frame types that can be transmitted in a given state are grouped into classes. In State 1, only Class 1 frames are allowed. In State 2, either Class 1 or Class 2 frames are acceptable. In State 3, all frames are permitted (Classes 1, 2, and 3). The frame classes are shown in Table 1.

The 802.11 standard is very large and reviews made by the ruling committee can take a while to be ready. To speed up the decision process, as well as to ratify important subsets of standards, nearly almost wireless equipment manufacturer joined the Wi-Fi Alliance. This group is dedicated to manage the Wi-Fi specification, a subset of the 802.11 standard, and defines the "right thing" to do if any ambiguity in the

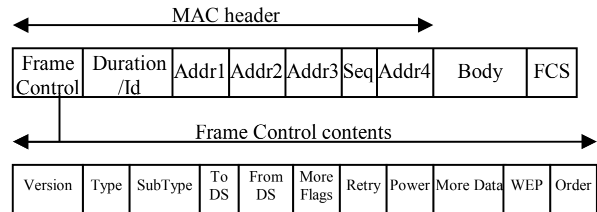

Figure 2. Generic Wi-Fi MAC frame format

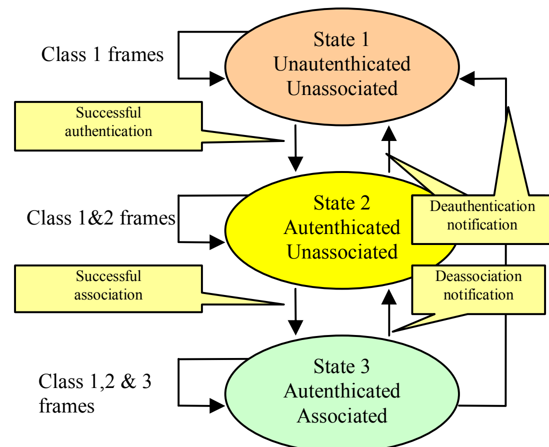

Figure 3. Relationship between messages and services

Table 1. Tested Wi-Fi frames

Frame	Type	Sub Type	To AP	From AP	Class
Association Request	Mgt	0	X	-	2
Association Response	Mgt	1	-	X	2
Reassociation Request	Mgt	2	X	-	2
Reassociation Response	Mgt	3	-	X	2
Probe Request	Mgt	4	X	-	1
Probe Response	Mgt	5	-	X	1
Beacon	Mgt	8	-	X	1
Disassociation	Mgt	10	X	X	2
Authentication	Mgt	11	X	X	1
Deauthentication	Mgt	12	X	X	1,3
Power Save	Ctrl	10	X	-	3
Request To Send	Ctrl	11	X	-	1
Clear to Send	Ctrl	12	-	X	1
Acknowledgment (Ack)	Ctrl	13	X	X	1
Contention Free (CF) End	Ctrl	14	-	X	1
CF-End + CF-Ack	Ctrl	15	-	X	1
Data	Data	0	X	X	1,3

(Mgt – Management, Ctrl – Control)

802.11 standard arises. It is committed to guarantee the interoperability among vendors, assuring that all products with the "Wi-Fi certified logo" work together.

In this work, we utilize the Wdev-Fuzzer to evaluate the Wi-Fi implementation of a Windows Mobile 5 handheld device. Since these equipments are mostly used as a STA rather than as an AP, the device will be configured as an STA. The evaluation of an AP is left

out for future work. Additionally, we will not use the IBSS configuration because handheld devices are many times operated in a connected BSS. In the tested scenarios, the Wdev-Fuzzer is going to simulate a malicious AP that sends potentially erroneous frames to a UUT.

2.3 Tested Faulty Values

Table 2 displays the fuzz operators that are applied to each field type, to build Wi-Fi frames in the experiments. The 'X' character indicates that the operator was applied to the field and the '-' the opposite.

The operator "Not present" omits an element from the frame. The "Repeated" operator produces multiple occurrences of the same field in the frame. The operators "All bits Zero" and "All bits One" are self explanatory. The "MIN" and "MAX" operators produce the minimum and maximum values that a field might contain, as stated in the 802.11 specification. Often, the "All bits Zero" and "MIN" operators produce equal values, whenever the minimum value is zero. The same applies for operators "MAX" and "All bits One". In these cases, the "MIN" or "MAX" operators are not utilized, since they create test results equivalent to the "All bits Zero" and "All bits One" (respectively).

