BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

BLACK-BOX ANALYSIS OF WI-FI STACKS SECURITY
BLACK-BOX ANALÝZA ZABEZPEČENÍ WI-FI

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. ADAM VENGER
SUPERVISOR VEDOUCÍ PRÁCE	Mgr. KAMIL MALINKA, Ph.D.

BRNO 2021

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

23755

Student:	**Venger Adam, Bc.**
Programme:	Information Technology
Field of study:	Cybersecurity
Title:	**Black-Box Analysis of Wi-Fi Stacks Security**
Category:	Security

Assignment:

1. Get familiar with existing IoT platforms with focus on ESP32 platform, study existing attacks on Wi-Fi stack level and principles of fuzz testing.
2. Analyze existing vulnerability detection approaches and existing tools.
3. Based on the results of the analysis, design set of opensource tools for detection of IoT vulnerabilities on WIFI stack level using black-box fuzzing techniques (i.e. injecting semi-random data into a real communication in order to reveal buffer overflows and other potential flaws).
4. Implement designed toolset. Consider future extensions to cover wider set of platforms.
5. Test and evaluate the effectiveness of the tool.

Recommended literature:

- Bart Pleiter: Fuzzing Wi-Fi in IoT devices, bachelor thesis (https://www.cs.ru.nl/bachelors-theses/2020/Bart_Pleiter___4752740___Fuzzing_Wi-Fi_in_IoT_devices.pdf (2020))
- Wifuzzit - wireless fuzzer focused on 802.11 technology (https://github.com/0xd012/wifuzzit (unmaintained))
- L. Butti: Wi-Fi Advance Fuzzing (https://www.blackhat.com/presentations/bh-europe-07/Butti/Presentation/bh-eu-07-Butti.pdf)

Requirements for the semestral defence:

- Items 1 to 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Malinka Kamil, Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 19, 2021
Approval date:	November 11, 2020

Abstract

The devices on which we rely in our day-to-day lives use complicated protocols. One of the heavily used protocols is Wi-Fi. The growing complexity also increases the room for error during implementation. This thesis studies the Wi-Fi protocol and the use of fuzz testing to generate semi-valid frames, which could reveal vulnerabilities when sent to the tested device. Special attention was devoted to testing the Wi-Fi stack in the ESP32 and ESP32-2S systems. The output of the thesis is a fuzzer capable of testing any Wi-Fi device, a special ESP32 monitoring tool and a set of ESP32 test programs. The tools did not find any potential vulnerabilities.

Abstrakt

Zariadenia, na ktoré sa každodenne spoliehame, sú stále zložitejšie a využívajú zložitejšie protokoly. Jedným z týchto protokolov je Wi-Fi. S rastúcou zložitosťou sa zvyšuje aj potenciál pre implementačné chyby. Táto práca skúma Wi-Fi protokol a použitie fuzz testingu pre generovanie semi-validných vstupov, ktoré by mohli odhaliť zraniteľné miesta v zariadeniach. Špeciálna pozornosť bola venovaná testovaniu Wi-Fi v systéme ESP32 a ESP32-S2. Výsledkom práce je fuzzer vhodný pre testovanie akéhokoľvek Wi-Fi zariadenia, monitorovací nástroj špeciálne pre ESP32 a sada testovacích programov pre ESP32. Nástroj neodhalil žiadne potenciálne zraniteľnosti.

Keywords

Wi-Fi, 802.11, ieee802.11, Wi-Fi frames, frame injection fuzzing, fuzz testing, testing, model fuzzing, black-box testing, negative testing, protocol fuzzing, protocol testing, Wi-Fi fuzzing, wifuzz++, ESP32, vulnerability

Klíčová slova

Wi-Fi, 802.11, ieee802.11, Wi-Fi rámce, fuzzing, fuzz testovanie, testovanie, modelový fuzzing, black-box testovanie, fuzzing protokolu, testovanie protokolu, Wi-Fi fuzzing, wifuzz++, ESP32, zranitelnosť,

Reference

VENGER, Adam. *Black-Box Analysis of Wi-Fi Stacks Security.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Kamil Malinka, Ph.D.

Rozšířený abstrakt

Bezpečnosť zariadení, na ktoré sa každodenne spoliehame, stále naberá na dôležitosti. Tieto zariadenia sú čoraz zložitejšie a so zložitosťou sa zvyšuje aj potenciál pre chyby v implementácií. Jeden z protokolov, na ktorý sa každodenne spoliehame je Wi-Fi. Táto práca sa zaoberá automatizovaným testovaním Wi-Fi stacku rôznych zariadení, predovšetkým však systémom ESP32. Práca vznikla v spolupráci s firmou Espressif, ktorá je výrobcom spomínaného systému.

Pre čo najlepšie otestovanie implementácie Wi-Fi bolo potrebné preštudovať protokol Wi-Fi. Konkrétny štandard pre protokol všeobecne známy ako Wi-Fi je IEEE802.11. V práci je rozdiel medzi týmito menami konkrétnejšie vysvetlený. Sú tu priblížené základné pojmy potrebné pre prienik do problematiky. Väčšia pozornosť je venovaná zasielaným rámcom, kedy sú zasielané, a obzvlášť štruktúre týchto rámcov. V súvislosti s rôznymi typmi rámcov je priblížený aj proces asociácie a pripojenia k prístupovému bodu.

Práca sa zaoberá potencionálnymi zraniteľnosťami, preto boli analyzované najčastejšie chyby, ktoré ich môžu spôsobiť. Sú predstavené existujúce nástroje určené pre hladanie zraniteľností. Medzi týmito nástrojmi sú aj fuzzery. Nástrojom pre fuzz testing bola venovaná bližšia pozornosť a aj samostatná kapitola.

Cieľom práce bolo navrhnúť fuzzer, ktorý by využil znalosti získané štúdiom Wi-Fi, zraniteľností, ktoré sa bežne môžu vyskytovať pri implementácií Wi-Fi a existujúcich fuzzerov. Počas dizajnu boli tieto skúsenosti zohľadnené a bol navrhnutý nový fuzzer. Ten je rozdelený na tri hlavné časti: generátor testovacích dát, časť zaisťujúca komunikáciu a časť monitorujúca testovaný systém.

Generátor dát využíva modelový fuzzing a voliteľne náhodný fuzzing niektorých elementov. Tento prístup bol zvolený z dôvodu najväčšej efektivity testovania s ohľadom na dostupné informácie o systéme. Efektivitu testovania hodnotíme z pohľadu potenciálu odhaliť vstupy spôsobujúce pád systému alebo iné neželané správanie. Informácie, ktoré sú dostupné o vnútornom stave testovaného systému sú minimálne a nie je možné zistiť, aká časť kódu je skutočne otestovaná. Z dôvodu príliš veľkej náročnosti testovania pomocou čisto náhodných vstupov, bol zvolený spôsob generovania rámcov podľa vopred stanovených pravidiel. Výsledné rámce boli vždy skoro valídne, okrem vybranej jednej časti. Ako príklad môže byť testovanie dĺžky pola s podporovanými rýchlosťami prenosu. Podľa štandardu musí byť toto pole dlhé najviac 8 bytov, no fuzzer skúsil aj väčšie veľkosti. Generované dáta boli volené s ohľadom na už existujúce zraniteľnosti odhalené v iných systémoch.

Pre monitorovanie testovaného zariadenia boli vytvorené viaceré monitorovacie nástroje. Každý nástroj je vhodný pre iný účel. Najdôležitejšia je sonda, ktorá je využívaná pri monitorovaní IoT systémov ako je ESP32. Je pripojená na sériový port a sleduje jeho výstupy. Pri reštarte túto skutočnosť zašle monitoru, ktorý zaznamená posledné zaslané rámce, ktoré môžu byť ďalej skúmané.

Časť, ktorá generuje rámce a časť, ktorá monitoruje testované zariadenie sú spojené jadrom. To sa stará o komunikáciu medzi týmito časťami a s testovaným zariadením.

Výsledný nástroj bol vyhodnotený na základe toho, koľko z už známych vytipovaných zraniteľností by dokázal odhaliť. Na základe porovnaní je možné tvrdiť, že väčšinu by odhaliť aj dokázal. Ako záverečný test boli pomocou implementovaného nástroja otestované systémy ESP32, ESP32-S2, mobilný telefón, tlačiareň a smart hodinky. Boli nájdené len minimálne odchýlky od očakávaného správania. Nič nenasvedčovalo tomu, že by testované zariadenia zlyhali, alebo obsahovali nejaké z testovaných zraniteľností.

Black-Box Analysis of Wi-Fi Stacks Security

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mgr. Kamil Malinka Ph.D. The supplementary information was provided by Martin Vychodil. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Adam Venger
May 17, 2021

Acknowledgements

I would like to thank the thesis supervisor Mgr. Kamil Malinka Ph.D. for his guidance during the writing of the thesis. Also I would like to thank a consultant from the Espressif Martin Vychodil for his expert knowledge and provided information. And thanks to both of them for the opportunity to work on such an interesting topic, which sparked a new interest in me.

Contents

1 Introduction **4**

2 IEEE 802.11 **5**

2.1 Different names . . . 5
2.1.1 Wi-Fi . . . 5
2.1.2 Wireless Ethernet . . . 5
2.1.3 Names in this publication . . . 6
2.2 IEEE 802 networks . . . 6
2.3 Transmission . . . 7
2.3.1 Frequencies . . . 7
2.3.2 Physical components . . . 8
2.4 Network types . . . 8
2.5 Frame Format . . . 9
2.5.1 Radiotap . . . 9
2.5.2 Frame control . . . 9
2.6 Control frames . . . 11
2.6.1 Acknowledgement . . . 11
2.6.2 Request to send and Clear to send . . . 11
2.7 Management frames . . . 11
2.7.1 Beacon . . . 12
2.7.2 Probe request and response . . . 12
2.7.3 Disassociation and deauthentication . . . 12
2.7.4 Association and Reassociation Request . . . 12
2.7.5 Authentication . . . 12
2.8 Data frames . . . 12
2.9 Association process . . . 13
2.9.1 Scanning . . . 13
2.9.2 Authentication . . . 14
2.9.3 Association . . . 15
2.10 Association process in the newer versions. . . . 15

3 Vulnerability detection **16**

3.1 Security goals . . . 16
3.2 Potential vulnerabilities . . . 16
3.2.1 Categories of vulnerability causes . . . 16
3.3 Exploitation of the vulnerabilities . . . 17
3.4 Exploit mitigation . . . 18
3.4.1 Data executaion prevention . . . 19

3.4.2 Stack cookies 19
3.4.3 ASLR 19
3.5 IoT vulnerabilities 19
3.6 ESP32 20
3.6.1 Exploit mitigation 20
3.6.2 Previous exploits 21
3.7 Existing tools 21
3.7.1 Nonexploitation vulnerability scanners 21
3.7.2 Exploitation scanners/Frameworks 21
3.7.3 Fuzzers 22
3.8 Wi-Fi vulnerabilities 22

4 Fuzz testing 24
4.1 What is fuzz testing? 24
4.2 Advantages and disadvantages of fuzzing 24
4.3 Intelligent fuzzing 25
4.3.1 Fuzz data generation 25
4.3.2 Price of intelligent fuzzing 26
4.4 Fuzzing process 26
4.5 Black-box and white-box fuzzing 26
4.6 Case study 27
4.7 Wi-Fi fuzzing 27
4.7.1 Johnny Cache and David Maynor 28
4.7.2 Laurent Butti 28
4.7.3 Mathy Vanhoeft 29
4.7.4 ASSET 29
4.8 Existing tools 30
4.8.1 Fuzzers 30
4.8.2 Frameworks 30
4.9 Summary 31

5 Design 32
5.1 Design Requirements 32
5.2 Structure 33
5.3 Fuzzer core 34
5.3.1 Set-Up and Tear-Down 35
5.3.2 Communication with frame fuzzer 36
5.3.3 Communication with monitor 36
5.4 Monitor 36
5.4.1 Universal vulnerability detection 36
5.4.2 ESP32 monitoring 37
5.4.3 Logging 37
5.5 Frame fuzzer 38
5.5.1 Fuzzing approach 38
5.6 Summary 40

6 Implementation 41
6.1 Fuzzer Core 41

6.1.1 Configuration 41
6.1.2 Sniffing 41
6.2 Monitor 41
6.2.1 Watchdog 42
6.2.2 Notification monitor 42
6.2.3 Passive monitor 42
6.2.4 Probe monitor 42
6.2.5 Remote probe 43
6.3 Frame fuzzer 43
6.4 ESP32 test programs 44
6.4.1 Probe response test 44
6.4.2 Beacon frame test 44
6.4.3 Deauthentication and disassociation frame test 45
6.5 Test Runner 45
6.6 Summary 45

7 Testing **46**
7.1 Design requirements 46
7.2 Performance requirements 46
7.3 Functional testing 47
7.3.1 Beacon frames 48
7.3.2 Probe response 49
7.3.3 Disassociation and Deauthentication 50
7.3.4 Authentication 50
7.4 Fuzzing coverage 50
7.4.1 Categorization of the found vulnerabilities 51
7.4.2 Results 52
7.5 Device testing 52
7.5.1 Experiment design 52
7.5.2 ESP32 and ESP32-S2 53
7.5.3 Samsung Galaxy S9 53
7.5.4 HP Deskjet 3545 56
7.5.5 Samsung Galaxy Watch Active2 56
7.6 Summary 56

8 Conclusion **57**

Bibliography **58**

A Test vulnerabilites **62**

B Contents of the included storage media **64**

Chapter 1

Introduction

There is a saying: “Every non-trivial program has at least one bug.” This is especially true when dealing with input from an external source, as, many times, there is an infinite number of different inputs that can be passed to a program. Devices in wireless networks must be able to validate the input passed to them. The input may come from a friendly device, a malicious attacker, or a malfunctioning device. When an input is not considered, it can potentially lead to complete control by the attacker. Fuzzing or creating pseudo-random data to the programs is one way to find such inputs.

This thesis aims to design a tool (or set of tools) able to find vulnerabilities in the Wi-Fi implementation of, mainly, the ESP32 by Espressif Systems Co. The thesis was written, and the tool was developed as an assignment by the Espressif. The ESP32 chips are used by millions of devices worldwide, and such vulnerabilities would have tremendous impacts. The developed tool could also serve as a form of regression testing for the developers of the ESP32 Wi-Fi stack. Apart from the ESP32 chip, the tool should be able to test any device with Wi-Fi capability. The Wi-Fi specification, together with known previous types of vulnerabilities, should be studied to develop the best possible testing strategy. The tool should provide an effective way to create Wi-Fi frames, which are the most likely to cause problems and reveal faults in the Wi-Fi stack implementation. Potential to reveal faults should be based on the research and previous experience with the vulnerabilities found. The effectiveness of the tool is not evaluated as the ability to find all possible faults. Instead, it is considered the ability to test most of the known and possible vulnerabilities in a reasonable amount of time. Fuzz testing is an appropriate form of testing for this objective. It has many forms. As a part of this thesis, they will be studied, and the best possible design for the specific use-case should be used. The existing tools should be evaluated and compared to the newly created testing tool.

This thesis first describes the Wi-Fi standard and its internals in Chapter 2. Then the potential vulnerabilities with the common causes and tools for their discovery are outlined in Chapter 3. Chapter 4 takes a closer look at fuzz testing, its advantages and some tools, which could be used for fuzzing Wi-Fi. The design of a new fuzzer, based on lessons learned from previous chapters, is described in Chapter 5 and the implementation details in Chapter 6. Lastly, the final design was evaluated by its ability to reveal previously known vulnerabilities and tested on real devices in Chapter 7.

Chapter 2

IEEE 802.11

Detailed knowledge of the Wi-Fi protocol, the frames sent, and the communication process is needed to test the Wi-Fi implementations adequately. This chapter describes a family of network protocols for wireless local area networking called `IEEE 802.11`. It was created and now is maintained by the IEEE [1] 802 LAN/MAN Standards Committee. The description is based mainly on the second edition of the book *802.11 Wireless Networks: The Definitive Guide* by *Matthew Gast* [21]. Here different names for the 802.11 protocol will be described, the relation to other IEEE 802 standards, the terms needed to understand later chapters, and what comprises the different 802.11 packets.

2.1 Different names

Common names for the IEEE 802.11 standard are *Wi-Fi* or Wireless Ethernet. To establish what they mean, we need to look at the historical evolution and content of the standard.

2.1.1 Wi-Fi

The IEEE 802.11 is most often called Wi-Fi. There is, however, a difference between them. Wi-Fi Certified devices are marked by the logo shown in figure 2.1, which is *"an internationally-recognized seal of approval for products indicating that they have met industry-agreed standards for interoperability, security, and a range of application-specific protocols"* [36]. The device must be tested by one of the authorized test laboratories. The certificate granted to a device by Wi-Fi Alliance [37] to mark its compatibility with the IEEE 802.11 standard. The implication is that all Wi-Fi devices are IEEE 802.11 capable, but not all devices capable of using the IEEE 802.11 standard are Wi-Fi certified.

2.1.2 Wireless Ethernet

Another name for the 802.11 is *wireless Ethernet.* The name comes from the shared linage with the wired 802.3 – Ethernet (also not entirely correct name) standard.

[1] IEEE – Institute of Electrical and Electronics Engineers

Figure 2.1: The *Wi-Fi Certified* logo [38] is used to confirm that the device is compatible with 802.11 and was tested by the Wi-Fi Alliance certified laboratory.

2.1.3 Names in this publication

For the purposes of this publication, the mentioned names can and will be used interchangeably. When describing technicalities of the standard, the standard will be called 802.11. The preferred name elsewhere will be Wi-Fi, as it is the most common.

