POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Thesis

Identifying Software and Protocol Vulnerabilities in WPA2 Implementations through Fuzzing

Supervisors
Prof. Antonio Lioy
Dr. Jan Tobias Müehlberg
Dr. Mathy Vanhoef

Candidate
Graziano Marallo

Academic Year 2018-2019

Dedicated to my parents

Summary

Nowadays many activities of our daily lives are essentially based on the Internet. Information and services are available at every moment and they are just a click away. Wireless connections, in fact, have made these kinds of activities faster and easier. Nevertheless, security remains a problem to be addressed. If it is compromised, you can face severe consequences. When connecting to a protected Wi-Fi network a handshake is executed that provides both mutual authentication and session key negotiation. A recent discovery proves that this handshake is vulnerable to key reinstallation attacks. In response, vendors patched their implementations to prevent key reinstallations (KRACKs). However, these patches are non-trivial, and hard to get correct. Therefore it is essential that someone audits these patches to assure that key reinstallation attacks are indeed prevented.

More precisely, the state machine behind the handshake can be fairly complex. On top of that, some implementations contain extra code to deal with Access Points that do not properly follow the 802.11 standard. This further complicates an implementation of the handshake. All combined, this makes it difficult to reason about the correctness of a patch. This means some patches may be flawed in practice.

There are several possible techniques that can be used to accomplish this kind of analysis such as: formal verification, fuzzing, code audits, etc. Among all of these, software fuzzing is, de facto, one of the most popular vulnerability discovery solutions and for this reason has been selected. Feasibility and methodology on how to fuzz an open-source implementation with a goal to detect potential flaws will be discussed and presented. The goal of this thesis is then to define whether it is possible to detect software bugs in a protocol implementation, including protocol-level vulnerabilities like KRACK, using a systematic approach and in an automated way.

Acknowledgements

I would like to express my deep gratitude to Professor Antonio Lioy and Dr. Jan Tobias Müehlberg, my research supervisors, for their patient guidance, enthusiastic encouragement and useful critiques of this research work. I would also like to thank Dr. Mathy Vanhoef, my mentor, for his advice and assistance.

I must express my very profound gratitude to my girlfriend Francesca and to my best friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

A special thanks go to my parents and my whole family for their love, support and constant encouragement throughout my life and unviversity career.

I would also dedicate this achievement to my mother, who taught me that no matter how much life is beating you down, there is always a reason to get up and fight back.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays many activities in our daily lives are essentially based on the internet. Information and services can be accessed with one click, making life easier. Various forms of communication, online shopping, financial services, entertainment and administration are condensed in one place. Along with the bright side of the internet, it hosts a dark side as well. The risks of using it are real and boundless. Whenever personal information, messages, banking information, picture or videos are sent via the internet, one can expect those to be secure and private. For the most part, internet is indeed secure and private, however there are still a number of severe security risks.

Many methods are used to protect against these dangers to the multiple layers of the internet, to try and ensure a certain degree of security. At network layer, for example TCP/IP protocol, can be protected using cryptographic methods and security protocols. Secure Sockets Layer (SSL) and Transport Layer Security (TLS) are two clear example of this kind of security protocol. These two cryptographic protocols are designed to provide communications security over a computer network. The TLS protocol finds its major use in providing privacy and data integrity between different computer applications. For instance, several versions of the protocol are used in applications such as web browsing, email, messaging and Voice over IP (VoIP) [1].

In the last few years, we have witnessed a wide spread of Wi-Fi networks all over the world. Of course, there are security protocols specifically designed to protect this kind of communication. These protocols are WPA[1] and WPA2, and more recently WPA3. On a physical layer, in order to establish a connection, an Access Point (AP) is necessary. This device allows other Wi-Fi devices to connect to a wired network.

Usually, an access point is connected to a router via wired network, but can be also an integral component of the router itself. Since wireless networks do not require physical access to network equipment, unlike wired networks, it became easier for unauthorized users to passively monitor a network and capture all transmitted packets. To avoid this kind of situation, link-layer encryption can be used

[1]Wi-Fi Protected Access

and provide a layer of security for wireless networks also. After the initial usage of WEP (Wirless Equivalent Privacy) proved to be flawed, WPA and IEEE 802.11i amendments to the wireless LAN standard introduced a much improved mechanism for securing wireless networks. Exploiting encryption mechanisms based on strong cryptographic algorithms, for example AES(CCMP), ensured a greater efficiency in protection against unauthorised access. The software implementation that is in charge of taking care of all these issues is the *wpa_supplicant*. As a matter of fact, it implements WPA key negotiation with a WPA Authenticator and EAP authentication with an Authentication Server. It is designed to be a daemon program that runs in background and act as back-end component able to control the wireless connection. Of course, the supplicant can do its work whenever the network interface is available, meaning that the physical device must be present and enabled. The wpa_supplicant has a primary role in the wireless connection making it a crucial component when considering security[2].