The "Random" operator generates random values that are between the values produced by the "MIN" and "MAX" operators. At last, the "Specific Value" operator places a pre-defined value in a field. This operator is used for example to force certain frames to have UUT's MAC address.

2.4 Tested Scenarios

At first we considered testing the UUT in all 3 states represented in Figure 3. However, since in real situations State 2 is only available for shorts periods of time, only States 1 and 3 were considered.

Tests were carried out in 3 different scenarios (A, B and C). In scenario A, the UUT was in State 1, meaning that it was not associated or authenticated with any AP. In scenario B, the UUT was in State 3, linked to a Real AP using no authentication. At last, in scenario C, the UUT was also at State 3 but using authentication. In scenarios B and C, the Traffic Generator forced the exchange of data packets between the UUT and the Real AP to stress the communication stack by opening a TCP-IP socket and exchange packets between the UUT and the Real AP.

Table 2. Tested Faulty Values

Fuzz Operator	Fixed Length Field	Tag Length Value Field
Not Present	-	X
Repeated	-	X
All bits Zero	X	X
MIN-1	X	X
MIN	X	X
MIN+1	X	X
Random	X	X
Specific Value	X	X
MAX-1	X	X
MAX	X	X
MAX+1	X	X
All bits One	X	X

Table 3. Expected failure modes

ID	Description
F1	No problems were detected in the system execution.
F2	Packet Listener detects invalid frame.
F3	UUT was disassociated.
F4	UUT was de-authenticated.
F5	Monitor hangs.
F6	OS hangs.
F7	The system crashes and then reboots.

Table 4. Detailed F1 failure mode

ID	Description
F1A	Device provides correct information about AP (either detecting it or not)
F1B	Device does not detect the AP but it should.
F1C	Device detects the AP but it should not.

2.5 Expected Failure Modes

The Packet Generator uses the Message Specification and the fuzz operators to build Wi-Fi frames. Depending on the values produced, the UUT is going to receive good and bad Wi-Fi frames, which may be handled correctly or may lead to some failure. Table 3 summarizes the expected failure modes of the UUT when it receives Wi-Fi frames. It was elaborated after some preliminary experiments and also based on information provided in the literature [15][16].

F1 represents the case where the system appears to continue to work without any problems. However, in general, it does not mean that the injected fault was handled correctly. Whenever a test uses Beacon or Probe frames, the UUT Monitor returns some feedback to the Controller, saying which AP have been detected.

In these cases, we are able to further extend F1 in three other categories, as represented in Table 4.

F1A represents the scenario when the Monitor correctly reports the information about the AP, either because it was detected (the packet was well-formed) or because it was not detected (the packet was incorrectly formed, and therefore, the UUT discarded it and the report indicates no AP). The F1B value applies to the cases where the Monitor does not detect the AP but it should, and F1C corresponds to the cases where the AP is detected but it should not.

When the UUT is at State 3, the F3 failure mode means that the device became disassociated from the AP, as a result of some attack. Likewise, the F4 mode indicates that the attack successfully deauthenticated the UUT from the AP.

F5 failure mode signals that the Monitor Application hangs as a consequence of an attack, denoting that some problem with the DD has propagated to the application. Whenever the OS hangs, the F6 mode is used. The F7 failure mode happens if the system crashes and then reboots.

3. The Testing Infrastructure

In the Windows OS family, the Network Driver Interface Specification (NDIS) defines a standard Application Program Interface (API) for Network Interface Cards (NIC's). The details of a NIC hardware implementation can be wrapped by a Media Access Controller (MAC) device driver, in such a way that all NIC's for the same media (e.g., Ethernet) are accessed using a common API. Applications interact with NIC's through a stack of device drivers, where each driver adds functionality to the entire communication infrastructure.

Probably, the main difficulty in building a Wi-Fi test infrastructure is the implementation of the operations for injecting and capturing the Wi-Fi raw frames. Our first attempt to address the problem utilized a filter DD that was placed in the lower parts of the driver stack, hoping to intercept packets sent and received by each NIC (as well as control instructions given by the OS to the DD). Windows, however, implements the Wi-Fi protocol in the MAC DD, which emulates the Ethernet protocol to the drivers above it. Therefore, our DD was only able to capture Ethernet frames and not Wi-Fi raw frames.