2.2 IEEE 802 networks

IEEE 802 describes a series of specifications for local area network technologies, one of which is the 802.11. Based on the OSI model, networks are divided into seven layers (pictured in Table 2.1). The specifications concerning IEEE 802 networks are focused on the two lowest layers of the OSI model – the physical and data link layers. They all specify both of these components.

7	Application
6	Presentation
5	Session
4	Transport
3	Network
2	Data Link
1	Physical

Table 2.1: The OSI network model with the highlighted Data Link and Physical layers, which are specified by the 802.11.

802 divides the data link layer into two sublayers MAC (Media Access Control) and LLC (Logical Link Control), as shown in figure 2.2. As the name suggests, MAC is responsible for controlling the access to the shared medium for sending and receiving data. Its implementation is different for different protocol standards (such as 802.3, 802.11). 802.2 specifies LLC as the upper sublayer of the data link layer. It encapsulates data for higher layers, provides error checking and frame synchronization. The 802.11 uses this encapsulation, as do other 802 specifications.

The physical layer is different for every 802 protocol, as well as for every version of 802.11. The base version, for example, includes a specification for two physical layers: frequency-hopping spread-spectrum (FHSS) and direct-sequence spread-spectrum (DSSS) modulation. Radio transmission of 802.11 requires a complex physical layer. More information about the physical layer is probably out of the scope of this thesis. If the reader

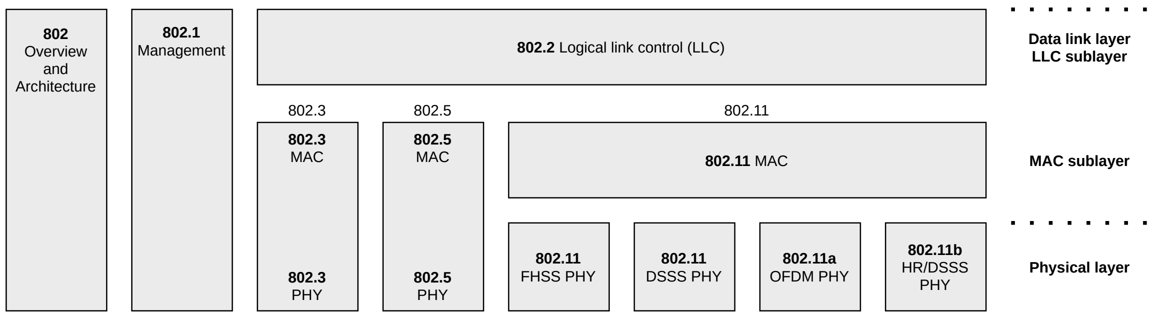


Figure 2.2: The division of the 802 standards into the OSI model layers and sublayers. The Figure is adopted from the *802.11 Wireless Networks: The Definitive Guide* [21].

is interested, the book *802.11 Wireless Networks: The Definitive Guide* [21], on which this chapter is based, contains more information.

2.3 Transmission

802.11 is a wireless standard. It uses (mainly) electromagnetic field as a medium, which means it must be treated as unreliable. With more and more Wi-Fi devices in close proximity, all using the same shared medium, the chances of disturbance and collision climb greatly. Details of the transmission vary version by version, but some of the terms remain. Those will be explained in this section.

2.3.1 Frequencies

When using the electromagnetic field as a medium, the wireless devices are constrained to operate in a particular frequency band. Regulatory bodies control the use of frequencies. Each region has its own regulatory body, which has its own rules for frequency use. The base version of the 802.11 standard uses the 2.4GHz frequency band. Those frequencies are generally free to use if the transmitted power is under a certain threshold. Other than by 802.11, the band is used by Bluetooth or microwaves. The frequency is divided into channels. What exactly constitutes a channel depends on the used physical layer. It could be a frequency band when a DSSS [2] physical layer is used or a hopping pattern when FHSS [3] is used. The devices agree on the physical parameters of the communication during the scanning and association.

More recent versions of the standard allow the use of the 5GHz band for higher throughput and lower latency. The two mentioned frequencies are the most widely used.

[2]DSSS – Direct Sequence Spread Spectrum
[3]FHSS – Frequency Hopping Spread Spectrum

2.3.2 Physical components

The 802.11 network consists of a few types of physical components.

Wireless medium

The 802.11 generally uses an electromagnetic field as a medium. Nevertheless, the standard permits also other types of physical layers. The base 802.11 standardized use of two radio frequencies and one infrared physical layer.

Access point

Access points serve as a bridge from a wireless network to the rest of the world.

Stations

Stations are computing devices whose data the networks wirelessly transmit. They are mostly battery operated.

Distribution system

Thanks to limited transmitting power and disturbance of the medium, one access point covers only a limited amount of space. A distribution system with more access points is used in order to make the covered area larger. It connects them into an extended service area.

2.4 Network types

The basic building block of an 802.11 network is the basic service set (BSS), a group of stations that communicate with each other. The communication takes place in a basic service area with fuzzy boundaries. The BSS is divided into two types.

Independent

The first type of BSS is the independent basic service set (IBSS) network. It is comprised of more peer stations. They communicate directly with each other, without any access points. The stations must be a direct communication range. IBSSs are sometimes called *ad hoc networks.*

Infrastructure

The second type of BSS is the infrastructure BSS. The name is never shortened to avoid confusion. Those networks use access points. An access point is used even when trying to communicate with stations within a direct communication range. Stations must associate themself with an access point to obtain network services. The use of access points has advantages other than the apparent ability to communicate with the outside world. It can assist stations trying to save power. During communication, a station can signal this to the access point, which then buffers frames for that station and transmits them later. This reduces the time the station needs to stay active, which reduces power consumption at the expense of latency.

2.5 Frame Format

Thanks to the more complicated nature of wireless networking, the MAC frames are more complicated than their wired equivalents. 802.11 uses three types of frames with different fields, which are then divided into more subtypes. In wireless networks, data transfer must be synchronized to avoid interference. There needs to be a mechanism to request access to the medium, confirm the operation's success, or find access points. Furthermore, the wireless interface must be able to configure itself based on the network correctly.

2.5.1 Radiotap

Radiotap is a type of pseudo-header, which is not sent over the network but is used to configure the network interface when sending and get information about the interface when receiving frames. The configuration is heavily dependent on network card drivers because the radiotap headers might be ignored. Some drivers set the interface based on the radiotap header, some need to be configured by other means, and some use only a subset of the parameters configured in the header. The data rate, timestamp, or channel used are a few of the fields contained in the radiotap header.

2.5.2 Frame control

Two bytes at the start of each 802.11 frame are *frame control*. They contain frame type, subtype, or direction flags. The semantics of some bits can be changed based on the type and subtype of the frame. The frame control structure, which can be seen in figure 2.3 and the Table 2.2 shows the type and subtype of the frame.

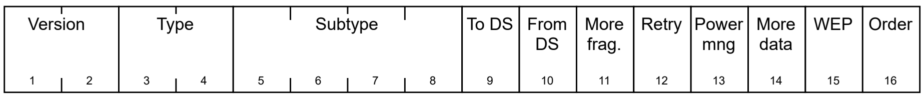


Figure 2.3: Frame control bits at the start of every 802.11 frame contain the most fundamental information about frames. The content affects the semantics of the following bytes. The figure is adopted from the *802.11 Wireless Networks: The Definitive Guide* [21].

Version

At the time of writing the text, only one version is used. It has version number 0.

To DS and From DS

They are separate fields because of the possibility to use IBSS without DS or use DS as a bridge. The exact meaning is shown in the Table 2.3.

	To DS=0	To DS=1
From DS=0	Control and management frames IBSS data frames	Data frames from station in infrastrucutre network
From DS=1	Data frames received for station in infrastructure network	Data frames in wireless bridge

Table 2.3: Meaning of different combinations of *To DS* and *From DS* bits.

Value	Subtype name
Management (type=00)	
0000	Association request
0001	Association response
0010	Reassociation request
0011	Reassociation response
0100	Probe request
0101	Probe response
1000	Beacon
1010	Disassociation
1011	Authentication
1100	Deauthentication
1101	Action
Control (type=01)	
1010	Power Save (PS)-Poll
1001	Block ACK
1011	RTS
1100	CTS
1101	Acknowledgement (ACK)
1110	Contention-Free (CF)-End
1111	CF-End+CF-Ack
Data (type=10)	
0000	Data
0001	Data+CF-Ack
0010	Data+CF-Poll
0011	Data+CF-Ack+CF-Poll

Table 2.2: Types and subtype values together with their meaning. The type and subtype in this table are written least-significant-bit first. The table is not complete. It contains only a subset of all frame subtypes.

More fragment

More fragment flag is used when a higher-level protocol packet is too large to be transmitted at once. It can be used by some management frames as well. When another part of the fragment is to be transmitted, the bit is set to 1.

Retry

The frame is flagged by setting the retry bit to 1 to signal when the frame must have been retransmitted.

Power management

The 802.11 specification includes a power-saving mechanism, which allows the stations to power the network interface and receive frames targeted to them later. The station indicates this intention to the receiver by setting the power management bit of the last message to 1.

More data

This bit is used for power saving too. It should be set to 1 by the access point when more frames are available to be sent, addressed to a device that announced going into a sleep state.

WEP

Wired Equivalent Privacy (WEP) is used to protect and authenticate data. When this mechanism is used, the bit is set to 1. Other encryption mechanisms should be used instead of WEP since it is not considered safe anymore.

Order

Strict ordering can be enforced, which is indicated by setting order bit to 1. Strict ordering means the frames are transmitted in order. This costs additional time and resources at both ends of the communication.

2.6 Control frames

Control frames assist in the delivery of data frames. They conduct access to the wireless medium and provide MAC-layer reliability functions.

2.6.1 Acknowledgement

Acknowledgement (ACK) frames are used as a confirmation when the transmission was successful. They do not contain the sender's address, only the receiver's. The later version of the standard use more effective block acknowledgements (BA). They were introduced in 802.11e, and from 802.11n, their support was mandatory.

2.6.2 Request to send and Clear to send

Request to send (RTS) is used by the device when it wants to gain control of the medium in order to transfer larger frames. RTS is used only for unicast. Clear to send (CTS) frames are sent as a response to the RTS.

2.7 Management frames

Management frames are used for purposes of identification of access points, authentication, and association. They can contain up to 4 addresses based on subtype and variable-length information such as devices capabilities. The further mentioned information elements in the frames are often only a typical representation of the frame. The specification is extensive and adds more and more information elements to different subtypes in each version.

2.7.1 Beacon

Beacon frames are used by access points to announce the existence of the network. They contain information about supported data rates and other parameters needed by the device to join the network. The basic service area is defined by the area where the beacon frames appear.

2.7.2 Probe request and response

Probe request is used for similar reasons as a beacon frame. It is transmitted by a mobile station trying to find an access point. The request can contain a specific SSID of the network the station is trying to find, or it can contain a wildcard symbol (in reality, an empty string), which means it is trying to find all networks. The number of SSIDs in the request can be more than one.

The probe response is sent as a response from the access point to the requesting station. It can contain most of the same parameters about the network as a beacon frame. Some of the information elements should be only in beacon frames and some only in the probe responses.

2.7.3 Disassociation and deauthentication

Disassociation frames are used to end an association relationship, and Deauthentication frames are used to end an authentication relationship. Both contain reason code with fixed length.

2.7.4 Association and Reassociation Request

When the station wants to connect to the access point, it must associate with it first. This can be done by the association request frame. It contains capabilities, SSID of the access point, and listen interval. The capabilities must be compatible with the access point for the association to happen. The reassociation request can be used when moving within the same extended service area. In addition to the field in association request, it contains the address of the current access point.

The responses to those requests are called Association/Reassociation Response. They contain the status of the association and association ID.

2.7.5 Authentication

Authentication is done by exchanging series of authentication frames. They include authentication algorithm used, status code, and challenge text.

2.8 Data frames

Data frames are used to encapsulate higher-level protocols. Based on the direction flags (From DS, To DS), the number and semantics of addresses in the header change. The body with higher-level protocols comes after them. The body can be encrypted, which is signalled by the WEP flag in frame control. The encrypted data start with their own headers.

2.9 Association process

The frames are divided into three classes based on authentication and the association state of the devices. The states of the device can be:

1. Initial state
2. Authenticated
3. Authenticated and associated

The device can receive only the frames within its own class or lower, illustrated by Figure 2.5. The data can only be sent when a device is associated with the access point.

2.9.1 Scanning

The first step for the device to starts using the network is to find the network. Scanning can be done actively by sending probe request frames and waiting for a response or passively by listening to beacon frames sent by other devices. The scanning device can filter the found networks based on the parameters of the communication and the network:

- **SSID** – The SSID is a string of characters assigned to an extended service set.
- **BSS Type** – The device can select if it scans for the ad-hoc networks, infrastructure networks with access points, or both.
- **Channel List** – The scanning can be broad and include all the channels available, or it could be narrowed to include only some subset of them. The channels are dependent on the physical layer used.
- **Minimal and maximal channel time** – The device must change the physical parameters of the communication in order to change the channels. When scanning, it alternates between the channels. The time the scans one channel can usually be set by the minimal and maximal time.

Passive scanning

Passive scanning does not require transmission from the scanning device. It only listens to the beacon frames coming from the other devices. The beacon frame can be transmitted on different channels. The station systematically moves through the specified channels to and records the received beacon frames. They contain the channel the device operates in and other crucial information. The beacon frames are broadcasted and do not have to be acknowledged.

Active scanning

Some access points are set to not transmit beacon frames in order to stay hidden. Active scanning must be used to discover them. The station device sends a probe request with the specified SSID, to which the other device responds with the probe response containing similar info as the beacon frames. The probe response could also contain an empty SSID, which is considered a wildcard symbol, meaning the device is trying to scan for all networks in the radio range. The probe response destination is the device, which sent the probe request and must be confirmed by acknowledgement.

2.9.2 Authentication

Authentication is used to ensure the devices trying to associate are allowed to do so. Authentication can be *open-system* when all devices are allowed to associate or *shared-key* based on WEP. The standard does not restrict the authentication only to the access point. However, the authentication to the access point is characteristic.

Open authentication

Open authentication does not implement any cryptographic algorithms or any restrictions on authentication. The access point could implement a whitelist of the MAC addresses allowed to authenticate, but those could be changed in software. The authentication is initiated by the client, who sends the authentication frame with the algorithm set to open and sequence number set to 1. If the access point accepts the open authentication, it should respond by authentication frame with the algorithm set to open-system, sequence set to 2 and successful status code.

Shared-key authentication

The shared-key authentication requires the transmission of at least four frames. The authentication is also initiated by the client, who sends the authentication frame with the algorithm set to shared-key. The access point then sends an authentication frame with 128 bytes of challenge text.

The WEP uses RC4, which is a symmetric stream cypher. The encryption is done by XORing the cypher stream with the plaintext data. The cypher stream is generated from a 24-bit initialization vector (IV) and 40 bit secret key. When using the WEP, only the frame body is encrypted. The header and the IV are in plaintext, followed by the encrypted frame body and integrity check value, ending with the plaintext frame control sequence. The frame using WEB is illustrated in the figure 2.4.

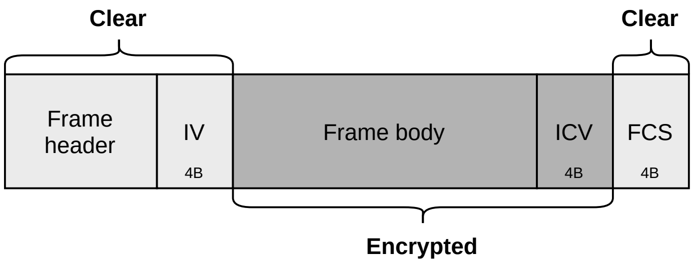


Figure 2.4: The diagram shows the encrypted and clear parts of the frame using WEP. The figure is adopted from the *802.11 Wireless Networks: The Definitive Guide* [21].

At the time, it was considered to be appropriate, but now flaws in the RC4 cyphers are known, and it is no longer safe to use. The RC4 itself must be licensed from the RSA Security and cannot be implemented without it. This prevents open-source implementations. WEP is often implemented in the hardware of the wireless cards, which allows its use even with open-source drivers.

2.9.3 Association

The device can be associated with only one access point at a time, as is specified by the standard. The process of association is started by the station with an association request. The access point responds with association response. Whether the response is positive or negative is on the access point, and the standard does not specify how to determine the response status.