A lot of effort has been made in order to provide and ensure security when communicating on the internet, both wired and wireless. Nevertheless, in the last few years, several major attacks have been made against the internet, exploiting flaws and vulnerabilities. One of the most serious is the "Heartbleed" bug, a vulnerability found in the popular OpenSSL cryptographic software library. This flaw allows the attacker to steal the information protected by SSL/TLS encryption used to secure the internet. Basically, anyone on the internet can read the memory of the system protected by the vulnerable versions of the OpenSSL software. This compromises the secret key used to identify the service providers and to encrypt the traffic, names and passwords. Thereby, the attacker is able to eavesdrop on communications and steal data impersonating both the user and the provider itself [3]. Another example is "CloudBleed", a security bug affecting Cloudflare's reverse proxies. This vulnerability caused edge servers to run past the end of buffer (i.e. buffer overrun) and return a portion of memory that was intended to store private information such as HTTP cookies, authentication tokens, etc. [4]. These are just two examples of the many software bugs that can affect the internet.

Even internet protocols can suffer from vulnerabilities and a clear example is the WPA2. Recent discoveries have proved that the WPA2 protocol and its implementation are not as secure as initially thought. The "(K)ey (R)einstallation (A)tta(CK)" abuses features of the protocol to reinstall an already in-use key, resetting the nonce and the replay counter associated to this key. The attack has been proved to be devastating with severe consequences on a large number of Linux and Android devices using the wpa_supplicant. In response to this discovery, vendors patched their implementations to prevent key reinstallation. Clearly, these patches are not simple at all, neither are they all-encompassing, meaning that flaws can still be around. In this work we propose a security analysis of one of the patches released by Intel.

Main research questions for this work are the following:

- If and how it could be possible to find software bugs, in a protocol implementation, using a systematic approach and in automated way?
- Is it possible to find protocol-level vulnerabilities, like KRACK, exploiting fuzzer analysis?
- Is it possible to devise a generalised methodology to apply automated vulnerability detection to cryptographic protocol implementations?

An established approach to automate the search for vulnerabilities is software fuzzing. Fuzzing, for instance, is an automated software testing technique that allows the tester to provide either invalid, unexpected or random data inputs to a target program. In doing so, the program is monitored for exceptions such as crashes, failing built-in code assertions, and potential memory leaks. The way the fuzzer performs its analysis differs based on what kind of strategies it uses [5]. In this work we focus on evaluating how American Fuzzy Lop (AFL) can be applied to protocol implementation with respect to the above research questions.

We have proved that is possible to devise a methodology to allow software fuzzing with the intent of detecting software vulnerabilities and protocol-level vulnerabilities in an open-source implementation of the 4-Way Handshake. By turning unit tests into test harnesses for a fuzzer it is possible to analyse, and eventually identify, potential flaws in this kind of implementation.

The thesis is structured as follows: a background overview about well-known vulnerabilities, WPA2, 4-Way Handshake and KRACK will be given. Subsequently a background overview about discovery techniques will be presented, stressing the concept of fuzzing and discussing AFL in detail. Eventually the attention will be focused on the methodology needed to prepare Inet Wirless Deamon, the successor of wpa_supplicant, to be fuzzed in order to discover software vulnerabilities. In the end, a potential approach to fuzzing protocol-level vulnerabilities will be given along with the implementation needed to do so.