Still there are other possible ways for capturing Wi-Fi frames in Windows, neither of them very easy to achieve. One approach is using an internal interface to the MAC DD, but to the best of our knowledge no vendor provides it. Another consists in developing

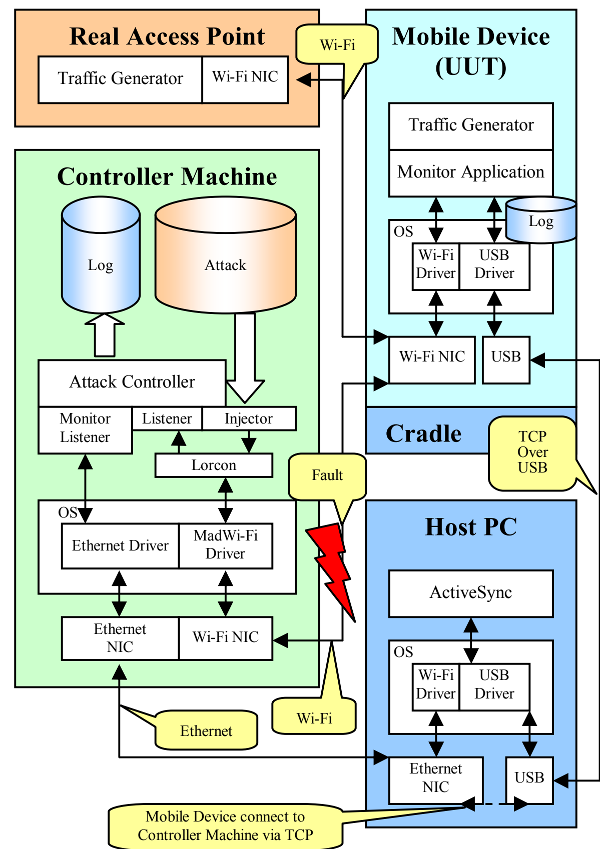

Figure 4. The process of fuzzing Wi-Fi frames

our own MAC DD, but this would require a direct interaction with the NIC and complete knowledge of its specification (something that usually is not available). A commercial solution based on this idea is Airpcap [5], which uses a proprietary MAC DD and their own capture hardware.

In the end, we decided to build a heterogeneous testing infrastructure, since in Linux there are several cards and open drivers that support Wi-Fi frame injection and capture (although not every NIC can be used due to hardware limitations). One simple way to find them is to search in the Internet for Wi-Fi sniffers and look for compatible NICs. Figure 4 displays the current testing infrastructure that is composed by 4 components: the Controller Machine, the Mobile Device (UUT), the Host PC and the Real Access Point.

3.1 Controller Machine and UUT

The Controller Machine generates the Wi-Fi packets containing malicious data (e.g., out-of-bound values, repeated tags) and sends them through the Wi-Fi

interface to the UUT. Each packet is sent several times to assure that the UUT is able to receive it.

This element also monitors the outcomes of the tests, and saves the collected data in the disk for future analysis. Currently, the Controller is installed in a Linux OS machine, with the MadWi-Fi driver [3] for wireless LAN chipsets from Atheros. The Packet Injector uses a modified version of Lorcon [4] as a generic library for injecting Wi-Fi frames. The Monitor Listener receives any incoming frames from the Monitor installed in the UUT and forwards this information to the Attack Controller to synchronize the next attack. The Packet Listener informs the Attack Controller of each incoming packet sent by the UUT. These packets have to be carefully examined to detect any unexpected behavior.

The UUT is the target Wi-Fi device of the experiments. It runs a Monitor Application that regularly connects to the Monitor Listener of the Controller, informing the current list of detected AP and the status of any existing connection. This data is especially useful when testing Beacon and Probe frames, as the detection of the AP is crucial to determine the correction of the error handling mechanisms.

3.2 Host PC and Real AP

The UUT is physically attached to the Host PC through an USB port. This way, the Monitor Application can reach the Attack Controller through an out of band link, leaving the Wi-Fi medium free for the experiments. The Host PC runs Windows XP and Microsoft's ActiveSync, allowing the communication between the UUT and the Host PC with TCP over USB, which is then followed by TCP over Ethernet in the connection between the Host PC and the Controller Machine.