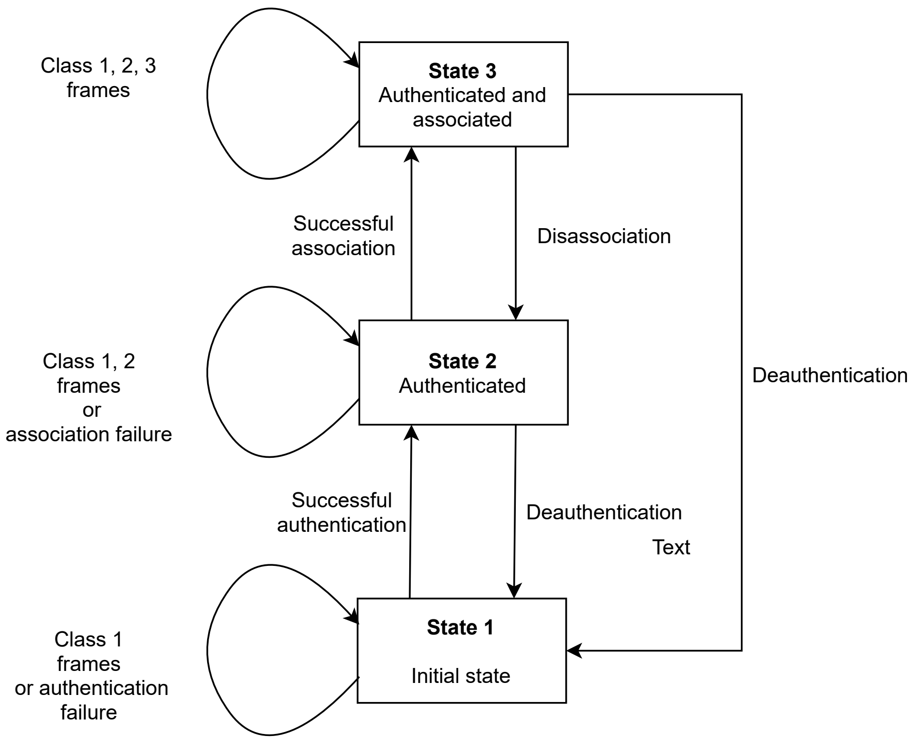


Figure 2.5: Diagram of the 802.11 communication for different classes of frames. The figure is adopted from the *802.11 Wireless Networks: The Definitive Guide* [21].

2.10 Association process in the newer versions.

The association model of the original Wi-Fi protocol is expanded to allow more advanced encryption and authentication to be used. The newer version of the protocol's encryption is built on the association to the open network and subsequent authentication and encryption by more advanced methods. For example, the widespread WPA2 encryption uses the data frames (four frames in total) to arrange secure connection.

Chapter 3

Vulnerability detection

When software engineering was in its infancy, security was not considered to be a priority. Thanks to the rise of the internet and the wide usage of wireless networks, this has changed. Vulnerabilities could enable an attacker to access the user's data, corporate, or government secrets. Some estimates from authors of the book *Fuzzing for Software Security Testing and Quality Assurance* [32] state: "More than 70% of modern security vulnerabilities are programming flaws, with only less than 10% being configuration issues, and about 20% being design issues." Using tools to increase the chances of finding programming flaws could significantly reduce the number of vulnerabilities. This chapter describes the security goals, potential vulnerabilities, and tools designed to discover them. It is based on the findings from the book mentioned.

3.1 Security goals

To define what a vulnerability is, first, we need to define the goals of the security. In some applications, where the intended use does not depend on them, not all security goals must be preserved. The three primary security goals are:

- Confidentiality
- Integrity
- Availability

3.2 Potential vulnerabilities

Vulnerability analysis is a process of finding weaknesses in software or systems, which could potentially lead to breaking one of the wanted security goals. Not all bugs result in security vulnerabilities. For those that do, proof of concept can be used to prove its potential to be harmful.

3.2.1 Categories of vulnerability causes

Potential vulnerabilities can be categorized into programming, configuration, or design issues, as previously stated. The categories can be further divided into their causes. The causes mentioned in this section are related to this thesis, but many more exist.

Memory corruption errors

The most prevalent methods of computer systems exploitation have their roots in memory management and corruption. The memory corruption can result in full execution rights on the attacked system.

The memory corruption errors include:

- Stack overflow – memory on the stack getting corrupted
- Heap overflow – memory corruption on the heap, usually allocated at runtime
- Format string errors – now can be easily detected by static analysis
- Integer errors – errors in signedness, numerical wrapping, or field truncation
- Off-by-one – typically one too many bytes in the buffer, or improper condition checking
- Other memory overwrites – overwriting function pointers, credentials, etc.

Race condition

Race conditions are a type of vulnerability, which arise due to unforeseen timing events and are typical in multithreaded or asynchronous applications. They are difficult to exploit, but with enough patience, an attacker could manage to do it. Sometimes putting the system under more load can increase the likelihood of exploiting the fault.

Denial of service

As the name suggests, denial of service is the act of overwhelming a system to the point at which it can no longer serve its legitimate purpose. It could be caused by a crash of the system due to a programming error. Alternatively, it could be due to a design decision that allows the attacker to send lots of simple requests, which causes enormous resource consumption. The stress caused by this could also lead to a revelation of other vulnerabilities.

3.3 Exploitation of the vulnerabilities

How the vulnerabilities are exploited and what mitigation techniques can be used to lessen the impact of vulnerabilities will be explained on the example of stack overflow – one of the most prevalent vulnerabilities. The first widely accessible tutorial on exploiting buffer overflow is considered an article published in the *Phrack* web magazine. Its name was *Smashing the stack for fun and profit* by Elias Levy, also known as *Aleph One* [2]. This article is the source of most of the section.

Before the attacker is able to exploit a buffer overflow, the attacker must know the architecture and some essential information about the exploited system. The knowledge of architecture is crucial to perform code execution by exploiting a buffer overflow. The architecture determines the instruction codes, which the attacker must know. If the exploited device is desktop or server, the chances are it uses an x86-64 instruction set. If the device is a smartphone or smartwatch, it probably uses an Arm instruction set. And if the device is an IoT device, it could use an Arm or RISC-V instruction set.

The memory layout of the program is also crucial. The attacker must know the direction of growth of the stack and if the stack pointer points to the top of the stack or the first empty space. Another crucial piece of information for the attacker is the calling convention of the platform. This determines how the parameters are passed to the function and how the result is returned. These information are not hard to get and cannot be treated as a secret.

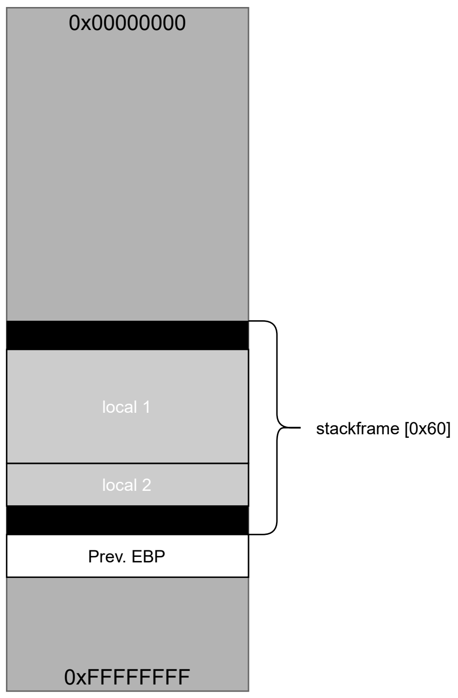


Figure 3.1: Address space of the program loaded into the memory after the function was called.

For example, if the device uses x86 32bit architecture and runs a Linux operating system, the address space during the execution of some program could look like in the figure 3.1. The stack grows from the higher addresses to the lower. If the program contains stack overflow, its exploitation could be relatively simple. The longer input could rewrite the function's return address, and from the moment the program tries to return, the attacker controls what instruction is executed next. The typical payload also contains a *shellcode*, which runs an interactive shell. If this is executed in a program running with root privileges, the attacker also has those. The less typical stack layout, which grows from the lower addresses, is also exploitable, as shown in the article by Zhodiac [40].

3.4 Exploit mitigation

Some techniques were developed to combat exploitation, which make the exploitation harder. Some of them require even hardware support. However, the techniques mentioned cannot completely protect the vulnerable device. The information about the mitigations is sourced mainly from the book *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities* [12]. The list is not complete. Many more protections against code injection, control-flow hijack, and data corruptions exist.

3.4.1 Data executaion prevention

The address space is divided into pages, which could belong to the stack, heap, or contain the program's instructions. The pages could be marked by NX (non-executable) bit, which prevents the data from being decoded and executed as instructions. Effective use of the data execution prevention (DEP) requires hardware support.

3.4.2 Stack cookies

As a way to combat stack overflow specifically, Stack cookies (or stack canary) are used. It is a few bytes (how many depends on the architecture) written to the address space after the local variables and before the return address. The bytes are random and are different in each function invocation. Before the function returns, it compares the content of the stack canary with the source of the random data. When they are different, the content has been modified and so could be modified the return address. Instead of return, the program exits without giving control to the attacker. In certain situations, the attacker can control the program even before returning from the function by overwriting the data around the return pointer. Local variables are not protected at all. They are often preventively reordered by the compiler to move the buffer after other variables, so the attacker cannot overwrite them by the stack overflow.

3.4.3 ASLR

Address space layout randomization is one of the more advanced and more effective mitigation techniques. Each time the program is loaded into the memory, it is loaded to a different address. The same is true for the stack, heap and libraries. The program must be compiled with the support for position independent code for this to work. The PaX project states the goal of the ASLR as following: *"The goal of Address Space Layout Randomization is to introduce randomness into addresses used by a given task. This will make a class of exploit techniques fail with a quantifiable probability and also allow their detection since failed attempts will most likely crash the attacked task."* [29]

3.5 IoT vulnerabilities

According to market analysis by Mordor Intelligence [27] the IoT chip market in 2020 was valued at USD 12 billion and is expected to grow to more than twice its size in 2026. This presents enormous potential for chipmakers but also for malicious agents. Currently, the IoT is present mainly in home entertainment, but more and more home appliances use the IoT approach. It is expected that more IoT devices will be used as asset trackers, health monitors, security systems, smart city sensors, and smart meters, or wearable trackers. The security of IoT devices was not always such an important quality. The Mirai malware, which caused one of the most disruptive DDoS attacks, was spread by connecting to the IoT devices and trying common default passwords. Its variants, like Satori, also used some implementation vulnerabilities [3].

The exploit mitigation techniques used in desktop and server computers are often not implemented in embedded devices. Many of them run their programs in privileged modes on "bare metal" without any operating system. Their runtime, memory, or power usage can be constrained by hard limits, which are hard to achieve. Additional protection would also impose additional load on the devices.

For example, DEP in desktop computers is implemented in MMU (Memory Management Unit). The MMU is not present in the microcontrollers, but many of them have an MPU (Memory Protection Unit), which can enforce read, write and executive permissions. However, through the fact that most of the programs in embedded computer run in privileged mode, it can be disabled. Its use in the FreeRTOS is only optional. The FreeRTOS is currently one of the most popular operating systems for IoT devices. Additionally, the MPU implementations leave some things to be desired [41]. The ASLR is also rarely used in the embedded systems. Some effort is being put into making the code produced by the compiler more resistant to the attacks. The most popular C/C++ compilers have the option to enable stack cookies. This is available without special hardware support. The addition of new LLVM passes to the LLVM toolchain also allowed the privilege overlaying used in the EPOXY [10].

3.6 ESP32

The thesis focuses on the ESP32 system. The ESP32 is series of microcontrollers with integrated Wi-Fi and Bluetooth capabilities [13]. The systems in the series differ in their performance, capabilities and even architecture, with the ESP32-C3 being based on RISC-V [16] architecture and the rest of them being Xtensa CPUs [15]. They are very popular, together with their predecessor, the ESP8266. The ESP8266 gained popularity thanks to its price and Wi-Fi capabilities. The ESP32 series continues the trend. The ESP32 devices usually use the FreeRTOS operating system, as it is the most popular across all the IoT systems.

From the security features, the ESP provides:

- Secure boot
- Flash encryption
- Hardware acceleration for the AES, SHA-2, RSA, ECC, RNG

Probably the most attractive features of the ESP32 are the Bluetooth and Wi-Fi capabilities. Even though the usual operating system for the chip is an open-source FreeRTOS, the Wi-Fi stack implementation is proprietary. This makes security assessment significantly harder. The Wi-Fi provides the following features:

- 802.11 b/g/n certification
- Automatic beacon monitoring
- Simultaneous support for Infrastructure Station, Software Access Point and promiscuous mode

3.6.1 Exploit mitigation

According to the ESP32 datasheet, the device has MMU [15]. The memory is divided into instruction memory and data memory. This results in an effect similar to the NX marked pages in x86 architecture. The data from the stack cannot be executed.

During compilation, Stack cookies can be turned on. This can be done for all function or their subset. Impact on performance is dependent on the level of selected stack protection. The system also supports a similar feature targeting heap corruptions.

3.6.2 Previous exploits

During the life of the ESP32 chips, a few critical vulnerabilities were found. One vulnerability even required hardware revision, and the affected devices could not be fixed. Other exploits will be presented in section 3.8. The vulnerabilities are often identified by the CVE identification number. The mission of the CVE program [1] according to them is: *"The mission of the CVE (Common Vulnerabilities and Exposures) Program is to identify, define, and catalogue publicly disclosed cybersecurity vulnerabilities."* Most of the recently publicly disclosed information are listed in the catalogue.

CVE-2019-17391

The ESP32 provides a One Time Programmable (OTP) memory used for the secret keys used in the secure boot process and flash encryption. The talk at Black Hat Europe by *Limited Results* [23] revealed critical vulnerability allowing the attacker with physical access to read the value of the OTP memory and bypassing the secure boot process. The vulnerability was assigned CVE identification number CVE-2019-17391 [14]. The method used was glitching, which is manipulating the input voltage by the attacker to inject faults during critical operations. The author also found other ESP32 vulnerabilities by this method.

3.7 Existing tools

To effectively find vulnerabilities in a system, a variety of different tools could be used. They could be used to help with manual analysis (i.e. reverse engineering tools as radare2 [1]) or even fully automatic scripts to gain access to Wi-Fi access point with weak encryption. This section will focus more on the semi-automatic or fully automatic tools called *vulnerability* or *security scanners.*

3.7.1 Nonexploitation vulnerability scanners

The tools that run specific tests to find vulnerabilities from the database of known vulnerabilities are *nonexploitation* vulnerability scanners. They may perform port scanning, capture traffic, or other techniques to get more information about the scanned system to compare it to the database. The tools will not perform any exploitation but will only log the results. Because of this, the results may contain false positives. The scanner will also not find any unknown vulnerabilities. One example of a nonexploitation scanner is Nessus [2].

3.7.2 Exploitation scanners/Frameworks

The exploit itself, or something called proof of concept (POC), must be performed, which undeniably prooves exploitability. Exploitation scanners, same as the nonexploitation scanners, have a database of known vulnerabilities that they test. Unlike them, however, they

[1]radare2 – `https://rada.re/`
[2]Nessus – `https://www.tenable.com/products/nessus/`

try to exploit the tested system. They can utilize the ability to pivot, which uses one exploit to enable others until sufficient privileges are gained.

Metasploit [3] framework represents one of the most popular scanning and remote exploitation tools. It is a free collection of scripts and programs. The paid version includes even more features.

Aircrack-ng [4] is a collection of tools to assess Wi-Fi network security. It includes tools for monitoring, attacking, testing, and cracking Wi-Fi networks. It can be used to find weaknesses in Wi-Fi networks, perform replay attacks, or create fake access points. Network driver capabilities are different for every driver. Aircrack-ng is able to verify them or switch the card to monitor mode, which is needed for frame injection into a wireless interface.

3.7.3 Fuzzers

Fuzzers are another tool for finding vulnerabilities in systems. Unlike the mentioned scanners, the fuzzing can reveal previously unknown exploits as it does not employ any exploit database. Fuzzing is explained in detail in Chapter 4 together with examples of existing fuzzers.

3.8 Wi-Fi vulnerabilities

Wi-Fi was described in chapter 2. This section includes some examples and vulnerabilities specific to Wi-Fi. The source for this information is the book *Hacking Exposed Wireless* [39]. This thesis focuses on fuzzing the Wi-Fi implementation of devices. The types of attacks presented here are unlikely to be executed or revealed by fuzzing, but they are provided to illustrate other exploitation methods. The most likely error types found by fuzzing are memory corruption errors explained in 3.2.1.

Denial of service

Availability is a security goal for all Wi-Fi networks. One of the most accessible attacks that the 802.11 family of protocols is vulnerable to is the denial of service by deauthentication frames. The attacker can send a deauthentication frame as if it was from an access point, which forces the client to unconnect, and he must then try to reconnect again. The client device has no way to be protected if the device follows the standard in all versions of the protocol, where management frames are not encrypted. This attack can also be used to find the SSID of hidden networks. Network hiding is not a very effective way of protection, as it is transmitted in plain text in many packets, one of which is a reassociation request.

Apple AWDL exploit

Recent (at the time of writing) publication of iOS zero-click radio proximity exploit by Ian Beer [5] at Google Project Zero is an example of what exactly could happen when untrusted input is improperly checked. The exploit allows for full access to the user data without the user's knowledge, denial of service by restarting the device, or execution rights to the attacker, all wirelessly.

[3] Metasploit – https://www.metasploit.com/
[4] Aircrack-ng – https://www.aircrack-ng.org/

The exploit is based on buffer overflow in drivers for Apple Wireless Direct Link (AWDL), Apple's proprietary mesh networking protocol. The AWDL is based on 802.11 but has significant differences, especially in the management frames. It is enabled by default on every iOS device.

The vulnerability was found by reverse engineering and analyzing leaked function name symbols, and it took approximately six months. The exploit illustrates the creative use of resources and hard work required to find vulnerabilities in modern devices. It also illustrates that usage of test automation tools for Wi-Fi testing could be beneficial. It is possible, the Wi-Fi fuzzing would reveal such vulnerability, and it is highly probable that the use of fuzz testing during development would reveal it before deployment.

ESP32/ESP8266 Zero PMK Installation

The vulnerability found by Matheus Garbelini [20] allows the attacker to take control of the Wi-Fi device in an enterprise network. The attack is performed by sending EAP-Fail message in the final step of the connection to the access point. It allows the attacker in radio range to replay, decrypt, or spoof frames via a rogue access point. The catalogue number assigned to the vulnerability is CVE-2019-12587. A fuzzer called Greyhound was used for finding this vulnerability. The tool is better described in the next chapter in section 4.7.4. The tool also discovered two other vulnerabilities in the ESP32 and ESP8266. They are described in the articles *ESP32/ESP8266 EAP client crash (CVE-2019-12586)* [18] and *ESP8266 Beacon Frame Crash (CVE-2019-12588)* [19]. The exploits can be found on GitHub [5].