Chapter 2

Background on WPA2 and its vulnerabilities

2.1 Vulnerability

The term **vulnerability** is rather common in the computer security field and over the years this term has become more and more relevant and crucial. A vulnerability is generally associated with a violation of a security policy and may be caused due to weak security rules or just a problem within the developed software itself. Theoretically speaking, all computer systems present some kind of vulnerabilities, whether they are devastating or not depends on how they can be exploited to damage the system. In the past years we have seen many attempts to clearly define what a vulnerability means. One of the most valuable is MITRE's one, a US federally funded research and development group, that has its own focus on analysing and solving critical security issues. They state that:

"A universal vulnerability is a state in a computing system (or set of systems) which either: allows an attacker to execute commands as another user allows an attacker to access data that is contrary to the specified access restrictions for that data allows an attacker to pose as another entity allows an attacker to conduct a denial of service MITRE believes that when an attack is made possible by a weak or inappropriate security policy, this is better described as exposure." [6]

An exposure can be seen as a precise state in a computing system which is not meant to be considered a universal vulnerability. In fact, general exposure is either:

- a possibility for an attacker to conduct information-gathering activities aimed to break the system;
- a possibility for an attacker to hide certain kinds of activities;
- when the inclusion of a set of capabilities behave as designed but can be easily compromised and caused to break down;
- a primary entry point, through which an attacker may attempt to gain access to the system or data, which is meant to be a problem according to some reasonable security policies;

- when trying to gain unauthorised access to a targeted system, an attacker usually performs a routine scan of it, collecting any exposed data, and then takes advantage of it exploiting security policy weaknesses and/or vulnerabilities.

For this reason, vulnerabilities and exposures are therefore both crucial points to carefully check and ensure while securing a system against unauthorised access or potential breaks [6]. Based on this definition it is clear enough how important it is to be able to detect these kinds of flaws and devise a plan, or a methodology, to somehow stem them in the most appropriate way.

2.2 Low-level Software Security

Once it has been defined what a vulnerability is, it is legitimate to give a brief overview of the well-known C language vulnerabilities and how they can be eliminated or at least contained. When dealing with software development many assumptions are made by the programmer, both implicitly and explicitly. Those assumptions, like concurrency, address encoding in pointers, order of heap allocation, size of a memory buffer, etc. and may lead to incorrect execution when the environment changes, even slightly. All of the cited assumptions are strictly related to low-level software security, and each one of them may cause the software to be unguarded towards attacks. On the other side, attackers make assumptions as well, and low-level software attacks rely on a number of specific properties about the hardware and software architecture of their target. Of course, also defence mechanisms are based on assumptions, which includes the capabilities of the attacker himself, the likelihood of the several types of attacks, the property of the software being defended and the execution environment. In order to have a full picture of the topic a brief overview and analysis of four low-level softwares will be given. These, for major class of attacks, are: stack-based buffer overflows, heap-based buffer overflows, return-to-libc attacks and data-only attacks.

Stack-based Buffer Overflows

Whenever a function is invoked at a particular call site, running until completion without incurring any exception, it is expected that that function will return to the instruction immediately following the caller. But, as said previously, this is an assumption that may be wrong in the presence of software bugs. For example, if the invoked function contains a buffer, and writes into that buffer are not guarded in a proper way, then the return address on the stack may be altered. This means it will be overwritten and corrupted. In such a case, if an attacker controls the data manipulated by the function, they will be able to trigger that corruption and change the return address to an arbitrary value. If so, on function returns, the attacker can direct execution to a malicious code and gain full control of over the entire behaviour of the software from that moment on [7]. Figure [2.1][1] shows an example of that.