To keep the complexity of the code of the Controller manageable, a Real AP is utilized to take the UUT through the various states of the Wi-Fi protocol. This way, specific frames can be injected in every state. The Real AP was implemented in Windows XP using an off-the-shelf AP application.

4. Experimental Results

This section presents the results of the various experiments carried out with the Wdev-Fuzzer in an 802.11b network. The test bed was composed by a Controller Machine implemented in a Dell Optiplex 170L Pentium IV computer, installed with Fedora Core 6. It used a NetGear WPN311 wireless PCI card and the built-in Ethernet card as communication means. The UUT was an HP iPAQ hw6915 PDA running Windows Mobile 5 and equipped with a built-in Texas Instruments Wi-Fi chip. The Host PC machine was a HighScreen Pentium IV computer with Windows XP Professional Edition. The UUT was attached to an USB port on the Host and uses ActiveSync 4.1 build 4841 to establish the connection. This machine was also equipped with an Ethernet card, which was connected to the Controller Machine with a 100Mbits link. It also hosts the Real AP using a GigaByte AirCruiser GN-WP01GS wireless PCI card and the companion AP application. The UUT was attached to an USB port on the Host PC and placed at about 2m distance from the Controller Machine and the Real AP.

4.1 Observed Failure Modes

The results of the test campaigns are displayed in Table 5 and Table 6. A total of 89489 attacks were carried out for each of the three scenarios. The tables only show the outcomes for frames that flow from the AP to the UUT (see Table 1), since frames on the other direction never caused any problems (i.e., the failure mode was always of type F1). The first column of the tables shows the field type being tested, and the second column displays how many different values were tried. The following columns display the results obtained for the various different frames. The '-' character is used to indicate that the corresponding field does not belong to the frame being tested, otherwise it is filled with the code of the observed failure mode (see Table 3).

Since in most cases the result was F1, to make the table reading simpler, the number of times that it occurs is omitted (it is equal to number of tried values displayed in the second column). For failure modes different than F1, the table presents in the cell the number of tests that caused a problem.

4.1.1 Failure Modes in Scenario A

The UUT is in State 1 in the test campaign of scenario A. The UUT is placed in this state by powering on the Wi-Fi component of the device and by making sure that no association exists with any STA or AP. The test results for this scenario are displayed in Table 5. It shows that in general the UUT was able to handle correctly the malicious frames. Nevertheless, some interesting outcomes were observed for certain specific scenarios, which are summarized in the following points.

Table 5. Observed failure modes in scenario A

Field	#Tests & Results	CTS	Ack	CF-End	CF-End CF-Ack	Data	Assoc. Resp	Reass. Resp	Prob. Resp	Beacon	Disass.	Auth.	Deauth.
Protocol* Version	4	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
To/From* DS	4	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
More Flags*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
Retry*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
Power* Management	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
More* Data	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
WEP*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
Order*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
Duration	3500	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
RA/Addr1	8	F1	F1	F1	F1	F1	-	-	-	-	-	-	-
TA/Addr2	8	-	-	-	-	F1	-	-	-	-	-	-	-
DA	8	-	-	-	-	-	F1	F1	7x F1C	7x F1C	F1	F1	F1
SA	8	-	-	-	-	-	F1	F1	F1A	F1A	F1	F1	F1
AID	15	-	-	-	-	-	F1	F1	-	-	-	-	-
BSS ID	8	-	-	F1	F1	-	F1	F1	F1	F1	F1	F1	F1
Addr3	8	-	-	-	-	F1	-	-	-	-	-	-	-
Sequence Control	10	-	-	-	-	F1	F1	F1	F1	F1	F1	F1	F1
Addr4	7	-	-	-	-	F1	-	-	-	-	-	-	-
Frame Body	7	-	-	-	-	F1	-	-	-	-	-	-	-
TimeStamp	6	-	-	-	-	-	-	-	F1A	F1A	-	F1	-
Beacon** Interval	2700	-	-	-	-	-	-	-	F1A	F1A	-	F1	-
Capabilities**	2050	-	-	-	-	-	F1	F1	F1A	F1A	-	F1	-
SSID**	1275	-	-	-	-	-	F1	F1	32x F1B	32x F1B	F1	F1	F1
Supported** Rates	256	-	-	-	-	-	F1	F1	F1A	F1A	F1	F1	F1
FH** Parameter	256	-	-	-	-	-	F1	F1	F1A	F1A	F1	F1	F1
DS** Parameter	256	-	-	-	-	-	F1	F1	F1A	F1A	F1	F1	F1
CF** Parameter	256	-	-	-	-	-	F1	F1	F1A	F1A	F1	F1	F1
IBSS** Parameter	256	-	-	-	-	-	F1	F1	F1A	F1A	F1	F1	F1
TIM**	256	-	-	-	-	-	F1	F1	F1	1x F6	F1	F1	F1
Reason Code	15	-	-	-	-	-	-	-	-	-	F1	-	F1
Status Code	5	-	-	-	-	-	F1	F1	-	-	-	F1	-
Auth. Algorithm Nbr	5	-	-	-	-	-	-	-	-	-	-	F1	-
Auth. Transaction Nbr	5	-	-	-	-	-	-	-	-	-	-	F1	-
Other TLV**	1255	-	-	-	-	-	F1	F1	F1	F1	F1	F1	F1