[5]ESP32/ESP8266 Wi-Fi Attacks GitHub – https://github.com/Matheus-Garbelini/esp32_esp8266_attacks

Chapter 4

Fuzz testing

We want to test the behaviour of the Wi-Fi stack of IoT devices. Fuzz testing or fuzzing is ideal for this. The effectiveness of fuzzing can be seen in the vulnerabilities found, like the ones presented in the previous chapter. It would test the devices against real-world threats, which means, if we find some vulnerability, an attacker can use it too if he is in the radio proximity of the device.

This chapter describes what fuzz testing is, its purpose, capabilities, and limitations. The information in the chapter is based on the book *Fuzzing for Software Security Testing and Quality Assurance* [32].

4.1 What is fuzz testing?

Fuzzing is a technique of negative testing. The most traditional form of negative testing is *fault injection*, which exists in two forms. The first form of fault injection is injecting faults to the actual tested device or system to test the testing capability. This technique was used traditionally in hardware development, but it could also be used in the software. The second form refers to injecting faults into data with the purpose of testing data-processing capability. The fuzz testing belongs to the second form and is performed by programs called *fuzzers*.

4.2 Advantages and disadvantages of fuzzing

Fuzzing is a type of testing. No type of testing is perfect, and as such, every type has some weak points and some advantages. It cannot replace other test procedures, but it should complement them. Fuzz testing is, at the moment, not considered a mainstream testing technique. Not all developers routinely use it. Whether it is due to knowledge requirements, time constraints, or something different. The fuzzing is effective for finding specific types of errors, which are less complex, but it is not a "silver bullet" for every error. It can be used during development or by an attacker with malicious intent.

One of the first practical uses of fuzzing was by Professor Barton Miller and group around him, when they tried to input random data into core UNIX utilities (such as ls, grep) in the article *An Empirical Study of the Reliability* [25]. He was also the first to use the term *fuzzing* in the context of testing software. They found that many of them would crash. This shows that random inputs could lead to new execution paths previously thought not to be reachable or just not sufficiently secured. Random inputs do not assume anything about the program's internal structure, leading to finding new corner cases. The fuzzers can be used to test almost all programs or protocol implementations. Due to the consequences of a remote breach by a hacker, one of the areas where fuzzing is the most critical is testing network protocols.

Fuzz testing can be considered a cost-effective testing tool or tool for vulnerability analysis. It can cover test cases, which would require a deep knowledge about a specific system, such as reverse engineering. Additionally, it takes less time. Simple fuzzers can be quick and cheap to be implemented. Those are generally based on sending purely random data to the system under test (SUT). For some programs, this could reveal a significant number of bugs, but for the better-tested programs, this is almost always too inefficient. The input space of most of the test subjects is too large to employ purely random fuzzing. The book *Fuzzing for Software Security Testing and Quality Assurance* says about the goal of the fuzzing the following:

> *If we consider a program to be a complex finite state machine, then the goal of fuzzing is to perform a random walk through the state space, searching for undefined states.*

This is too inefficient to be performed just by random data. For those purposes, more fuzzers started to employ a more "intelligent" approach to fuzzing. Section 4.3 contains further information about more efficient fuzzing techniques.

The intelligent fuzzers have some information about the protocol or program they are fuzzing. They are usually made to conform to the formal definition rather than concrete implementation. This encourages resource reuse, and one fuzzer can be used as a test for many competing implementations.

4.3 Intelligent fuzzing

The motivation to make fuzzers more "intelligent" came from the inefficiency of fuzzers, which provide only random data, and do not consider their meaning. A random test will only scratch the surface and model just the first message in communication. However, a more complex approach could test message structures or even message sequences. Each of the more knowledgeable fuzzers comes with some assumptions and is not as generic as purely random fuzzers. Some heuristics are used based on experiences with where the most critical vulnerabilities are usually located. For example, the incorrect checksum can be tested just once. Alternatively, the length of an array can be tested for corner cases, off by one, signedness errors, etc. Based on perceived knowledge about fuzzed message structures, the fuzzers could be divided into groups.

4.3.1 Fuzz data generation

Fuzzer test cases can be generated based on the input source, attack heuristics, and randomness. Because the fuzzers are used only for negative testing, the generated data should be semi-valid, meaning it contains some fault.

Four main approaches and their combinations are used to create semi-valid data:

- Test cases are created and hand-tuned for a specific protocol by an expert in the field. It mostly resembles traditional automated testing. Each test has a specific purpose. Many of the commercial fuzzers use this method.
- Cycling through protocol and inserting data incrementally and deterministically.
- Randomly inserting data for a specified period of time. This can be deterministic if the seed of the pseudo-randomness is configurable.
- Library of known errors is used.

4.3.2 Price of intelligent fuzzing

Creating more intelligent fuzzers comes at a price. It is significantly more time-consuming. The trade-off needs to be taken into account when creating a fuzzer. The focus on effectiveness, by design, also reduces the possible input set size, which could hide some errors.

4.4 Fuzzing process

The process of fuzzing includes a few steps. The fuzzer sends a sequence of messages to the SUT. Then the response and changes to the system are analyzed.

A typical response to fuzzing is one of the following:

- Valid response
- Error response – may be valid, if the protocol describes it as such
- Anomalous response – unexpected but nonfatal reaction (slowdown, corrupted message)
- Crash or other failures

Monitoring is a critical part of the fuzzing process. Its result should be a pass or fail. If one of the inputs causes a crash of the system, it can resolve itself by restarting. This means the monitoring must be constant. The failures should be noted and analyzed manually later.

4.5 Black-box and white-box fuzzing

The term black-box testing is used when testing is done without knowing the system's internal structure. The opposite of this is white-box testing, where the tester has access to all the source code and information about the tested system. Fuzzing was generally a black-box testing technique. Later, some information about system internals was used to help with testing. This is often called grey-box testing. Coverage guided fuzzing is one of the most effective tools that use grey-box testing. The coverage of the code is used to find new internal states of the target binary. This allowed general fuzzers to be used for effective fuzzing of arbitrary binaries.

The fuzzing with knowledge of the internal structure can still be considered white-box testing. John Neystadt, in the article *Automated Penetration Testing with White-Box*

Fuzzing [28] provides introspective how to use fuzzing as a white-box testing tool. For him, the fuzzing is used for:

> *White-box fuzzing or smart fuzzing is a systematic methodology that is used to find buffer overruns (remote code execution); unhandled exceptions, read access violations (AVs), and thread hangs (permanent denial-of-service); leaks and memory spikes (temporary denial-of-service); and so forth.*

White-box testing cannot be used every time. When the attacker does not have access to binaries or design of system internals, the only solution is to use black-box testing with public interfaces. Those public interfaces, however, are not always reliable enough and do not provide enough information. When this is the case, specific solutions must be applied to each one.

4.6 Case study

A case study on the fuzzing usage on small web-application (450 lines of code) done in the article *Automated Penetration Testing with White-Box Fuzzing* [28] shows the differences in the levels of intelligence and fuzzer knowledge about the system. The application was developed with four planned defects.

Technique	Effort	Code coverage	Defects found
Combination of black box + dumb	10 min	50%	25%
Combination of white box + dumb	30 min	80%	50%
Combination of black box + smart	2 hr	80%	50%
Combination of white box + smart	2.5 hr	99%	100%

Table 4.1: The results of the case study in *Automated Penetration Testing with White-Box Fuzzing* [28].

The results in the Table 4.1 confirm that with the increasing levels of intelligence and fuzzer knowledge, the percentage of the code covered by the tests increases. Also, the simple fuzzers which do not use any introspective, only random inputs can find a significant number of defects with a small amount of effort. This suggests that if the effort required for creating specially tuned fuzzers is too big, the random fuzzers can cover a relatively big portion of code and should be used.

4.7 Wi-Fi fuzzing

This thesis is focused on fuzzing Wi-Fi implementation of IoT devices, which do not have to have any other interfaces than Wi-Fi and whose source code can be secret. Fuzzing such devices is challenging. Not all wireless devices advertise themself, so there needs to be a reliable way to detect the tested device. Nearby devices can affect the test results or can crash by also receiving fuzzed frames. Some protection such as the Faraday cage should be used to limit the impact of the environment. Wi-Fi fuzzing is rarely used. The more notable accomplishments are mentioned further.

4.7.1 Johnny Cache and David Maynor

The first known use of Wi-Fi fuzzing was presented by Johnny Cache and David Maynor in their Black Hat USA 2006 talk *Device Drivers:Don't build a house on a shaky foundation* [9]. Around that time, the fuzzers were used to find many critical vulnerabilities in browsers and kernels [26]. The authors identified Wi-Fi as a potential target based on things, which are still valid with implementations of modern versions of Wi-Fi. The Wi-Fi is a complicated protocol, which is now, after another 15 years of development, even more extensive. This leads to some inconsistencies between the implementations, which could be used for fingerprinting the devices (can be used for targeting exploits to the level of firmware version). The complexity also leads to the big room for errors. The fuzzing space of their fuzzer was relatively small. They focused on the fundamental parts of the protocol.

The fuzzed information:

- Association redirection (association response from different MAC than the device tried to associate)
- Different Source, BSSID, or both
- Other unspecified fields

They found many critical vulnerabilities in multiple devices allowing remote code execution, even under root privileges. Even though, according to them, most often a direct return shell is not possible, bots and other malicious shellcode can be designed. The findings also say that a bug could be triggered by the frames sent a long time ago. The developed tool is not publicly available at the moment.

4.7.2 Laurent Butti

Laurent Butti and his colleagues followed the findings of Johnny Cache and David Maynor by creating the open-source fuzzing tool *wifuzzit* [6][8]. The details about the tool can be found further in the chapter with other tools (4.8.1).

Their work on the proposed fuzzer identified some limitations with fuzzing Wi-Fi:

- The resulting fuzzer should be fast. The scanning process requires channel hopping, and if the fuzzer cannot respond in time, the message will not be delivered.
- Another limitation is that we cannot be sure that the frame was analyzed by the driver, which could introduce false negatives.
- Some drivers accept beacon frames only if there are also probe responses.

The research also identified some interesting information elements to test:

- WPA, RSN (Security)
- WMM (Quality of Service)
- WPS (Wireless Provisioning Services)
- Proprietary IEs (Atheros, Cisco, ...)

The research continued after the talks mentioned [7], and the final results were multiple critical vulnerabilities. One of them was confirmed to be exploitable to remote code execution with the exploit available to the public through the Metasploit [33].

4.7.3 Mathy Vanhoeft

The author of the KRACK [1] and Dragonblood [2] exploits also notably contributed to the Wi-Fi fuzzing. Mathy Vanhoeft presented his Wi-Fi fuzzing research at the Black Hat USA 2017 conference. It was called *WiFuzz: detecting and exploiting logical flaws in the Wi-Fi cryptographic handshake* [35]. His goal was not only to expose common programming errors such as buffer overflows but to also expose logical vulnerabilities. One example is that some messages in a handshake can be skipped, causing it to use or negotiate an uninitialized cryptographic key.

The method of their research was to build a model of the Wi-Fi handshake, which describes the correct behaviour and then automatically generate invalid executions. The invalid handshake could be something like dropping some message, injecting messages, generating an invalid field, or switch encryption.

The results were quite surprising, when all twelve of the tested access points showed some irregularities. Not all of them were exploitable, but they at least showed some diversion from the standard. This can be used for fingerprinting. However, some of the diversions found can be exploitable with proof of concept available online [34]. The most prominent finding is that the OpenBSD client was missing the state machine of the 4-way handshake, leading to a trivial man-in-the-middle attack against it.

WiFuzz

The developed tool was called WiFuzz. It is a model-based fuzzer with a proven record. Unfortunately, the tool is not publicly available.

4.7.4 ASSET

The ASSET Research Group based in Singapore University of Technology and Design developed another promising Wi-Fi fuzzer called Greyhound [17]. The tool has some similarities to the WiFuzz. It also uses the model of the Wi-Fi protocol to find irregularities and deviations from the standard. The fuzzer speculates about the state of the tested device, then generates, mutates, sends a frame, and finally analyzes the response. Similarly to the WiFuzz, it can alter the order of the sent frames, send them more times, or change the fields inside the frame. The Greyhound uses probabilities in mutating the protocol layers based on discovered vulnerability reports, which should accelerate the fuzzing. According to the developers, the tool is a part of an unspecified commercial pen-testing tool and is not open-sourced [4]. Unfortunately, the tool was found too late to influence the design of a new fuzzer.

To date, the published results include three vulnerabilities with CVE numbers assigned to them and a claimed new Denial-of-Service attack. The newly claimed attack was found when the fuzzer was integrated into a commercial pen-testing tool. The vulnerabilities with the CVE numbers assigned were connected to the ESP32 and ESP8266.

[1]KRACK – https://www.krackattacks.com/
[2]Dragonblood – https://wpa3.mathyvanhoef.com/

4.8 Existing tools

During the research, a few tools used for Wi-Fi fuzzing were found. Some tools advertise that they can fuzz Wi-Fi, but in reality, they test IP frames through a wireless network. The tools were examined and tested, but they were found not to be appropriate for further development for the needs of the thesis. Only open-source projects were included in this list. The previous section lists other projects. Even though their results may be better and are more advanced, they are not publicly available under open-source licenses.

4.8.1 Fuzzers

cfuzz

cfuzz is the result of a more recent open-source effort with capabilities similar to the goals set out in this thesis. It was made as a Bachelor thesis by Bart Pleiter at Radboud University [30]. The fuzzer is written in C language. It should be capable of fuzzing probe response, authentication frames, and association response frames. According to the thesis results, it was successfully used to discover a vulnerability in Nintendo DSI XL.

The source code had problems with compilation. The code had to be significantly modified to allow correct linking of the compiled targets. The inspection of the code revealed that the interface is not ready to be easily extended.

wifuzzit

A Wi-Fi fuzzer *wifuzzit* has been used for similar purposes as the intentions of this thesis. The fuzzer is open-source and capable of finding vulnerabilities, as documented by the list of discovered vulnerabilities on the GitHub page [3]. It was developed by Laurent Butti, who also presented the work on BlackHat Europe 2007.

The main reason not to use this program is its age. It was developed in 2007 in Python2, which is now not supported. The fuzzer used a patched version of the Sully fuzzing framework. The patches are no longer compatible, and Sully is also not maintained anymore. The effort to make the fuzzer work under a modern system with some support would be greater than the rewrite to Python3 with a maintained fork of the Sully framework. This fork would have to be patched again to add support for frame injection to wireless interfaces.

Another reason not to use this solution is its performance. The use of a Python interpreter means the execution is slower than it could be. The author himself sees it as a drawback [6]. Wi-Fi fuzzing is time-sensitive. For future extensions and more advanced operations, the performance of the Python program does not have to suffice.

4.8.2 Frameworks

Scapy

Scapy [4] is a popular framework as the authors say able to: *"forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more."* The framework is easy to use, able to craft Wi-Fi semi-valid frames for fuzzing with minimal effort. It is even recommended by one of the authors of the first public wifi

[3]wifuzzit – `https://github.com/0xd012/wifuzzit/`

[4]Scapy – `https://scapy.net/`

fuzzing, David Maynor, in the article *Beginner's Guide to Wireless Auditing* [24] However, its major drawback is that it is slow, which can cause problems in fuzzing Wi-Fi.

BooFuzz

BooFuzz [5] is a fork of the now unmaintained Sully open-source Python framework with some improvements. According to documentation, its main selling points are easy data generation, extensible failure detection, and recording of test data. The SUT can be reset by the framework in case of a crash. Even low-level network protocols as Ethernet, IP, or UDP are supported. Unfortunately, 802.11 support is not present.

4.9 Summary

The Espressif Systems company is interested in the improvement of their testing procedures. Fuzzing would be a convenient method for this goal. From the research done, we concluded that the existing solutions are not suitable. The existing solutions are not available under an open-source licence (WiFuzz, Greyhound), are not usable (cfuzz), or the potential to be extended is small (wifuzzit). This means a fuzzer must be designed and implemented, which is described in the following chapters.

[5]BooFuzz – `https://github.com/jtpereyda/boofuzz`

Chapter 5

Design

In this chapter, the design decisions taken during the development of the fuzzer will be discussed. Designing a good fuzzer is not easy. It should not rely too much on the knowledge of the fuzzed protocol or make too many assumptions. The designer could be making the same assumptions as the vendors implementing the devices, resulting in not discovering some vulnerabilities. Nevertheless, it also must use some heuristics to make the fuzzing time feasible.

5.1 Design Requirements

During the research, a few of the existing solutions and prototypes were tried, leading to the establishment of the requirements in this section.

Performance requirements

During testing, we validated a claim that some devices have a threshold under which they must receive the response to their requests [6] [30]. This threshold can be hard to maintain using interpreted languages. For this reason, we set for ourselves the goal that we must be able to respond to a request in under 0.01 seconds. This threshold was found to be a realistic boundary for the examined real access points. The measurements can be seen in performance testing in Table 7.1.

Autonomous detection of vulnerabilities

The fuzzer aims to be configured once and be run without intervention, so vulnerabilities must be detected autonomously. Not all of them can be detected this way because of the nature of fuzzing. It only detects vulnerabilities, which cause significant side effects.