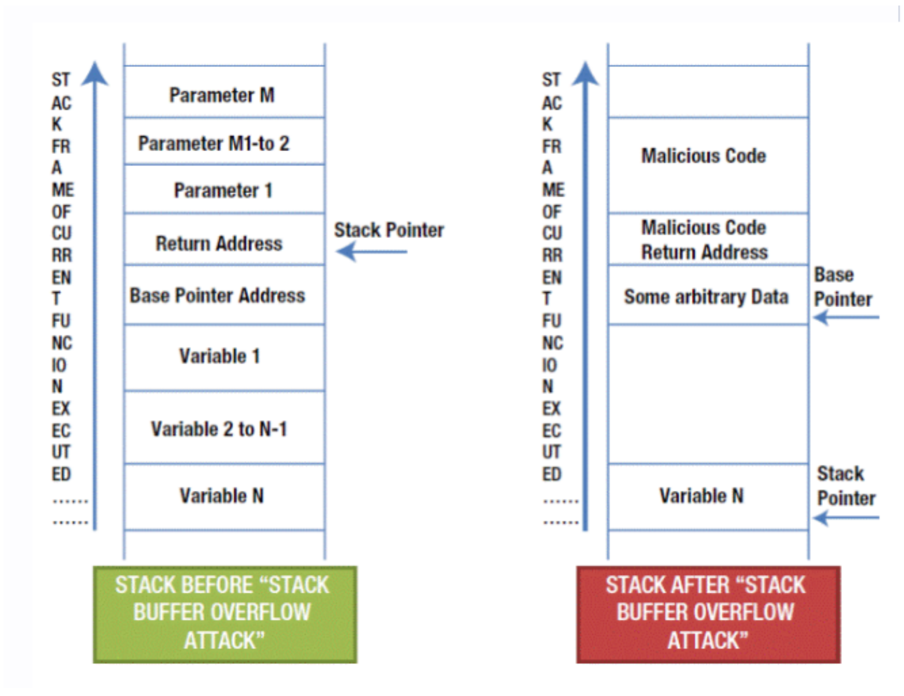

STACK FRAME OF CURRENT FUNCTION EXECUTED ......
Parameter M
Parameter M1-to 2
Parameter 1
Return Address
Base Pointer Address
Variable 1
Variable 2 to N-1
Variable N
Stack Pointer
STACK BEFORE "STACK BUFFER OVERFLOW ATTACK"
STACK FRAME OF CURRENT FUNCTION EXECUTED ......
Malicious Code
Malicious Code Return Address
Some arbitrary Data
Base Pointer
Variable N
Stack Pointer
STACK AFTER "STACK BUFFER OVERFLOW ATTACK"

Figure 2.1. Buffer overflow example

Heap-based Buffer Overflows

In the C language it is rather common that data buffers and pointers into data structure are combined. The programmer may not be aware of the fact that the data values can somehow affect the pointer values themselves. As said before, assumptions have to be taken into account and in the case of software bugs disastrous situations can arise. Specifically, the pointers may be corrupted as result of an overflow of the data buffer. This situation can verify that either the data structures

[1]Source: https://ebrary.net/26682/computer_science/applications

reside onto the stack or in the heap memory. For example, copying data into a buffer using the *strcpy* and *strcmp* library functions can lead to such a situation. The mechanism to carry out this kind of attack is very similar to the previous one, except for the fact that it is exploiting heap instead of stack. The consequences of such attack can result in the attacker gaining full control over the target which makes him free to completely change the behaviour of the software [7].

Return-to-libc

Differently from the attacks previously presented, that are vulnerable to *direct code injection*, there some countermeasures and situations in which the attacker needs to find a different strategy to exploit software flaws. In particular, one different approach to craft an attack may be executing a machine code that is already present on the target software in manner that is unexpected and not intended by the programmer. The attacker, as a matter of fact, could corrupt a function pointer to cause the execution of a library function that usually should be unreachable in the original source code and never executed. For this reason those kind of attacks are typically referred to as *return-to-libc*, as demonstrated by figure [2.1][2]. It is easy to see why those are preferable to others when the target software is based on architecture where input data is not executed directly as a machine code [7]. So, the *indirect code injection* is based on the execution of the target software's existing machine code in way that makes the attacker able to execute their own chosen input as a machine code instead, in order to gain control.