* Frame Control, **Tag Length Value

Table 6. Observed failure modes, scenario B and C

Field	#Tests & Results	CTS	Ack	CF-End	CF-End CF-Ack	Data	Assoc. Resp	Reass. Resp	Prob. Resp	Beacon	Disass.	Auth.	Deauth.
Protocol* Version	4	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
To/From* DS	4	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	3x F3	F1	3x F4
More Flags*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
Retry*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
Power* Management	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
More* Data	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
WEP*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
Order*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
Duration	3500	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	3500x F3	F1	3500x F4
RA/Addr1	8	F1	F1	F1	F1	F1	-	-	-	-	-	-	-
TA/Addr2	8	-	-	-	-	F1	-	-	-	-	-	-	-
DA	8	-	-	-	-	-	F1	F1	7x F1C	7x F1C	2x F3	F1	2x F4
SA	8	-	-	-	-	-	F1	F1	F1A	F1A	1x F3	F1	1x F4
AID	15	-	-	-	-	-	F1	F1	-	-	-	-	-
BSS ID	8	-	-	F1	F1	-	F1	F1	F1	F1	1x F3	F1	1x F4
Addr3	8	-	-	-	-	F1	-	-	-	-	-	-	-
Sequence Control	10	-	-	-	-	F1	F1	F1	F1	F1	10x F3	F1	10x F4
Addr4	7	-	-	-	-	F1	-	-	-	-	-	-	-
Frame Body	7	-	-	-	-	F1	-	-	-	-	-	-	-
TimeStamp	6	-	-	-	-	-	-	-	F1A	F1A	-	F1	-
Beacon Interval	2700	-	-	-	-	-	-	-	F1A	F1A	-	F1	-
Capabilities**	2050	-	-	-	-	-	F1	F1	F1A	F1A	-	F1	-
SSID**	1275	-	-	-	-	-	F1	F1	32x F1B	32x F1B	1275x F3	F1	1275x F4
Supported** Rates	256	-	-	-	-	-	F1	F1	F1A	F1A	256x F3	F1	256x F4
FH** Parameter	256	-	-	-	-	-	F1	F1	F1A	F1A	256x F3	F1	256x F4
DS** Parameter	256	-	-	-	-	-	F1	F1	F1A	F1A	256x F3	F1	256x F4
CF** Parameter	256	-	-	-	-	-	F1	F1	F1A	F1A	256x F3	F1	256x F4
IBSS** Parameter	256	-	-	-	-	-	F1	F1	F1A	F1A	256x F3	F1	256x F4
TIM**	256	-	-	-	-	-	F1	F1	F1A	1x F6	256x F3	F1	256x F4
Reason Code	15	-	-	-	-	-	-	-	-	-	15x F3	-	15x F4
Status Code	5	-	-	-	-	-	F1	F1	-	-	-	F1	-
Auth. Algorithm Nbr	5	-	-	-	-	-	-	-	-	-	-	F1	-
Auth. Transaction Nbr	5	-	-	-	-	-	-	-	-	-	-	F1	-
Other TLV**	1255	-	-	-	-	-	F1	F1	F1	F1	1255x F3	F1	1255x F4

* Frame Control, **Tag Length Value

Since Beacon frames are directed to everybody in the coverage area, APs should announce themselves using the broadcast MAC address (FF:FF:FF:FF:FF:FF) as the Destination Address. Windows Mobile, however, reports a new AP when the Destination Address uses a distinct MAC address (see row DA). This occurs even when the Destination Address is different from the MAC address of the UUT. This behavior is an implementation issue and doesn't seam to be a problem.