Reproducible tests

The fuzzer can run for long periods of time. When the program finds input that uncovers vulnerability in SUT, the program must be able to repeat the inputs.

Vulnerability logging

Reproducible results are related to another requirement – the logging of the results. It would be counter-productive to log all frames sent, so there needs to be a way to log detected vulnerabilities together with the inputs, which lead to their discovery. It may be possible that it was triggered before detection, so we need to log a few of the last frames.

Minimizing interference

Due to the nature of wireless networks, we need to be able to minimize interference from other devices. Unless the devices are in a Faraday cage, other devices in proximity may be using Wi-Fi and alter the results. This means the resulting program must be able to filter out traffic other than that from SUT.

Configurability

Parameters that make sense to be configurable must be configurable without recompilation of the code. This includes:

- type and subtype of frames, which should be fuzzed
- SUT MAC address
- sensitivity of vulnerability detection
- amount of logged data
- seed for pseudorandom operations

Modularity

Some aspects of the solution may change with time or with the tested devices, such as the monitoring tool used. That is why the program must be flexible enough to be able to change modules without big changes to the rest of the system.

We decided to focus the testing on the ESP32 platform. This dictates the need to integrate ESP32 monitoring. The coupling of the ESP32 monitor, however, should not be mandatory.

5.2 Structure

From the requirements stated, we designed a modular program with the following modules:

- Core – registering other parts, data handling, primary communication with SUT
- Frame fuzzer – creating fuzzed frames, this can be in response to some request received, or just by internal state
- Monitor – monitoring vulnerabilities, can communicate with SUT by secondary channels

5.3 Fuzzer core

The core is the central part of the fuzzer. It connects individual parts to each other and to the SUT. It also parses configuration and starts other modules. We made a decision to use *libpcap* [1] for communication with SUT. The decision was based on the experiments with Wi-Fi injection in Python done in the bachelor thesis *Fuzzing Wi-Fi in IoT devices* [30], which were repeated and confirmed. The solution implemented in Python did not conform to the performance requirements. The libpcap was chosen thanks to its interface and low-performance penalty.

The individual parts (frame fuzzer, monitor) each have a specific interface. This means they can be easily replaced by a different implementation, which is used when choosing different frame types to fuzz.

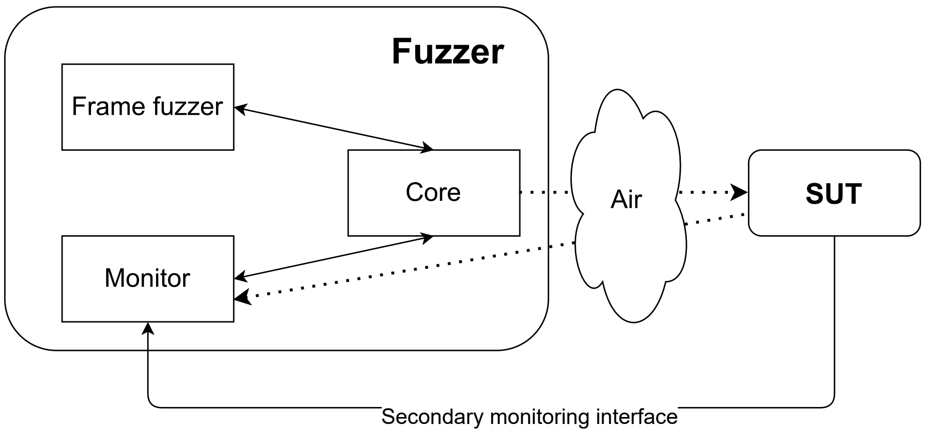


Figure 5.1: Diagram for newly designed Wi-Fi fuzzer.

Some of the frame subtypes should be sent as a response. Others are sent without any external action. That is why those two cases need to be separated and configured by themself.

Push fuzzing

The simpler case is when the fuzzer sends packets without external prompts. We call this case *push fuzzing* because it only pushes the produced frames to the wireless interface. The fuzzer requests new semi-valid data from the frame fuzzer and then sends them using the specified interface. We do not have guarantees of the frame reaching the fuzzed device. Real devices try to mitigate this problem by sending the requests multiple times preemptively, even before the acknowledgement could be delivered. This increases the likelihood of at least one frame reaching the device. The user should be able to configure this behaviour. Also, the Wi-Fi drivers modify the behaviour, which cannot be configured without changes in them (at least the tried Realtek [2] drivers and Intel drivers). The sent data should be logged for the purposes of recreating potential found vulnerabilities. The logging is provided by the monitor, and the sent data is passed to it. The next frame should not be sent right after the previous, and the waiting period should be configurable. When the frame fuzzer

[1]libpcap – `https://www.tcpdump.org/`
[2]Realtek Aircrack drivers – `https://github.com/aircrack-ng/rtl8188eus`

exhausts its pool of possible generated frames, the core should gracefully shut down the rest of the system and provide fuzzing results and detected abnormalities.

Response fuzzing

The response type fuzzing should listen on the specified interface for request from the tested device. When the appropriate request from the appropriate device is detected, the core should request the data from the frame fuzzer and log it using the monitor. The next request should be sent only when another request is detected.

5.3.1 Set-Up and Tear-Down

Some vulnerabilities found by previous works revealed that the Wi-Fi state of the device (authenticated, associated) could affect whether the fault is accessible. This is why there should be a way for the fuzzer to get the device to a specified state before sending the crafted frame and then, after the test was performed, return it to a resumable state. In testing, this mechanism is generally called *set-up* and *tear-down* functions, and the fuzzer core should allow the test to utilize them. Illustration of such mechanism is in Figure 5.2.

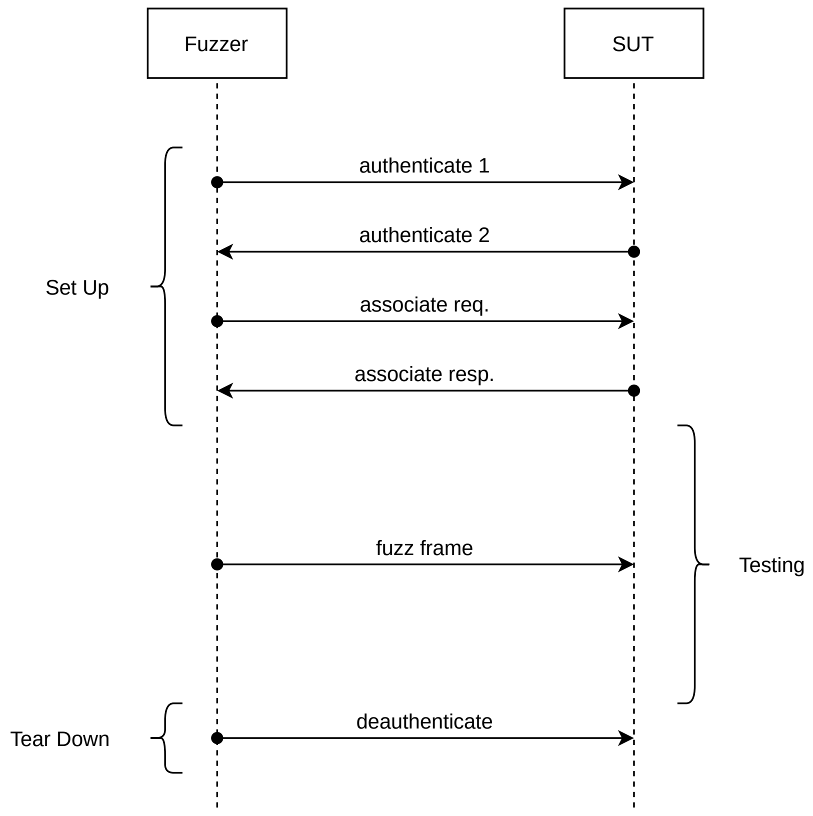


Figure 5.2: Diagram showing example of the initial Set-Up phase associating the device to the fuzzer, then fuzzing itself and then Tear-Down phase – deauthentication.

5.3.2 Communication with frame fuzzer

The communication with the frame fuzzer could be two-way. At a minimum, the frame fuzzer needs to know the MAC address of the fuzzed device. This can be provided during the initialization. If the need arises, the request must be possible to pass to fuzzer for the analysis or response generation. One of the possible applications would be fuzzing an encrypted network or authentication.

5.3.3 Communication with monitor

The communication between frame fuzzer and monitor should not be direct. It is possible that in some cases, the monitor has information about the internal state of the SUT from the probes (6.2.5). This information, however, is not likely that it would fundamentally help with the data generation. When the need arises, the monitor informs the core, which decides the best course of action and which data generation source for subsequent frames should be used.

5.4 Monitor

For purposes of the thesis, the tested platform can be every device with Wi-Fi networking implemented. We could be testing any client or access point since we are implementing a black-box fuzzer. The problem with the completely black-box approach with the generic interface is inconsistent feedback from the system under test (SUT). It cannot be reliably monitored just by Wi-Fi. If we use other interfaces, they must be tailored to a specific device. We chose to focus mainly on testing an ESP32 microcontroller development board. But the design of the fuzzer should be flexible enough to allow fuzzing of other devices.

5.4.1 Universal vulnerability detection

When a frame is sent through the Wi-Fi, it can be considered undelivered until an ACK is received. But this has a few caveats. For example, with beacon frames, it is impossible to know if the SUT received the frame. Even under relatively controlled conditions, we do not have a way to know if a frame was received and parsed unless we have some external monitoring tool installed on the device.

Acknowledgement monitoring

This means the only way to detect any malfunction in the device by Wi-Fi is by checking for their responses. The simplest way is to set a threshold. For example, if the device does not respond to 10 frames in a row with ACK, we can suspect the device is malfunctioning. This approach was used by *cfuzz* [30]. The problem with this approach is the structure of the ACK frame. It does not contain the frame source, only the receiver. We cannot be sure that nearby devices did not respond to the fuzzed frame. This can be problematic, for example, in beacon frames, which are broadcasted to every listening device. For this reason, the resulting fuzzer does not provide monitoring based on this approach.

Passive monitoring

Another way to detect malfunction is to have a different threshold for time without any communication. A typical device connected to a network should be sending and receiving some frames within a reasonable timeframe. When a device sends some frame containing its MAC address as a source, we can assume the device is still working properly, or at least the damage did not cause any denial of service. This approach can be called *passive monitoring* because the monitor only passively listens to communication around it and does not actively send any frames. Alternatively, similar solutions are called heartbeat signal monitoring.

Active monitoring

The last option is to pick a valid frame to which the device should respond. We can call this approach *active monitoring*, as the monitor actively participates in communication with SUT. Theoretically, one candidate for the frame type could be a request to send. This, however, has the same problem as the acknowledgement frames – the response does not contain the sender. Also, from our trials, not every device responds to such requests. For access points, the probe response frame is a good candidate. For stations, we did not find a suitable frame type.

5.4.2 ESP32 monitoring

The fuzzing can be improved by providing some insight into the inner workings of the system under test. The wider the range of the targeted devices, the less they have in common. If we focus on a specific set of devices, we can create more accurate monitoring tools.

The ESP32 provides a serial interface through UART, which could be used for reliable communication with the fuzzing program. This interface also allows us to estimate its internal state closer. The ESP32 can be configured to output verbose logs which contain the state of the Wi-Fi module (initialized, authenticated, associated) and changes between them (initialized $\rightarrow$ authenticated).

The IoT devices generally try to minimize power consumption by reduction of the executed instructions to the bare minimum. This leads to disabling security features as stack and heap smashing protection. The ESP32 provides those protections. During testing, we do not need to look at the power consumption, so they can be activated. They provide additional information about possible unwanted stack or heap modifications. Based on the previous data, buffer overflows are one of the most prominent critical implementation bugs, so it is important, we have the potential to monitor them.

Remote probes

We focus mainly on the ESP32, but the final fuzzer must be ready to fuzz this wide variety of devices. The solution we used is to create a distinct probe for every supported device and then communicate remotely with the monitor used in fuzzer. This probe can be as complicated as the specific user needs. When the probe detects malfunction or state change, it notifies the monitor, which then executes appropriate action. The used approach allows for flexibility, a wide array of possible use-cases and implementations. A special monitor connected to the fuzzer must complement this probe.

5.4.3 Logging

The logging is an important part of the resulting fuzzer. It should notify the user about detected anomalies and the frames, which potentially caused them. The identification of the exact frame, which caused the anomalous response is likely impossible during the fuzzing itself. Also, it can be caused by a stream of frames. Logging should store a certain number of the previously sent frames, which should be written to more permanent storage when the anomaly is suspected.

5.5 Frame fuzzer

Model-based, protocol-specific fuzzing will be used to create semi-valid frames. The fuzzing will be divided into specific management frame subtypes (probe response, authentications) done by a specific fuzzer.

The decision to use model-based fuzzing came from various requirements. The totally random fuzzing in itself is highly inefficient. To approximate the semi-valid communication to the real one, it must include some delay between sending the requests. For example, the time from sending the first authentication frame to sending the successful association response to the open access point is approximately 0.05 seconds. This would limit the number of association frames the devices could receive from the fuzzer.

The model-based approach was chosen because it utilizes the only information available about the tested device. We do not know anything about the executed code or about the design of the Wi-Fi stack. This somewhat limits the potential of the fuzzing, but it also allows to test a wide range of devices when the only thing they have in common is the Wi-Fi protocol.

5.5.1 Fuzzing approach

The goal of the resulting fuzzer is to find major implementation flaws, which will heavily alter the functionality of the tested device. As such, the priority was on finding vulnerabilities already identified by other means in other projects. For example, the historically very prevalent flaws are connected to buffer overflows or string copying.

The buffer overflows could be triggered by specifying a different length of the field than the real length. The string vulnerabilities are mostly caused by improper usage of functions like `strcpy`, `strcmp`, or `strlen`. Unexpectedly long frames could also cause some problems. Some vulnerabilities found in devices were caused by the changed order of the sent frames, or the frames arrived at the unexpected phase of association. We concentrate more on the fuzzing of the frames themself.

Even the model-based fuzzing does not guarantee sufficiently small fuzzing space. Some vulnerabilities could theoretically be found only by specific order of the fields with the specific values. This is, however, still too broad space to fuzz practically. To avoid the exponential state explosion, we can fuzz only one specific feature at a time. Hopefully, the reduced space will contain most of the vulnerabilities.

Advertisement subtypes fuzzing

While fuzzing the tag parameters, in the frame subtypes used for the advertisement of the access point and its parameters, like the probe response, or the beacons, the frame will contain some valid parameters. During testing, when contained parameters were not sufficient, the tested devices dropped the frames without looking at the content (the logs contained the message about missing information elements, even if the elements would be parsed and checked later, we assume the added valid information elements do not influence the fuzzing). The frame must still contain at least the parameters absolutely needed for correct operation. The fuzzed parameter will be at the last place, before the checksum. The reason for this is, when we declare the wrong length of the field in the first place, all that happens is, the data behind it will be interpreted as the type of the next tag parameter. This assumption is based on the fact that its correct interpretation is required for the basic operation of the device and other checks are not likely to be needed.

String fuzzing

Character strings are a part of the Wi-Fi protocol, and often, in other applications, they are a common source of vulnerabilities. Typical causes are improper uses of string functions (in C language `printf`), where the user-provided string is passed as the first argument, which is a format string. The vulnerability can be triggered by providing format strings such as „`%s"`, which could be used to read from the memory or „`%n"`, which could be used to write to memory. The copying without bounds checking (using functions as `strcpy`) could be revealed by providing strings containing null byte such as `"aaa\0aaa"`.

The device might allow using UTF-8 or even old UTF-1 encoding. The wast variety of the characters and their proper handling proved to be challenging, which is proved by some known vulnerabilities like the CVE-2015-1157 in iPhone [11].

Length fuzzing

The various fields in the frames are usually preceded by their length. A buffer overflow may happen when the length of the field is not correct and if the length check is not correctly implemented. This has caused many issues in the past and should be part of fuzzing.

The time complexity of trying every possible length combination is too high. For this reason, the fuzzing is divided into a few levels based on the fuzzed coverage. The simplest set is designed to be the most effective, with only the boundary values included. For example, the supported rates parameter in beacon frames should have a length at most 8, and its length is stored in 8bits. The lengths tried during fuzzing should include: 0, 1, MIN-1, MIN, MIN+1, MAX-1, MAX, MAX+1, 127, 128, 253, 254, 255

- 0 – invalid length, the field is not pointless
- 1 – could potentially reveal an off-by-one error
- MIN-1, MIN, MIN+1, MAX-1, MAX, MAX+1 – the boundary checks
- 127 – greatest signed number in 8bits using the two's complement
- 128 – smallest signed number in 8bits using the two's complement
- 255 – greatest unsigned number in 8bits

5.6 Summary

The designed fuzzer is modular, with all the parts replaceable. It employs the set-up and tear-down procedures common from the classic software unit testing. In contrast to other fuzzers, the newly designed fuzzer can use secondary channels to communicate with the tested device in addition to the Wi-Fi interface. The design of the data produced took into account previously found vulnerabilities, common errors and other problematic areas. Testing will be reproducible thanks to the deterministic data generation and configurable seed for semi-random data.

Chapter 6

Implementation

The implementation of the fuzzer is in the language C++. The standard version of the language is specifically C++20. The implementation is divided into various libraries and programs, which together make the final testing tool.

6.1 Fuzzer Core

The core of the fuzzer is used to couple different parts of the system, configure and lunch them.