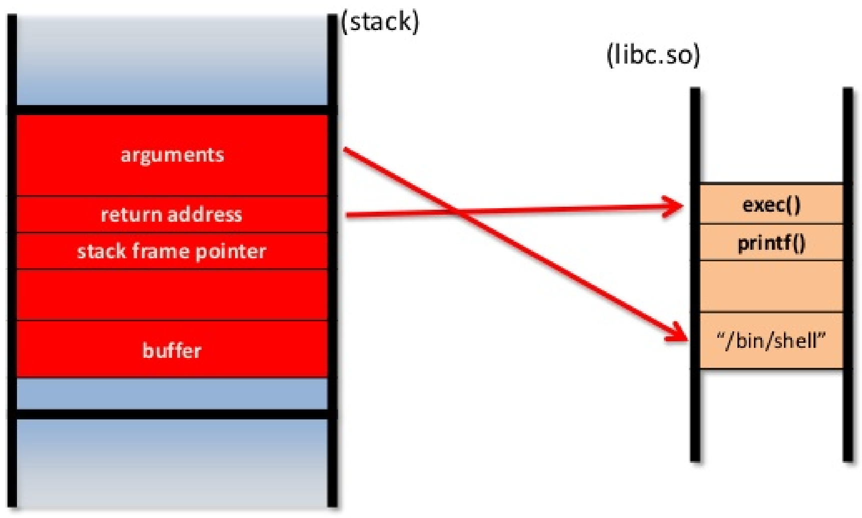


Figure 2.2. Return-to-libc example

[2]Source: https://www.slideshare.net/stogers/day-11introswvuls

Data-only Attacks

The last assumption that can be made while writing code is about the integrity of the data that is going to be manipulated. One example could be the value of a global variable, initialised at same point, and thought to be unchanged throughout the software's execution if it was never written by the software itself. Once again the assumption may not be correct in presence of a bug. This can obviously open the door to the injection of malicious codes aiming to alter the normal behaviour. An example could be usage of the *getenv* routine of the standard C library. This routine returns the exact string that was passed through the routine call *system* that determines what external command is launched. In such situations, the attacker is able to corrupt the table of the environment strings in order to launch an external command of their choice. By doing that it is clear how, for the attacker, is easy to make the software execute a malicious code [7].

All of those attacks that have been presented can be stemmed and blocked with specific techniques. These defences are stack canaries, non-executable data, control flow integrity and address-space layout randomization (ASLR). In this work we will not discuss them, but is still important to keep in mind that for the presented software bugs we do have countermeasures that somehow can guard and defend our software from being attacked and exploited by a malicious third party. Furthermore, apart from the well-know ones, there are other software bugs that have to be taken into account that have not been discovered, or at least analysed, yet.

2.3 4-Way Handshake Principles

Indeed, it might be possible that software vulnerabilities are also present in software used to deal with the functioning of an internet protocol. The protocol that is treated in this work is the WPA2 protocol. For this reason it is legitimate and necessary to have an overview of the protocol itself and the vulnerability that has been discovered as affecting it. Most of the Wi-Fi networks that are used today are protected and secured by the WPA2, also known as Wi-Fi Protected Access 2. Even the public hotspots that are wide spread around the world, are running under the WPA2 protocol despite the fact that authentication is not performed for obvious reasons but the encryption should be used instead. Such encrypted networks are based on the Wi-Fi handshake that is able to securely negotiate a fresh pairwise key and use this key to ensure that the normal traffic exchange across the network is encrypted. An essential point of the work is surely to understand where the 4-Way Handshake is placed and how it works during the establishment of a secure connection between a generic client and a generic server.

2.3.1 Technical and Historical Background

At its creation, the 802.11 original standard supported a basic security protocol called WEP (Wired Equivalent Privacy). However, unfortunately it was designed in a such way that many flaws were present and today it is considered completely broken[8][9][10]. For this reason the IEEE proposed two solutions: a long-term one and short-term one. The long-term solution is called (AES-)CCMP, that basically uses an AES in counter mode for confidentiality and CBC-MAC for authenticity. The problem at the time was that most of the vendors implemented the cryptographic primitives of WEP in hardware and, as a result, an incompatibility issue arose and older devices were unable to support CCMP. In order to solve this problem a short-term solution was designed and it was called (WPA-)TKIP protocol. In principle it is very similar to WEP but it is based on RC4 cipher meaning that the older devices can support TKIP using only a firmware update. Once the protocol 802.11i was standardized, the WPA2 certification program started, requiring that a device had supporting CCMP and was still able to achieve retro-compatibility with the TKIP. This led to an important misconception: even though WPA2 was claimed to be used, meaning (AES-)CCMP, the reality is that the WPA2 network might still be using TKIP instead.

2.3.2 Functioning

There are 5 different stages that regulate the functioning of the 4-Way Handshake protocol.

Stage 1: *Network Discovery* Networks can be discovered essentially in two ways: one way is by passively listening for a beacon or, alternatively, by actively sending a probe request. When the Access Point (AP) receives the latter kind of request it will reply to the sender with a probe response. In both cases, the response contains the name and the capabilities of the network and, along with those two, the RSNE[3] will be sent. This will contain several pieces of useful information such as the supported pairwise cipher suites of the network, the group cipher suite that will be used, and the security capabilities exposed by the AP. After the client has found a network to connect to, the handshake process can start and the connection can be established. During this phase the client will be known as the **supplicant**, while the access point will be called **authenticator**.