SSID is the identifier of the AP, and it has a maximum size of 32 characters. The experiments show that the UUT does not report an existing AP if the SSID field has '0x00' as one of the ASCII characters of the identifier (see row SSID). The same behavior was also seen when we run an equivalent test with another Windows Mobile equipment, which gives evidence that this problem may extend to several other implementations. From a security perspective, this behavior is undesirable since it allows the creation of networks which are hidden from certain devices (e.g., a group of hackers could keep a network secret if they found out that the security officers use a Windows Mobile-based solution for diagnosing Wi-Fi networks).

When multiple SSID fields are sent in a given frame, the UUT assumes the last value as the correct one. If other vendors take a different view, and choose for instance the first SSID, then this could lead to incompatibility problems. The 802.11 specification does not address this particular issue.

Whenever the UUT receives a Beacon frame with a TLV field with TAG = 5 (Traffic Information Map – TIM), Length = 255 and Value = 0xFF, the OS hangs at the first user interaction with the device (see F6 value in row TIM). The same kind of failure also occurred when the UUT was in States 2 and 3, as shown in Table 6. When a similar test was made with another Windows Mobile equipment, everything went fine and no hangs were felt. This probably means that the flaw is in HP iPAQ device driver. Even so, the vulnerability is critical from an availability standpoint because exploitation is simple (e.g., since Beacon frames are processed in all states, a hacker would only need to walk around with a malicious AP to hang all vulnerable devices in a surrounding area).

The Probe Response failure modes were identical to the Beacon frame, with the exception of the TIM field where no OS hangs were seen.

4.1.2 Failure Modes in Scenario B

To perform the experiments corresponding to the scenario B, the UUT was associated and authenticated to the Real AP using no encryption protocol. The results for the Beacon and Probe Response frames are equivalent to those obtained in scenario A, which is not surprising, as the process of detecting APs while connected to another AP remains the same.

Fuzzing Disassociation and Deauthentication frames confirmed a known problem with the Wi-Fi protocol. Since the various fields of the frame are not cryptographically protected with some authentication data (e.g., a message authentication code), a rogue AP can transmit Disassociation and Deauthentication frames and cause the Wi-Fi communication to be disrupted (i.e., the Wi-Fi protocol is vulnerable to a Denial of Service (DoS) attack). This can happen if the Destination Address (DA) is equal to the address of the associated STA or the broadcast address. Nevertheless, we found out that several checks are made before accepting the frames, making the attack harder to execute. Several flags of the frame control part of the packet are verified (To/From DS, More Flags, Retry, Power Management, More Data, WEP and Order), reducing significantly the combinations that break the communication.

We also discovered that, whenever the UUT became disassociated and got associated after terminating the attack, the Traffic Generator could not recover the TCP-IP communication. This aspect reveals that some implementation problems may exist in the TCP-IP stack. Contrarily, whenever the UUT become deauthenticate and got authenticated at the end of the attack, the Traffic Generator always recovered the TCP-IP communication. This shows that the DoS attacks performed with Dissassociation frames can be more harmful than the ones made with Deauthentication frames.

4.1.3 Failure Modes in Scenario C

In scenario C, the test campaign was performed with the UUT associated and authenticated to the Real AP using shared key mode encryption protocol. The results observed in scenario C were equal to the ones obtained in the scenario B.

5. Related Work

Fault injection methods and tools can inject hardware or software faults in a target system under test (see for example, [12][13][19][25][27][30]). By forcing and reproducing the occurrence of irregular and unusual events, they can for instance evaluate the target system's ability to cope with them. The mimicked

faults were in most cases relatively simple, such as pin-level faults or single bit-flips in memory, registers, or instructions. Consequently, these techniques were mostly used for activities such as hardware validation or for the verification of fault handling mechanisms, and not for the discovery of security vulnerabilities.