6.1.1 Configuration

The parser parses the configuration file during startup, stored into config classes for each of the components. Parsing user input is error-prone, so the availability of YAML parsers can help signifficantly. The one used in this fuzzer is called *yaml-cpp* [1]. During startup, the configuration file is parsed by the parser, which is then stored into config classes for each of the components. The options include the selected monitor, frame subtype, or period between sending frames. Specific options and format can be found in the project README.md file [2]. The example of configuration used for fuzzing the beacon frames on the ESP32 can be seen in Figure 7.2.

6.1.2 Sniffing

For the frame capturing, the function `pcap_next` is used. This function takes one frame at a time from the captured queue and processes it. From testing, this solution seems to be sufficiently fast, and the delay was not an issue. The advantage of this solution compared to the `pcap_loop` callbacks is more straightforward interfacing with the rest of the fuzzer.

6.2 Monitor

The requirement for different monitor options resulted in the base `Monitor` interface, which is implemented by different final monitors.

[1]yaml-cpp library – https://github.com/jbeder/yaml-cpp/
[2]wifuzz++ – https://github.com/xvenge00/fuzzer-dp/

6.2.1 Watchdog

A resettable watchdog was implemented for the purpose of keeping track of the time since the last known state of the SUT. It is notified every time the monitor detects activity in SUT, which resets the countdown to the value specified during initialization. When this silent period expires, the watchdog runs a specified function, which is a notification for the monitor, that the SUT is likely inoperative. This period must be carefully chosen. A too short period means lots of false positives, but a period too long gives the SUT chance to reboot and be operational again before the detection takes place.

After the watchdog expires, it can be reset again by notifying it again. This allows the watchdog to be reused and to detect other malfunctions after the device was reset.

6.2.2 Notification monitor

The notification monitor is a type of monitor that relies on external calls to notify itself about the activity of the SUT. It is useful when the fuzzed frame subtype is a response. The fuzzer core must listen to traffic from the SUT anyway, so it can notify the monitor when doing so. It includes a watchdog for guarding the silent period.

6.2.3 Passive monitor

The other subtypes of frames do not require listening to the incoming traffic. During fuzzing of those frames, the passive monitor can be used. It listens on the specified interface, which does not have to be the same as the one sending fuzzed frames. When a frame that is from the SUT arrives, it resets its watchdog. Other functionality is identical to the notification monitor.

6.2.4 Probe monitor

The monitor used when the SUT is connected to the remote probe. The communication between the probes and the monitor is implemented with the gRPC [3] framework. The choice for this framework comes from its flexibility. It can be implemented in a wide array of languages and can send a wide array of data. In the future, when it is required, the probe can be easily expanded to send complicated data structures, which can be used for more effective fuzzing. It does not include a watchdog. Instead, it relies on the probe to tell the monitor about the state of the device.

Not every use-case requires external probes. When the functionality is not needed, the program can be compiled without the support for the gRPC based monitor and then it does not require them as dependencies.

[3]gRPC framework – https://grpc.io/

6.2.5 Remote probe

The remote probe for ESP32 is implemented as a separate Python application. The gRPC provides a compiler to the Python language. It also uses pyserial to deserialize the output from ESP32. Communication is only one way. The output is used to get information about the state of the ESP32. It is checked against a list of phrases, which indicate the malfunction. It was created from the documentation of the ESP32 API and independent testing. The list includes the following words:

- panic
- dump
- exception
- watch
- triggered
- corrupt
- failure
- protect
- reboot

6.3 Frame fuzzer

The creation of the frames is handled by the frame fuzzer. To make the creation of the new fuzzers easier, it uses the new feature of the C++20 – the coroutines. They are used in generators, which replicate the generator functions found in Python. The generator implementation is based on the article *From Algorithms to Coroutines in C++* by Kenny Kerr [22].

At the time of writing, the coroutines are not considered to be a very mature feature of C++. From the main three compilers, only GCC and Visual Studio does not consider it to be experimental. Even then, they require special options during compilation. In GCC, the compilation itself is not always hassle-free, and some versions of the compiler produce internal compiler errors. The performance penalty in real use in the application was not measurable, even though some overhead is required. Their use, however, has compelling advantages. The generators better represent the modelled logic of the fuzzing. They significantly sped up the development process and reduced the room for errors. Without them, the internal states of the fuzzers would have to be manually saved and loaded during each invocation or passed by other means to the producing function. This in itself distinguishes the fuzzer from similar products.

Similar model fuzzers are mostly written in interpreted languages like Python, which allow for those kinds of abstractions. For most applications, their performance is sufficient. However, for some applications, like fuzzing Wi-Fi or Bluetooth wireless protocols, it can be limiting, and better performance is needed. The resulting data-producing library is separated from the rest of the fuzzer so that it can be reused. The data is produced through a unified interface. The library provides useful components, which can be useful outside of the Wi-Fi fuzzing, such as fuzzing strings or general fields.

The fuzzer, for now, implements: probe response, beacon, deauthentication, disassociation and authentication frames fuzzing.

6.4 ESP32 test programs

We want to test the implementation of the ESP32 Wi-Fi stack. This is difficult to perform because we do not know what portion of the code we tested. The functions used for one operation (for example, active scanning) may use the same implementation as the same operation in another context, or it may not. We do not know unless we try to reverse engineer the proprietary library implementing the functionality. Also, compared to testing devices as access points, whose software is always running and the device always communicates with its surroundings, the applications for testing ESP32 Wi-Fi stack needs to be specially written. This brings some challenges. We could test some real application, which tries to connect to the access point. However, the testing would most likely discover bugs in the application itself and its logic, not in the Wi-Fi stack. A better way of testing the Wi-Fi implementation is to isolate the steps and only provide minimal software overhead. For this, special test programs had to be written. The software would only do one specific thing until something causes the reset of the device. For example, when testing the beacon frames, the device should listen for such frames and then and try to parse them. Further details are specified in each case.

6.4.1 Probe response test

The probe response frames are used for active scanning. The station device sends a probe request with some details about its capabilities, and the access point responds with details about its capabilities. The probe response can contain a specific SSID. The stated test cases should test only the active scanning.

In testing, the probe request from the tested device does not contain any SSID. It is done by using the `esp_wifi_scan_start` function with the scanning method set to active scanning. The channel is set to the specific channel, the same as the channel of the fuzzer. The ESP32 then prints the SSID and some information from the received probe response frame, if it was accepted. The sent frames are most of the time not valid, and the firmware drops them, as it should.

To simulate the process of connecting to specific access point, another test using the probe response was created. It tries to connect to the access point with a specific SSID, which is the SSID in the fuzzed frames (when the SSID is not fuzzer at the moment). The reason for it is that it may internally use different implementation than active scanning by itself, and this use-case is more common. Under connection is specifficaly meant the process of discovery, authentication, association and assignment of IP address. The fuzzer responds to every probe request from the tested device. It does not look at the SSID provided.

6.4.2 Beacon frame test

The beacon frames are used for advertising the presence of the access point. This can be then used to initialize the connection. In contrast to the probe response frames used for active scanning, the beacon frames are not responding to any frame. The access point sends them without any prompts. This passive scanning can be tested by a similar program to the active scanning. The difference is in the scanning method, which is set to passive scanning.

The test program that tries to connect to the falsified access point cannot be used. The connection only accepts the probe response frames. It does not try to associate itself with the access point when only the beacon frames are available.

6.4.3 Deauthentication and disassociation frame test

The mentioned frame types and the problems with fuzzing them are similar, so they are categorized as one. Creating a test program for fuzzing the two frame subtypes is not as straightforward as previous frames. The deauthentication frames by themself are simple. However, the states in which the device may accept them are numerous. From the specification of the 802.11, the deauthentication frames should be accepted in any state. This would mean that any test application with the Wi-Fi library initialized should always react the same. Whether in the initialized, authenticated, or associated state, when active or passive scanning. We cannot prove that this is the case in the real implementation, so the test cases include all of the mentioned states.

6.5 Test Runner

To effectively run tests on the ESP32, a bash script was written. It can compile and flash the prepared source files to the ESP32, run the remote monitor probe, and run the fuzzer with the specified configuration. The script is flexible, and the various parameters allow to specify the location of all the required files. The build and flash to the ESP can be skipped if the user knows the correct application already runs on the chip. Also, the monitor probe for the ESP32 can be disabled when the user wants to monitor the serial communication by themself or when other monitoring is wanted.

6.6 Summary

The implemented fuzzer follows the design decisions outlined in the Chapter 5. Together with the fuzzer, a remote probe for the ESP32 monitoring and test runner with test programs for the ESP32 were implemented. The latest language features were used to make the extensibility easier.

Chapter 7

Testing

The testing is divided into multiple parts. The first should confirm the fulfilment of the requirements set in the beginning and during the design stage, including the functional requirements. The second should be testing real devices using the implemented tools.

7.1 Design requirements

The design requirements are covered mainly by the general decisions while developing the fuzzing tool.

- The **reproducibility** of tests is ensured by the deterministic data generation.
- **Vulnerability logging** is performed by the monitor.
- **Minimizing interference** should be done by only reacting to the frames with the source address set to the SUT MAC.
- The resulting fuzzer can be **configured**.
- It is **modular** on more levels. The remote probes can be changed for more sophisticated ones. The crucial parts of the fuzzer can be replaced thanks to generic interfaces, and the fuzzer generation can also be extended without major changes.
- The **autonomous vulnerability detection** is hard to verify when the fuzzing did not find any vulnerabilities. However, the conditions can be simulated. The test application for ESP32 was written, which caused the watchdog to reset the device. The result is the fuzzer detecting reset by watchdog and saving the last sent frames. The same can be said about the monitors using the watchdog.
- The **performance requirements** were also satisfied. The details about testing are presented in the next section.

7.2 Performance requirements

The performance requirements were tested by running the probe response test and measuring the time between captured probe requests and the first probe response. The data were filtered to include only data from the measured devices. The measurement was done by a third device, as shown in Figure 7.1 that acts as an independent observer.

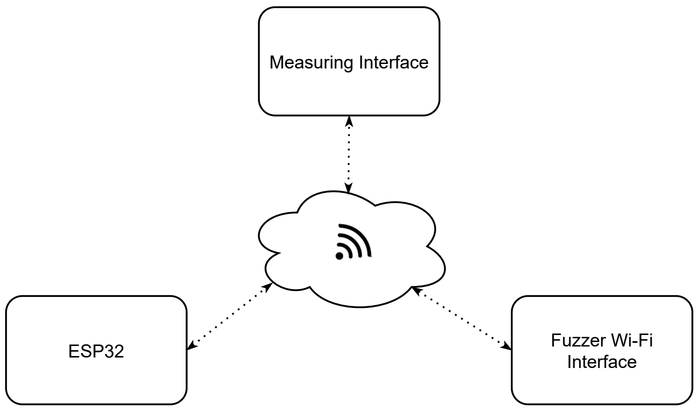


Figure 7.1: The diagram is showing the physical configuration of the participating devices and the measuring device. They were all located approximately 30cm from each other.

The measurements were compared to the real access points *ZTE Home Gateway Speedport Entry 2i*, TP-Link Archer C6 and software access points created by android phones and windows laptop. The reference measurements were done by mimicking the previous setup with the Samsung Galaxy S9 acting as the client and the access points in the place of the fuzzer. The results were processed by sorting and removing the top and bottom 10% of the measured intervals. They show that the time to respond of real access points is around 0.003 seconds, which can be seen in Table 7.1. The measured software access points vary more. The results also show that the implemented fuzzer is sufficiently fast and the requirement of response time under 0.01 seconds is satisfied. All of the response fuzzing is based on the same implementation, which means the measured performance is representative of all the fuzzed response frames.

Device	average time to respond (s)	max time to respond (s)
ZTE	0.003	0.003
TP Link	0.003	0.003
Galaxy S9	0.009	0.010
Huawei P30 lite	0.002	0.006
Windows laptop	0.004	0.005
Fuzzer	0.007	0.009

Table 7.1: The results from testing the time to respond to probe requests on various devices. The data was filtered from obviously outlying measurements.

7.3 Functional testing

To test the functionality of the fuzzer, Wireshark [1] was used. The tool allows for visualization of the sent and received frames. Two wireless interfaces were used. The first was used by the fuzzer, and the second, independent, interface was used for traffic monitoring to ensure the sent frames contained the data we expect. When the format of the captured frames is not according to the specification, the Wireshark marks them as malformed. We assume

[1]Wireshark – https://www.wireshark.org/

the Wireshark is able to capture and display the data correctly. It is tried and tested software, and if the parsing would be wrong, the hexadecimal representation of the frame could be checked. Before the experiment starts, the interfaces must be configured appropriately. They have to be in the monitor mode, which allows reading all of the frames that reach the wireless interface. This also allows for frame injection using certain drivers. The interfaces also have to be on the same channel.

7.3.1 Beacon frames

To test the correct injection of the fuzzed beacon frames, the fuzzer was configured with the configuration seen in Figure 7.2.

```
---
fuzzer_type: "beacon"
interface: wlp7s0f4u2u2
random_seed: 420
src_mac: "8c:dc:02:d3:28:1f" # zte router
test_device_mac: "3c:71:bf:a6:e6:d0" # ESP
channel: 5
set_up: "null"
tear_down: "null"
fuzz_random: 10

monitor:
  frame_history_len: 20
  dump_file: "/home/user/dump"
  type: grpc
  server_address: 0.0.0.0:50051

controller:
  wait_duration_ms: 100
  packet_resend_count: 3
```

Figure 7.2: The fuzzer configuration for testing the ESP32 system by sending the beacon frames. The wait duration was chosen based on testing done by modifying the data generation and manual inspection of the scan output, in order to make sure all the frames were scanned successfully.

The produced frames should be broadcasted and received by nearby devices. The reception depends on the physical properties of the signal, interference and distance from the fuzzer interface. The correct parsing and displaying of the frame depends on the Wi-Fi implementation of the receiving device. The expected behaviour is following:

- The Wireshark sniffing through the independent interface will capture almost all the frames sent.
- The devices nearby will display some of the SSIDs present in the frames sent.

The real findings exactly follow the expected behaviour. The output of the ESP32 system can be seen in Figure 7.3, and a screenshot from the Android phone scan is shown in Figure 7.4. Figure 7.5 shows a more detailed look at the frames sent by the fuzzer.

```
I (8790) scan: Total APs scanned = 2
I (8790) scan: SSID   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
I (8790) scan: RSSI   -20
I (8790) scan: Authmode  WIFI_AUTH_OPEN
I (8790) scan: Pairwise Cipher  WIFI_CIPHER_TYPE_NONE
I (8800) scan: Group Cipher  WIFI_CIPHER_TYPE_NONE
I (8800) scan: Channel   5
```

Figure 7.3: A sample output of the ESP32 through the serial interface during the functional test of the beacon fuzzing. The program running on the system is passive scanning the channel 5 and listing the found networks.

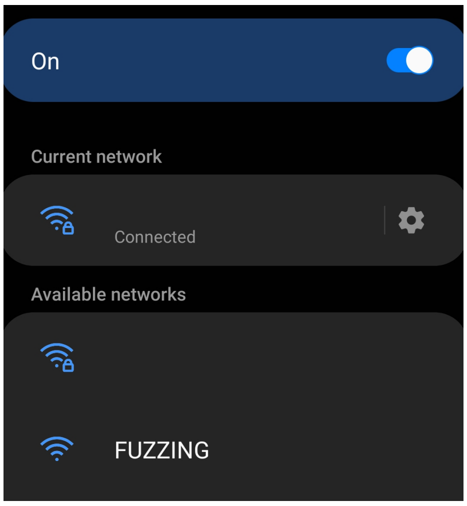
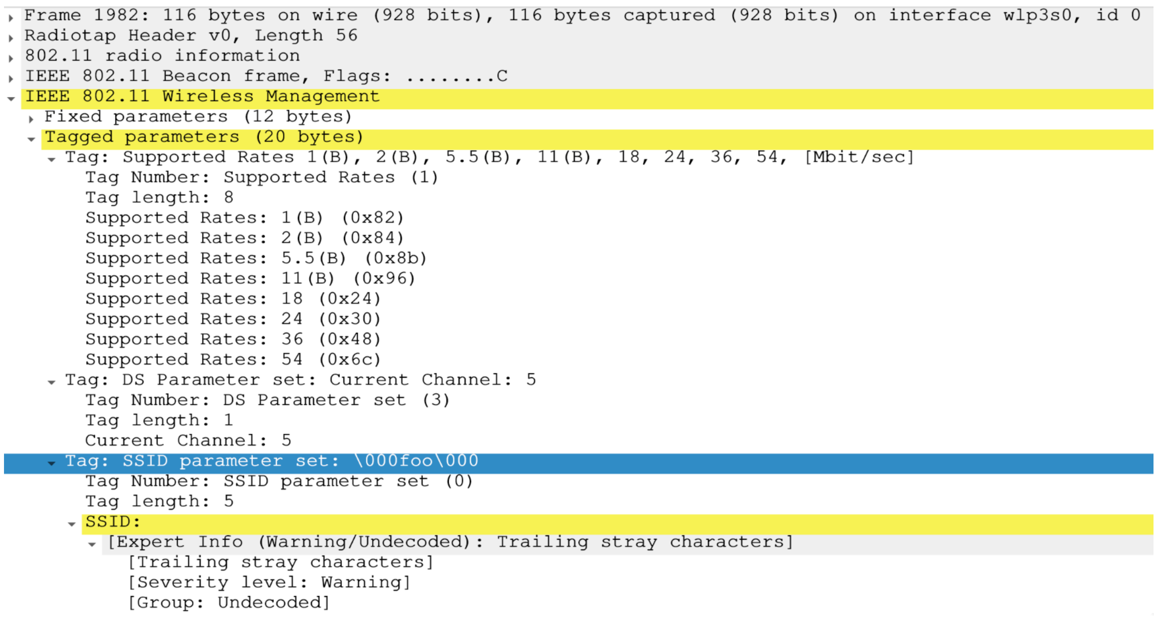


Figure 7.4: Screenshot from the Android phone scanning for the Wi-Fi networks during fuzzing beacon frames. The found networks include the SSID from the frames sent by the fuzzer.