Stage 2: *Authentication and Association* In stage 2 the supplicant will start authenticating itself with the authenticator and vice versa. After this action has been completed the supplicant will continue by associating with the network. The entire process is shown in figure [2.3]. There are four authentication methods defined in the 802.11i standard but here we will consider only one: the Open System authentication, that allows any supplicant to successfully authenticate. When considering a network using RSN security the real authentication is usually performed during stage 4, specifically with the 4-Way Handshake. In fact, once the supplicant has successfully authenticated itself, the association will be done by sending a request to the AP informing it about the features supported. It's important to notice that at this level

[3]Robust Security Network Information Element

pairwise and group chiper suite are chosen and so that has to be used by the participants. This choice is encoded inside the RSNE element.

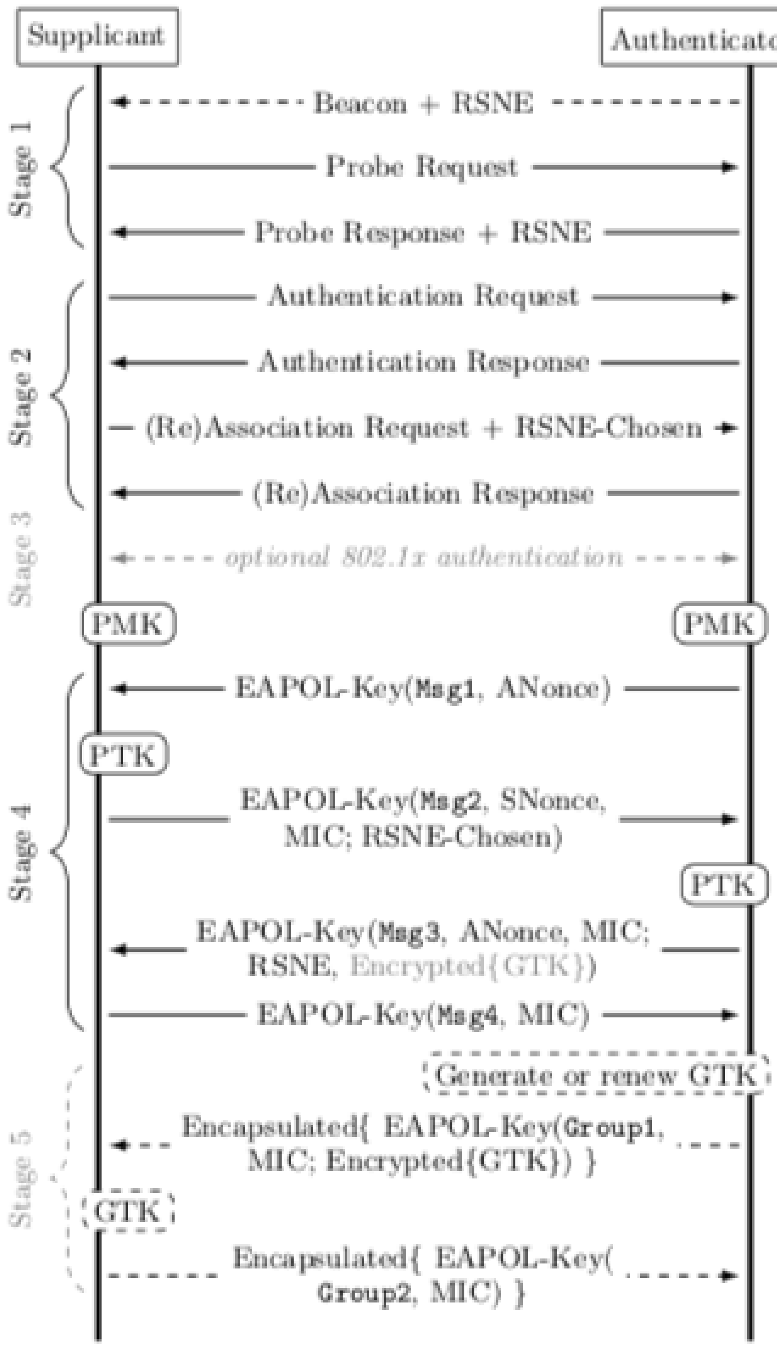


Figure 2.3. Frames sent in the 802.11 handshake

Stage 3: *802.1x Authentication* This stage is optional and it consists of 802.1x authentication to a back-end Authentication Server. In this phase the authenticator is in charge of acting as a kind of relay between the supplicant and the Authentication Server. Being that 802.1x is commonly used in enterprise networks, using a username and password for the login will result in the exchange of the shared secret between supplicant and authenticator. This shared secret is called the Pairwise Master Key or in short PMK.