Robustness testing can be applied to characterize the behavior of software components when they face exceptional inputs or stressful environmental conditions. Most of the robustness tools have targeted general propose OS, by supplying erroneous inputs to the functions that constitute the various application interfaces. For instance, the Ballista tool has assessed several OS that implement the POSIX standard [20]. Similarly, Shelton et al. have made a comparative study of six variants of Windows [28]. Other example studies with these tools include real time microkernels [14] and middleware support systems like CORBA [22][24]. More recently, this technique has been applied at the OS device driver interface [15][16][17][23]. Robustness testing has however been mainly applied to the internal interfaces of systems, which can not be directly exploited by an external adversary. Therefore, the discovered problems in most cases do not correspond to security vulnerabilities.

Fuzzing is a testing technique that generates invalid data and passes it to a target application for processing, and then observes the application to see if it fails while consuming the data [11]. A failure indicates the presence of some vulnerability, which can potentially be exploited by some adversary. Fuzz was one of the first projects to explore these ideas, and it was designed to test UNIX commands (and was later applied to other OS) [26]. It generates large sequences of random characters which were used as command-line arguments of programs. Many programs failed to process the illegal arguments and crashed. In the recent years, fuzzers have evolved into more intelligent and less random tools, capable of testing different kinds of software components (see for example, [18][21][29][31]). Fuzzing Wi-Fi is not new and reports of hacking and security problems were previously reported (see [32][33][34]).

In this paper, we present the design of a tool that aims at finding vulnerabilities in Wi-Fi driver implementations. The tool works in a completely automatic way, and is able to test several IEEE 802.11 commands in the various stages of the protocol execution. An experimental evaluation of the tool was performed with Windows Mobile, which has been extensively tested through the years. Nevertheless, as an interesting output of our investigation, we were able to uncover some previously unknown problems.

6. Conclusions

This paper presents the Wdev-Fuzzer tool, a fuzzer that targets device drivers of communication protocols. The proposed architecture is quite generic, allowing a detailed description of the protocol's messages. Therefore, the generated attacks are very effective at discovering new vulnerabilities and at verifying known issues. Additionally, the tool can also help to perform some of the tasks of conformance testing, by detecting misbehaviors of the device driver's implementation with respect to the specification of the protocols.

The current version of the tool was utilized to evaluate a Wi-Fi device driver of a handheld device running Windows Mobile 5. The results demonstrated that in most cases, Windows was able to handle correctly the malicious frames. They also showed that Wdev-Fuzzer can be successfully applied to reproduce denial of service attacks using Disassociation and Deauthentication frames. The tool revealed that there might be a problem in the implementation of the TCP-IP stack, uncovered by the use of disassociation frames when the UUT was associated and authenticated with an AP. Finally, it discovered a previously unknown vulnerability that causes OS hangs, using the TIM element in the Beacon frame.

7. References

[1] Sniffing tools, June 2007, http://insecure.org/

[2] Fuzzers, June 2007 http://www.infosecinstitute.com /blog/2005/12/fuzzers-ultimate-list.html

[3] Madwifi driver, June 2007, www.madwifi.org

[4] Lorcon project, June 2007 http://802.11ninja.net/lorcon

[5] Airpcap, June 2007, http://www.cacetech.com/products/ airpcap.htm

[6] Microsoft Corporation, "Introducing Static Driver Verifier", May 2006.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, On u-kernel construction, Proceedings of the Symposium on Operating Systems Principles, December 1995, pp. 237–250.

[8] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, The Flux OSKit: a substrate for OS language and resource management, Proceedings of the Symposium on Operating Systems Principles, October 1997, pp. 38–51.

[9] M. Swift, B. Bershad, and H. Levy, Improving the reliability of commodity operating systems, Proceedings of the Symposium on Operating Systems Principles, October 2003, pp. 207–222.

[10] M. Young, M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and A. Tevanian, Mach: A new kernel foundation for UNIX development, Proceedings of the Summer USENIX Conference, June 1986, pp. 93–113.

[11] P. Oehlert, Violating Assumptions with Fuzzing, IEEE Security & Privacy, pp. 58-62, March/April 2005.