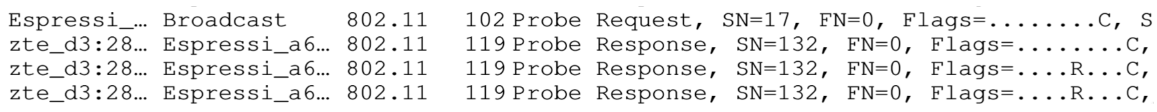
7.3.2 Probe response

Similar setup to the beacon frame fuzzing is required for the probe response fuzzing. The difference is that the response needs to receive a probe request frame from the SUT, and the frame is sent with the destination set to the tested device MAC address. In the configuration, only the `fuzzer_type` is changed to the `"prb_resp"`. The tested application was also changed to perform active scanning instead of passive. Wireshark can capture both the requests and the replies. This is shown in Figure 7.6. It includes the expected source and destination MAC addresses, The ESP32 output looks the same as in Figure 7.3.



```
▸ Frame 1982: 116 bytes on wire (928 bits), 116 bytes captured (928 bits) on interface wlp3s0, id 0
▸ Radiotap Header v0, Length 56
▸ 802.11 radio information
▸ IEEE 802.11 Beacon frame, Flags: ........C
▾ IEEE 802.11 Wireless Management
  ▸ Fixed parameters (12 bytes)
  ▾ Tagged parameters (20 bytes)
    ▾ Tag: Supported Rates 1(B), 2(B), 5.5(B), 11(B), 18, 24, 36, 54, [Mbit/sec]
        Tag Number: Supported Rates (1)
        Tag length: 8
        Supported Rates: 1(B) (0x82)
        Supported Rates: 2(B) (0x84)
        Supported Rates: 5.5(B) (0x8b)
        Supported Rates: 11(B) (0x96)
        Supported Rates: 18 (0x24)
        Supported Rates: 24 (0x30)
        Supported Rates: 36 (0x48)
        Supported Rates: 54 (0x6c)
    ▾ Tag: DS Parameter set: Current Channel: 5
        Tag Number: DS Parameter set (3)
        Tag length: 1
        Current Channel: 5
    ▾ Tag: SSID parameter set: \000foo\000
        Tag Number: SSID parameter set (0)
        Tag length: 5
      ▾ SSID:
        ▾ [Expert Info (Warning/Undecoded): Trailing stray characters]
            [Trailing stray characters]
            [Severity level: Warning]
            [Group: Undecoded]
```

Figure 7.5: Screenshot from Wireshark showing one of the frames sent by the fuzzer during SSID fuzzing.



```
Espressi_… Broadcast    802.11   102 Probe Request, SN=17, FN=0, Flags=........C, S
zte_d3:28… Espressi_a6… 802.11   119 Probe Response, SN=132, FN=0, Flags=........C,
zte_d3:28… Espressi_a6… 802.11   119 Probe Response, SN=132, FN=0, Flags=....R...C,
zte_d3:28… Espressi_a6… 802.11   119 Probe Response, SN=132, FN=0, Flags=....R...C,
```

Figure 7.6: Screenshot from Wireshark with the fuzzer responding to the probe request from the tested ESP32. The screenshot also shows that the used wireless interface driver automatically retries the transmission when acknowledgement was now received. This is indicated by the R in the listing.

7.3.3 Disassociation and Deauthentication

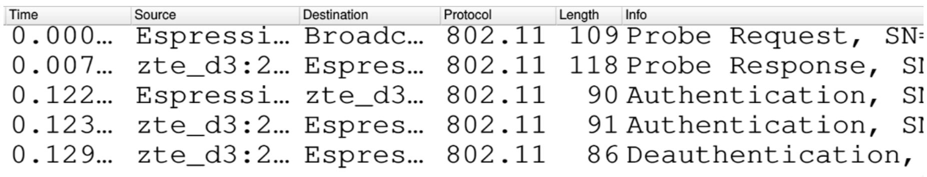
The functional testing of the disassociation and deauthentication frame fuzzing was similar to the beacon frame tests. The main tool for verification of the expected results was Wireshark.

7.3.4 Authentication

To test the responding to the authentication frames, authentication frame from the tested device must be sent first. This is done when the device is trying to connect. Most of the devices send probe request frames before the authentication. Presumably, it confirms the existence of the network and the correct destination MAC address of the authentication frame. Manually dissected messages from the authentication exchange can be seen in sFigure 7.7.

7.4 Fuzzing coverage

The fuzzing coverage is difficult to measure exactly. We could measure the ratio of the possible unique frames produced by the fuzzer to all unique frames. The design of fuzzed



```
Time      Source        Destination  Protocol  Length Info
0.000…    Espressi…     Broadc…      802.11    109    Probe Request, SN=
0.007…    zte_d3:2…     Espres…      802.11    118    Probe Response, SI
0.122…    Espressi…     zte_d3…      802.11     90    Authentication, SI
0.123…    zte_d3:2…     Espres…      802.11     91    Authentication, SI
0.129…    zte_d3:2…     Espres…      802.11     86    Deauthentication,
```

Figure 7.7: Screenshot from Wireshark showing the authentication frames sent from the SUT and the fuzzed response. They are preceded by the probe request – probe response exchange and followed by the deauthentication initiated by the fuzzer. The shown capture was dissected manually to not include retransmitted frames.

intentionally minimizes this ratio because of time limitations. This metric would also ignore the fact that the Wi-Fi devices can have different internal states during which they behave differently. A more telling metric could be the coverage of known vulnerabilities in Wi-Fi implementations discoverable by fuzzing. For this, a list of such vulnerabilities must be created. As a source, we used the list of vulnerabilities found by other fuzzers and evaluated the vulnerabilities found in the CVE list [1]. The list of vulnerabilities that could be found by fuzzing was created by filtering the database one by one for keywords: "Wi-Fi", "frame", "buffer overflow". The compiled table can be seen in the Appendix A.

The fuzzing coverage is still hard to validate against the selected vulnerabilities. The best way to validate our claims would be to run the fuzzer against the devices with the vulnerable software. Since we do not have access to those devices and the exact payloads are (with the exception of one) not publicly available, we can only make assumptions about them and evaluate whether the implemented fuzzer produces such frames.

7.4.1 Categorization of the found vulnerabilities

We divided the completed list into four categories: the reproducible by fuzzer, reproducible by fuzzer using random fuzzing, not reproducible by the new fuzzer, and the vulnerabilities we cannot reliably categorize without the details about exploit. The discoveries which are discoverable by fuzzing are not discussed in this section.

Vulnerabilities, which may be reproducible by the random fuzzing

This would be inefficient and would depend on the seed selected and the amount of produced frames with the random data. Those vulnerabilities would be better discoverable if all possible information elements were modelled in the fuzzer.

- CVE-2014-9902
- CVE-2017-6956
- CVE-2017-11121
- CVE-2019-12588

Unknown

The vulnerabilities mentioned could be found by fuzzing, but their categorization is impossible without further information about them.

- CVE-2011-0196 – no mention of the frame types or fields triggering the vulnerability
- CVE-2017-6957, CVE-2011-0172 – no mention of the exact vulnerable elements

Vulnerabilities not discoverable by the fuzzer

- CVE-2007-5651, CVE-2008-1144, CVE-2019-12587, CVE-2019-12586 – currently not fuzzing frames beyond the association frames
- CVE-2017-6957 – the fuzzer does not support reassociation frames fuzzing
- CVE-2017-11120 – would require implementation of action frame fuzzing and the RRM neighbour report fuzzer
- CVE-2019-1826 – would need to fuzz data type frames with QoS information

7.4.2 Results

From the categorized vulnerabilities, we can see, the fuzzer is able to fuzz most of the information elements fields. The cases not covered by the fuzzer are results of the fuzzer design and usage of model fuzzing. Better results would require modelling of every information element, type, and subtype of the Wi-Fi specification. This is somewhat mitigated by the usage of random fuzzing in specific instances. The most numerous reason for the fuzzer not being able to find the vulnerability comes from the focus on fuzzing the frames not requiring the knowledge of network password. The authentication through EAP is performed after the association and was not tested. In the future, fuzzing of all the vulnerabilities not discoverable currently should be added. The design of the fuzzer allows this extension without significant barriers in general design.

7.5 Device testing

The fuzzer was used to test devices. The test was focused on the ESP32 and ESP32-S2 chips but included mobile phone stations, a printer, or a smartwatch.

7.5.1 Experiment design

The setup used for the experiment is the same as the one described in Section 7.3. The exact configuration used is different for every test case. The configuration used for the ESP32 tests can be found in the repository [2] with the ESP32 program sources and the test runner.

The tests were trying to find anomalies in the SUT behaviour during and after the fuzzing. Such anomalies could be restarts, visible slowdowns, inability to use Wi-Fi or unexpected lack of send frames. During the whole duration of the test, the Wireshark was capturing frames on the channel used. The captured frames were monitored during the test. The captures were also later manually inspected for any anomalies.

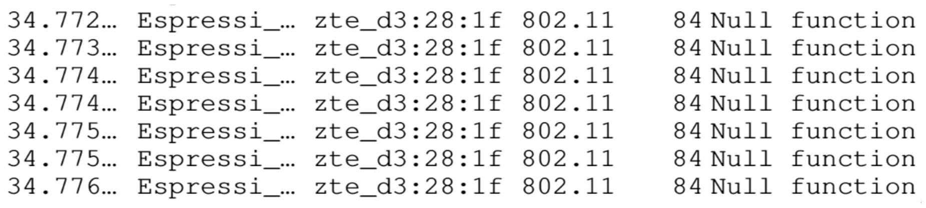
[2]Test runner with test cases and configurations – https://github.com/xvenge00/fuzz_runner

7.5.2 ESP32 and ESP32-S2

The main testing device was ESP32. The testing of the ESP32-S2 variant did not reveal any differences in their behaviour and was the same in every observable way. For this reason, they are described together. ESP32 is an MCU with integrated Wi-Fi and Bluetooth connectivity used in a wide array of devices. Testing an ESP32 chip as a station has an advantage compared to ordinary station devices. It can be automated, the scanning can be configured, and even the authentication can be automated. This was leveraged during the fuzzing to cover more possible errors.

Findings

The monitor did not find any critical failures or hangups. During the subsequent inspection of the communication by Wireshark [3], a large amount of traffic from the tested device was found. When the device is associated but does not have an IP address, the device spams null data frames, which can be seen in the screenshot in figure 7.8. This causes a large temperature increase in the device. The amount of sent frames is in hundreds. They are divided only by occasional DHPC Discover packets, which are sent in an effort of obtaining IP address.



```
34.772… Espressi_… zte_d3:28:1f 802.11    84 Null function
34.773… Espressi_… zte_d3:28:1f 802.11    84 Null function
34.774… Espressi_… zte_d3:28:1f 802.11    84 Null function
34.774… Espressi_… zte_d3:28:1f 802.11    84 Null function
34.775… Espressi_… zte_d3:28:1f 802.11    84 Null function
34.775… Espressi_… zte_d3:28:1f 802.11    84 Null function
34.776… Espressi_… zte_d3:28:1f 802.11    84 Null function
```

Figure 7.8: Screenshot from Wireshark showing the spam of null function frames.

The further inspection showed that the device created the event corresponding to the station connected state and then waited for the IP. This was in line with the test application, the example application provided by the ESP-IDF and any real application trying to communicate using the Wi-Fi stack. Every 10 seconds, the device sends probe request frames, to which the fuzzer replies with valid probe response frames. When the fuzzer does not send the response, the connection times out, the device deauthenticates by itself and, depending on the configuration, retries the association from the beginning. When the fuzzer responds to the probe requests, the device starts spamming null function frames (data frame without any data). The frames were a result of the device trying to tell the access point about its intention to go to a power-saving state. Similar behaviour can be seen in other station devices. On the ESP32, this could potentially mean reduced battery life for the device, as the chip started producing significant heat, which to the point of potential lighter burns (we do not have the exact temperature, as the ESP32 does not have an internal sensor).

7.5.3 Samsung Galaxy S9

The testing of the Samsung Galaxy S9 revealed a weakness in the design of the fuzzer. It is a smartphone running Android operating system version 10. The design of the fuzzer was based on the assumption that the MAC address can be tied to the device. Even though

[3]Wireshark – https://www.wireshark.org/

the MAC address can be changed, we assumed it would not happen during the connection/scanning.

What happened:

1. The fuzzer was started with the target device set to the current address of the phone. The address was collected using the Wireshark. The first three bytes indicated the vendor – Samsung.

2. The phone Wi-Fi menu was opened, which started active scanning indicated by probe requests captured by Wireshark and detected by the fuzzer.

3. The fuzzer started responding to the requests with valid probe responses.

4. The phone detected the frames and showed the SSID `"FUZZING"` from the responses.

5. We select the SSID to try to connect to the fuzzer.

6. The phone sends the disassociation frame to the previously connected access point.

7. The MAC address of the scanning phone no longer shows up in the network capture.

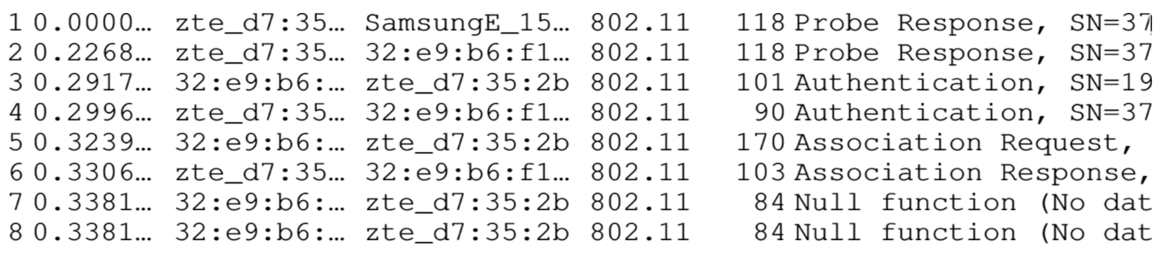
8. The device with a previously unseen MAC address (private address, not belonging to the Samsung vendor) sends a probe request frame containing the SSID `"FUZZING"`.

9. When the fuzzer does not respond to the device, the association fails, and the phone reconnects to the previous access point.

10. When the fuzzer responds with the same valid responses (same SSID and capabilities as previously send responses), the new device tries to authenticate and associate itself with the fuzzer.

11. The phone then considers itself connected but tries to get an IP address.

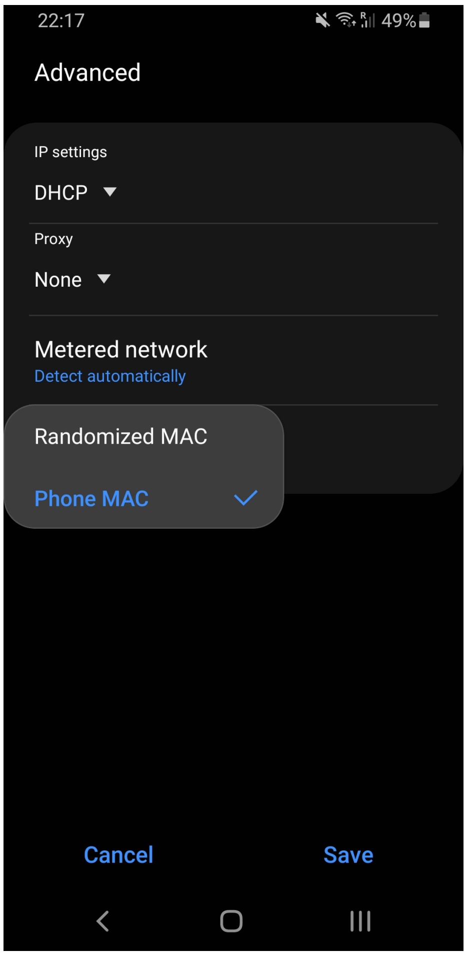
12. This is tried indefinitely until it does so or the user disconnects manually.



```
1 0.0000…  zte_d7:35…  SamsungE_15…  802.11   118 Probe Response, SN=37
2 0.2268…  zte_d7:35…  32:e9:b6:f1…  802.11   118 Probe Response, SN=37
3 0.2917…  32:e9:b6:…  zte_d7:35:2b  802.11   101 Authentication, SN=19
4 0.2996…  zte_d7:35…  32:e9:b6:f1…  802.11    90 Authentication, SN=37
5 0.3239…  32:e9:b6:…  zte_d7:35:2b  802.11   170 Association Request,
6 0.3306…  zte_d7:35…  32:e9:b6:f1…  802.11   103 Association Response,
7 0.3381…  32:e9:b6:…  zte_d7:35:2b  802.11    84 Null function (No dat
8 0.3381…  32:e9:b6:…  zte_d7:35:2b  802.11    84 Null function (No dat
```

Figure 7.9: A screenshot from Wireshark showing the randomization of the MAC addresses before authentication.

After a short investigation of the cause, the cause was found in the Android OS documentation – *MAC Randomization* [31]. The feature was introduced in Android 9 as an option in the developer menu and as a feature always present in Android 10. The MAC address randomization is used for increased privacy of the user. The randomized address is harder to track. The randomization can be turned off in the menu, but only after a successful connection and IP assignment.