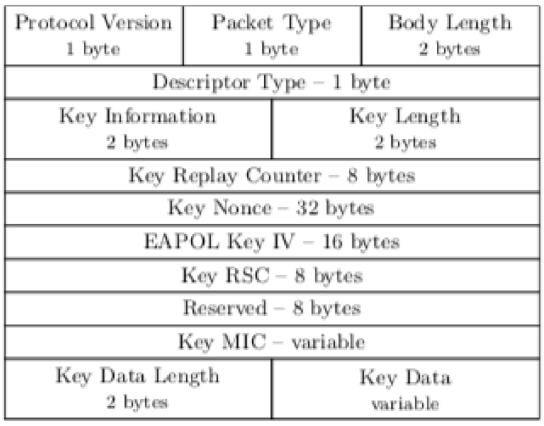
Stage 4: ***4-Way Handshake*** In this stage the formal 4-Way Handshake is performed in order to provide mutual authentication based on the Pairwise Master Key (PMK). That provides detection functions on possible downgrade attacks and negotiates a fresh session key called Pairwise Transient Key (PTK). Downgrade attacks can be detected by cryptographically verifying the RSNEs that have been received during stage 2. The PTK is derived from the Authenticator Nonce, Supplicant Nonce, and the MAC address of both supplicant and authenticator. The 4-Way Handshake generally also provides a Group Temporal Key from the authenticator to the supplicant. The 4-Way Handshake can be defined using EAPOL-Key frames, shown in figure [2.4]. The descriptor type determines the structure of the EAPOL-Key frame itself. Following this 5 byte header we find the key information field consisting of 3-bits key descriptor version and 1 bit for the key info flags. The key descriptor field, instead, defines the cipher suite that is in use in order to protect the frame. As said, the choice of this is negotiated in the first stage. The replay counter field is used to detect if any replay has been done. In fact, when the supplicant replies to an EAPOL-Key frame, coming from the authenticator, it must use the same replay counter that has been received previously proving its veracity. Eventually, the integrity of the frame is protected by using the Message Integrity Check, or MIC, and the key data field is encrypted in case it contains sensitive data. During this stage we found four messages:

Message 1 : sent by the authenticator and containing randomly generated ANonce of the authenticator itself. Here the two information flags need to be set and they are Pairwise and Ack, represented by label Msg1. It is important to notice that, at this level, there is no protection by a MIC. Hence, it could be possible for an attacker to forge this kind of message. Once the supplicant has received the message and is aware of the ANonce it is possible for it to compute and derive the PTK.

Message 2 : contains the random SNonce of the supplicant and is going to be protected by a MIC. In this message the key information flags to set are Pairwise and MIC, represented by label Msg2. Here it's important to note that in the Key Data Field of the EAPOL message the RSNE element is stored - that contains the cipher suite chosen by the supplicant in the association request sent before. Once the authenticator has received this message, is able to calculate the PTK, verify the MIC and check if any miss-match occurs between the old RSNE and the fresh one that has been received. If any discrepancy occurs the protocol is aborted instantly.

Message 3 : sent from the authenticator in response to the supplicant, it again contains the ANonce and secure by the MIC. The required key information flag here are Pairwise, MIC and Secure (labelled in Msg3), while the Key Data field will contain the supported cipher for the AP, wrapped into the RSNE. In case WPA2 is used the GTK is also included. When the supplicant receives this message, as done previously by the authenticator, the RSNE is checked with the one received when the protocol first took place. If the the two RSNE differ then a downgrade attack has been detected and the handshake is aborted.

Message 4 : this is the last message and it is sent by the supplicant in order to inform the authenticator that the handshake has been completed in a proper way. As the previous message, even this one is protected by a MIC and, the key info flags are Pairwise, MIC and Secure labelled as Msg4. Finally, once the authenticator has successfully received the message 4, encrypted data frames can be transmitted.