[12] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, GOOFI: Generic Object-oriented Fault Injection Tool, Proceedings of the International Conference on Dependable Systems and Networks, pp. 83–88, July 2001.

[13] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, Fault Injection and Dependability Evaluation of Fault-tolerant Systems. IEEE Transactions on Computers, 42(8):913.923, August 1993.

[14] J. Arlat, J.-C. Fabre, M. Rodríguez and F. Salles, Dependability of COTS Microkernel-Based Systems, IEEE Transactions on Computers, vol. 51, no. 2, pp. 138-163, 2002.

[15] A. Albinet, J. Arlat, and J.-C. Fabre, Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel, Proceedings of the International Conference on Dependable Systems and Networks, June 2004

[16] J. Durães and H. Madeira, Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation, Proceedings of the Pacific Rim International Symposium on Dependable Computing, pp. 201-209, December 2002.

[17] A. Johansson, and N. Suri, Error Propagation Profiling of Operating Systems, Proceedings of the International Conference on Dependable Systems and Networks, June 2005.

[18] T. Biege, Radius Fuzzer, September 2005. http://www.suse.de/ thomas/index.html.

[19] J. Carreira, H. Madeira, and J. G. Silva, Xception: Software Fault Injection and Monitoring in Processor Functional Units, Proceedings of the International Working Conference on Dependable Computing for Critical Applications, pp. 135–149, January 1995.

[20] P. Koopman, J. DeVale, The Exception Handling Effectiveness of POSIX Operating Systems, IEEE Transactions on Software Engineering, vol. 26, no. 9, pp. 837-848, September 2000.

[21] A. Greene, SPIKEfile, September 2005. http://labs.idefense.com/labs-software.php?show=14.

[22] E. Marsden, J.-C. Fabre and J. Arlat, Dependability of CORBA Systems: Service Characterization by Fault Injection, Proceedings of the 21st International Symposium on Reliable Distributed Systems, pp. 276-285, June 2002.

[23] M. Mendonça, N. Neves, Robustness Testing of the Windows DDK, Proceedings of the International Conference on Dependable Systems and Networks, June 2007

[24] J. Pan, P. J. Koopman, D. P. Siewiorek, Y. Huang, R. Gruber and M. L. Jiang, Robustness Testing and Hardening of CORBA ORB Implementations, Proceedings of the International Conference on Dependable Systems and Networks, pp. 141-150, June 2001.

[25] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, Fault Injection Techniques and Tools, IEEE Computer, 30(4):75.82, April 1997.

[26] B. P. Miller, L. Fredriksen, and B. So, An empirical study of the reliability of UNIX utilities, Communications of the ACM, 33(12):32–44, 1990.

[27] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A tool for the Validation of System Dependability Properties, Proceedings of the International Symposium on Fault-Tolerant Computing, pp. 336-344, June 1992.

[28] C. Shelton, P. Koopman and K. DeVale, Robustness Testing of the Microsoft Win32 API, Proceedings of the International Conference on Dependable Systems and Networks, pp. 261-270, June 2000.

[29] M. Sutton, FileFuzz, September 2005. http://labs.idefense.com/labs-software.php?show=3.

[30] T. Tsai and R. Iyer, Measuring Fault Tolerance with the FTAPE Fault Injection Tool, International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, volume 977 of Lecture Notes in Computer Science, pp. 26–40. September 1995.

[31] J. Röning, et al., PROTOS – Security Testing of Protocol Implementations, Computer Engineering Laboratory, University of Oulu, 1999–2003. http://www.ee.oulu.fi/research/ouspg/protos/.

[32] D. Maynor, J. Ellch, Researchers hack Wi-Fi driver to breach laptop http://www.infoworld.com/article /06/06/21/ 79536_HNwifibreach_1.html, June, 2006.

[33] D. Maynor, Beginner's Guide to Wireless Auditing, http://www.securityfocus.com/infocus/1877, September 2006

[34] R. Naraine, Wi-Fi Exploits Coming to Metasploit, http://www.eweek.com/c/a/Security/WiFi-Exploits-Coming-to-Metasploit/, October 2006.

[35] W. Gu, Z. Kalbarczyk, R. Iyer, Z. Yang, Characterization of Linux Kernel Behavior under Errors, Proceedings of the International Conference on Dependable Systems and Networks, June 2003