22:17 49%

Advanced

IP settings
DHCP

Proxy
None

Metered network
Detect automatically

Randomized MAC
Phone MAC ✓

Figure 7.10: A screenshot from the Samsung Galaxy S9 running Android 10 with MAC randomization and open menu of the connected network, which allow the switching off the feature.

The feature cannot be reliably added to the existing fuzzer, and for this reason, we cannot try automatic fuzzing beyond the class 1 frames (device does not have to be authenticated or associated). Even if the randomization was not present, automatic testing would require a more complicated setup or a user constantly trying to connect. The manual fuzzing would be too labour intensive. Only the class 1 frames were fuzzed. The results did not reveal any flaws in the system, and the device seemed to function without any problems.

The SSIDs from the beacon frames did show up in the network list menu. The resend count had to be increased to 20 in order to ensure reliable scanning. The disassociation and deauthentication frames had to be tested with manual intervention. The device did dissociate and deauthenticate every time, even though the length of the message was wrong. The authentication fuzzing had no effect because the device was dropping the authentication frames when it was not trying to authenticate.

7.5.4 HP Deskjet 3545

We tried the fuzzer on a Wi-Fi capable HP Deskjet 3545 printer. The Wi-Fi acts as an access point and a station device at the same time. It produces a large number of beacon frames. To test the station portion of the Wi-Fi stack, the scanning had to be periodically turned on manually. This turned on the active scanning. The printer also reacted to the beacon frames, and some SSID did show up in the menu.

The process was labour intensive and very inefficient. The scanning ran for about 15 seconds, after which the found networks are displayed. For this reason, only a small portion of the fuzzed frames was tried. The testing did not show any malfunctions or abnormal behaviour. The printer did react to the deauthentication and disassociation frames, even with invalid parameters.

7.5.5 Samsung Galaxy Watch Active2

The Samsung Galaxy Watch Active2 is a smartwatch with Tizen operating system and Wi-Fi station capabilities. The device was tested only by fuzzing the class 1 frames, which include beacon, probe response, deauthentication and disassociation frames. The SSIDs from beacon and probe response frames did show up in the network menu, but the watch continued to work without noticeable side effects.

The fuzzing of deauthentication frames did show some interesting behaviour. The watch did disconnect from the network when the fuzzer sent deauthentication frames with the source address of the associated access point, as expected. However, after the test was rerun, the watch could not connect to the access point. The watch removed the SSID from its database, and even after providing the correct password, the watch was unable to connect. This was tried repeatedly to confirm the results. The inability to connect was probably not caused by the fuzzed fields themselves.

The issue is resolved by itself when the watch is left alone and reconnects to the access point. The problem only shows when the user tries to connect manually and sets the password in the menu. At that time, the fuzzing is already turned off and should not affect the device. The watch cannot connect, but after a couple of minutes, it reconnects by itself. The result is DoS, but the same effect has sending valid deauthentication frames. This is an attribute of the Wi-Fi protocol itself.

7.6 Summary

The testing confirmed the fullfilment of the functional requirements set in the design stage. Compared to the commercial solutions, the fuzzer has some deficiencies. The evaluation of the fuzzing space revealed some room for improvement especially in the fuzzing of the EAP authentication. Even though the deficiencies exist, the fuzzer vould be able to expose majority of vulnerabilities from the compiled list. Testing of devices using the fuzzer did not reveal any vulnerabilities.

Chapter 8

Conclusion

The thesis was studying Wi-Fi protocol and its implementations in various devices. The examined devices included smartphone, smartwatch, printer and most importantly, the Wi-Fi stack of the ESP32 and ESP32-2S chips used in a wide array of devices worldwide. The knowledge gained when observing the Wi-Fi devices was used to implement testing tools. The testing used fuzzing, which is generally a technique of sending semi-random data to the system under test. Different fuzzing techniques were examined, with their respective advantages and weaknesses. Based on them, the most appropriate was selected for the design of the final fuzzer.

The goal of the thesis was to implement an open-source fuzzing tool capable of testing Wi-Fi protocol for different devices, which was successfully done. Its purpose is to detect implementation errors that could be exploited for malicious attacks. The capability to detect the errors was based on the list of the known vulnerabilities found by the already existing fuzzers or by other means. The resulting tool is one of only a few publicly available. The other available Wi-Fi fuzzing tools are impractical to extend or unmaintained and with little room to progress. The design of the fuzzer provides easy to extend architecture and set-up/tear-down function, which no other fuzzer provides, allow for automatic testing of more than basic frames. The testing revealed potential problems with fuzzing modern devices, which use MAC address randomization. We could not overcome those, and they would require an isolated testing environment with some modifications to the tool.

Another goal of the thesis was to test the implemented tool on the ESP32 and ESP32-S2 Wi-Fi stack. This was automated using the specialized testing applications for the chip, a special monitoring probe connected to the serial output of the chip and the runner script. The results did not reveal any anomalies in its behaviour during the testing. The manual examination of the captured data showed the same.

In the future, the tool could be developed further and provide more efficient fuzzing for the various information elements in the frames. Testing of other chips used for IoT devices could also be easily implemented thanks to the design of the decoupled remote probes. For now, the tool provides good coverage of the core features of the Wi-Fi standard using the model fuzzing and some coverage of the more obscure features using the random fuzzing.

Bibliography

[1] *About CVE* [obline]. 2021. Revised 2021-3-29 [cit. 2021-5-10]. Available at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1157.

[2] Aleph One. Smashing the Stack for Fun and Profit. *Phrack* [online]. November 1996, vol. 7, no. 49, [cit. 2021-05-07]. Available at: http://www.phrack.com/issues.html?issue=49&id=14.

[3] Anubhav, A. *Huawei router exploit involved in Satori and Brickerbot given away for free on Christmas by Blackhat Santa* [online]. December 2017 [cit. 2021-05-07]. Available at: https://blog.newskysecurity.com/huawei-router-exploit-involved-in-satori-and-brickerbot-given-away-for-free-on-christmas-by-ac52fe5e4516.

[4] *ASSET (Automated Systems SEcuriTy) Research Group: News* [online]. April 2021. News message. Available at: https://asset-group.github.io/.

[5] Beer, I. *An iOS zero-click radio proximity exploit odyssey* [online]. Google, Dec 2020 [cit. 2021-05-10]. Available at: https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html.

[6] Butti, L. *Wi-Fi Advanced Fuzzing*. 2007. Black Hat Europe. Available at: https://www.blackhat.com/presentations/bh-europe-07/Butti/Presentation/bh-eu-07-Butti.pdf.

[7] Butti, L. *Wifuzzit* [online]. 2007 [cit. 2021-05-10]. GitHub repository. Available at: https://github.com/0xd012/wifuzzit.

[8] Butti, L., Tinnes, J. and Veysset, F. *Wi-Fi Implementation Bugs: an Era of New Vulnerabilities*. 2007 [cit. 2021-05-10]. Hack.lu. Available at: https://www.cr0.org/paper/hacklu2007-final.pdf.

[9] Cache, J. and Maynor, D. *Device Drivers:Don't build a house on a shaky foundation*. 2006 [cit. 2021-05-10]. Black Hat USA. Available at: https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Cache.pdf.

[10] Clements, A. A., Almakhdhub, N. S., Saab, K. S., Srivastava, P., Koo, J. et al. Protecting Bare-Metal Embedded Systems with Privilege Overlays. In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, p. 289–303. DOI: 10.1109/SP.2017.37. Available at: https://doi.org/10.1109/SP.2017.37.

[11] *CVE-2015-1157*. [Available from MITRE, CVE-ID CVE-2015-1157.]. 2015 [cit. 2021-05-07]. Available at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1157.

[12] Dowd, M., McDonald, J. and Schuh, J. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006. ISBN 0321444426.

[13] Espressif. *ESP32* [online]. [cit. 2021-05-10]. Available at: https://www.espressif.com/en/products/socs/esp32.

[14] Espressif. *Security Advisory concerning fault injection and eFuse protections (CVE-2019-17391)* [online]. Shanghai, China: [b.n.], November 2019 [cit. 2021-05-07]. Available at: https://www.espressif.com/en/news/Security_Advisory_Concerning_Fault_Injection_and_eFuse_Protections.

[15] Espressif. *ESP32 Series: Datasheet* [online]. 3.6th ed. 2021, 2021-03-19 [cit. 2021-05-07]. Available at: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[16] Espressif. *ESP32-C3 Family: Datasheet* [online]. 0.8th ed. 2021, 2021-04-23 [cit. 2021-05-07]. Available at: https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf.

[17] Garbelini, M. E., Wang, C. and Chattopadhyay, S. Greyhound: Directed Greybox Wi-Fi Fuzzing. *IEEE Transactions on Dependable and Secure Computing (TDSC)* [online]. 2020. Available at: https://asset-group.github.io/papers/Greyhound.pdf.

[18] Garbelini, M. E. *ESP32/ESP8266 EAP client crash (CVE-2019-12586): Crashing ESP devices connected to enterprise networks* [online]. 2017 [cit. 2021-05-10]. Available at: https://matheus-garbelini.github.io/home/post/esp32-esp8266-eap-crash/.

[19] Garbelini, M. E. *ESP8266 Beacon Frame Crash (CVE-2019-12588): Easily crashing ESP8266 Wi-Fi devices* [online]. 2017 [cit. 2021-05-10]. Available at: https://matheus-garbelini.github.io/home/post/esp8266-beacon-frame-crash/.

[20] Garbelini, M. E. *Zero PMK Installation (CVE-2019-12587): Hijacking ESP32/ESP8266 clients connected to enterprise networks* [online]. 2017 [cit. 2021-05-10]. Available at: https://matheus-garbelini.github.io/home/post/zero-pmk-installation/.

[21] Gast, M. S. *802.11 Wireless Networks: The Definitive Guide*. 2nd ed. O'Reilly Media, Inc., April 2005. ISBN 0-596-10052-3.

[22] Kerr, K. *From Algorithms to Coroutines in C++* [online]. October 2017 [cit. 2021-05-05]. Available at: https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/october/c-from-algorithms-to-coroutines-in-c.

[23] Limited Results. *Fatal Fury on ESP32: Time to Release Hardware Exploits*. December 2019. Black Hat Europe. Available at: https://www.blackhat.com/eu-19/briefings/schedule/#fatal-fury-on-esp-time-to-release-hardware-exploits-17336.

[24] Maynor, D. *Beginner's Guide to Wireless Auditing* [online]. September 2006 [cit. 2021-05-10]. Available at: https://community.broadcom.com/symantecenterprise/

communities/community-home/librarydocuments/viewdocument?DocumentKey=ec8602a4-a4ec-4890-8771-9f24cd0bbb4b&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments.

[25] Miller, B. P., Fredriksen, L. and So, B. An empirical study of the reliability of UNIX utilities. *Communications of the ACM.* ACM New York, NY, USA. 1990, vol. 33, no. 12. DOI: 10.1145/96267.96279.

[26] Moore, H. D. November 2006. Available at: https://kernelfun.blogspot.com/.

[27] Mordor Intelligence. *IoT Chip Market - Growth, Trends, COVID-19 Impact, and Forecasts (2021 - 2026)* [online]. January 2021 [cit. 2021-05-07]. Available at: https://www.researchandmarkets.com/reports/4622612/iot-chip-market-growth-trends-covid-19.

[28] Neystadt, J. *Automated Penetration Testing with White-Box Fuzzing* [online]. February 2008 [cit. 2021-05-10]. Available at: https://docs.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10).

[29] PaX Team. *Aslr* [online]. March 2003 [cit. 2021-05-09]. Available at: https://pax.grsecurity.net/docs/aslr.txt.

[30] Pleiter, B. *Fuzzing Wi-Fi in IoT devices.* 2020. Bachelor's Thesis. Radboud University.

[31] *Privacy: MAC Randomization* [online]. April 2021 [cit. 2021-05-07]. Available at: https://source.android.com/devices/tech/connect/wifi-mac-randomization.

[32] Takanen, A., DeMott, J., Miller, C. and Kettunen, A. *Fuzzing for software security testing and quality assurance.* 2nd ed. Norwood, MA United States: Artech House, 2018. ISBN 978-1-60807-850-9.

[33] Tinnes, J. and Butti, L. *Madwifi remote kernel exploit* [online]. September 2010 [cit. 2021-05-10]. Available at: https://www.exploit-db.com/exploits/16835.

[34] Vanhoef, M. *Proof of concepts of attacks against wi-fi implementations.* [online]. 2017 [cit. 2021-05-10]. GitHub repository. Available at: https://github.com/vanhoefm/blackhat17-pocs.

[35] Vanhoef, M. *WiFuzz: Detecting and Exploiting Logical Flaws in the Wi-Fi Cryptographic Handshake.* 2017. Black Hat USA. Available at: https://www.blackhat.com/us-17/briefings/schedule/#wifuzz-detecting-and-exploiting-logical-flaws-in-the-wi-fi-cryptographic-handshake-6827.

[36] Wi-Fi Alliance. *Certification* [online]. [cit. 2021-05-05]. Available at: https://www.wi-fi.org/certification.

[37] Wi-Fi Alliance. Who We Are. *History* [online]. [cit. 2021-05-05]. Available at: https://www.wi-fi.org/who-we-are/history.

[38] Wi-Fi Alliance. *Look for the Logo* [online]. [cit. 2021-05-05]. Available at: https://www.wi-fi.org/look-for-the-logo.

[39] Wright, J. and Cache, J. *Hacking Exposed Wireless: Wireless Security Secrets & Solutions*. 3rd ed. McGraw-Hill Education Group, 2015. ISBN 0071827633.

[40] Zhodiac. HP-UX (PA-RISC 1.1) Overflows. *Phrack* [online]. December 2001, vol. 11, no. 58, [cit. 2021-05-07]. Available at: http://phrack.org/issues/58/11.html.

[41] Zhou, W., Guan, L., Liu, P. and Zhang, Y. Good Motive but Bad Design: Why ARM MPU Has Become an Outcast in Embedded Systems. *CoRR*. 2019, abs/1908.03638. Available at: http://arxiv.org/abs/1908.03638.

Appendix A

Test vulnerabilites

Res.	Vuln. ID	Type	Description
OK	CVE-2006-6059	STA	Buffer overflow in MA521nd5.SYS driver 5.148.724.2003 for NetGear MA521 via beacon or probe responses with a long supported rates information element
OK	CVE-2006-6125	STA	Heap-based buffer overflow in the wireless driver for NetGear WG311v1 via management frame with a long SSID
OK	CVE-2006-6332	STA	Stack-based buffer overflow in MadWifi via long information element – This bug was proven to be remotely exploitable.
OK	CVE-2007-0933	STA	Buffer overflow in the wireless driver D-Link DWL-G650+ via beacon frame with a long TIM Information Element
OK	CVE-2007-5474	AP	Atheros Vendor Specific Information Element Overflow
OK	CVE-2007-5475	AP	Marvell Driver Multiple Information Element Overflows
NO	CVE-2007-5651	AP	Extensible Authentication Protocol Vulnerability
NO	CVE-2008-1144	AP	Marvell driver EAPoL-Key length overflow
OK	CVE-2008-1197	AP	Marvell driver Null SSID association request vulnerability
OK	CVE-2008-4441	AP	Marvell driver vulnerability by malformed association request containing the WEP flag
RND	CVE-2017-6956	STA	Buffer overflow in Broadcom Wi-Fi when handling an 802.11r (FT) authentication response, leading to remote code execution via a long R0KH-ID field in a Fast BSS Transition Information Element (FT-IE)
OK	CVE-2009-0052	AP	Atheros driver truncated reserved management frame vulnerability

Table A.1: The first part of the vulnerability list used for the fuzz coverage evaluation. Their source is documentation of *wifuzzit* [7], which found them, the bachelor thesis by Bart Pleiter [30] and the vulnerability database [1].

Res.	Vuln. ID	Type	Description
-	CVE-2011-0196	AP	AirPort in Mac OS X denial of service via Wi-Fi frames.
-	CVE-2011-0172	AP	AirPort in Mac OS X denial of service via Wi-Fi frames (different vulnerability than CVE-2011-0162).
OK	CVE-2014-9901	STA	The Qualcomm Wi-Fi driver in Android device makes incorrect snprintf calls.
RND	CVE-2014-9902	STA	Buffer overflow in Qualcomm Wi-Fi driver in Android via a crafted Information Element (IE) in an 802.11 management frame
OK	-	STA	Nintendo DSi XL crash by ERP information tag in probe response frames
NO	CVE-2019-1826	AP	A vulnerability in the quality of service (QoS) feature of Cisco Access Points due to improper input validation on QoS fields.
NO	CVE-2017-6957	STA	Buffer overflow in the Broadcom Wi-Fi chips via a specialy crafted reassociation response frame with a Cisco IE (156)
NO	CVE-2017-11120	STA	Buffer overflow in Broadcom Wi-Fi chips via a malformed RRM neighbor report frame
RND	CVE-2017-11121	STA	Broadcom Wi-Fi chips, properly crafted malicious Fast Transition frames can potentially trigger internal buffer overflows.

Table A.2: The second part of the vulnerability list used for the fuzz coverage evaluation. Their source is documentation of *wifuzzit* [7], which found them, the bachelor thesis by Bart Pleiter [30] and the vulnerability database [1].

Appendix B

Contents of the included storage media

Directory	Description
bin/	Directory with the compiled fuzzer
doc/	Directory with the text documentation
fuzzer/	Directory with the fuzzer source files
tests/	Directory with test runner, with the ESP32 test program sources and configurations

Table B.1: The directory structure of the included storage media.