Protocol Version 1 byte	Packet Type 1 byte	Body Length 2 bytes
Descriptor Type – 1 byte		
Key Information 2 bytes	Key Length 2 bytes	
Key Replay Counter – 8 bytes		
Key Nonce – 32 bytes		
EAPOL Key IV – 16 bytes		
Key RSC – 8 bytes		
Reserved – 8 bytes		
Key MIC – variable		
Key Data Length 2 bytes	Key Data variable	

Figure 2.4. Layout of Eapol-Key frame

Stage 5: ***Group Key Handshake*** This last stage is required when a WPA1 is used to transport the group key to the supplicant and it is used to protect from broadcast and multicast traffic. The procedure is also used to periodically renew the group key. The figure [2.4] shows the steps that regulate the 4-Way Handshake during the connection establishment [11].

2.4 The KRACK attack

After a brief overview on how the WPA2 protocol works, it is important to understand in which way it is possible to attack and exploit its vulnerabilities. Therefore we introduce the attack. The Key Reinstallation Attack (KRACK) abuses main features of protocol to reinstall an already in-use key by resetting the nonces and/or the replay counters associated to it. In its 14 years lifetime the 4-Way Handshake has been thought to be secure due to the fact that no weaknesses were discovered during this period. Additionally, it was actually proven to be secure. But in this kind of attack, the adversary can trick the victim in to reinstalling an already in-use key without the latter even being conscious of it.

Nowadays all the Wi-Fi networks that are used worldwide have a WPA2 protection as a defence mechanism against possible attacks. This kind of technology relies obviously on the 4-Way Handshake defined in the standard 802.11i. As mentioned previously, the 4-Way Handshake provides mutual authentication and also a session key agreement for the authenticator and the supplicant. This can be achieved thanks to the (AES-)CCMP, data-confidentiality and integrity. Since its introduction in 2003 no breaches in the protocol were found, apart from the weaknesses encountered by using TKPI (that was uniquely intended as short term solution)[12][13].

The idea behind the attack actually is more trivial than one can imagine and can be sketched in the following way: Consider a client joins a network and the 4-Way Handshake is executed in order to negotiate a fresh session key. This key is usually installed after message 3 has been received and it will be used to encrypt data frames using a confidentiality protocol. At this point, it is important to notice the fact that messages can be lost or even dropped. Hence the AP, in this case, is in charge of retransmitting the message if it does not receive a proper response as acknowledgement from the supplicant. As a direct result, message 3 can be received multiple time. Each time it will reinstall the same session key, resetting the nonce and the receive replay counter used by the data-confidentiality protocol. Now what an attacker can do is force these nonce resets, by collecting and replaying transmission of message 3, and in this manner the data-confidentiality protocol can be exploited and attacked. For example, the packets sending through the channel are subject to replay, decryption and/or forging. The same technique is also used to attack the group key, PeerKey and the FBSS transition handshake. The attack described above turned out to be a devastating attack against version 2.4 and higher of "wpa_supplicant" - one of the Wi-Fi clients most commonly used in the Linux system. Since Android uses a modified version of the wpa_supplicant all the Android versions from the 6.0 to the newer ones are affected.

2.4.1 Supplicant state machine

In order to understand how the attack is mounted, it is necessary to have a look at the supplicant state machine and explain how the attack is executed in practice. The 802.11i amendment does not come with a formal state machine, that explains how the supplicant should be implemented, it just provides a pseudo-code that describes how, but not when, a certain message should be processed and handled [14]. In one of its extensions, precisely the 802.11r, the 4-Way Handshake protocol came along with a detailed state machine of the supplicant showed in figure [2.5]. When a connection to a network takes place for the first time and the 4-Way Handshake starts, the supplicant transitions to a PTK-INIT state. In this way it initializes the PMK and, after receiving message 1, it transitions to the PTK-START stage. This could occur in two situations: when connecting to a network for the very first time or when the session key is being refreshed after an already completed 4-Way Handshake. As soon as the supplicant enters in the PTK-START stage it generates a random SNonce, computes the TPTK (Temporary PTK) and then sends its nonce to the authenticator by using message 2. At this point the authenticator will reply using message 3, which will be accepted by the supplicant only if the MIC and the reply counter are valid. If it is the case, then the supplicant transits to the PTK-NEGOTIATING state, in which it marks the TPTK as valid and sends message 4 as reply to the authenticator. Then it moves to the PTK-DONE state, where the

PTK and GTK are installed for usage by the data-confidentiality protocol. At the end of this process the 802.1X port is opened and the supplicant is able to receive and send normal data frames. Here it is possible to notice that either message 1 or 3 can be retransmitted when message 2 or 4 are not received by the authenticator. These retransmissions are possible thanks to the EAPOL replay counter.

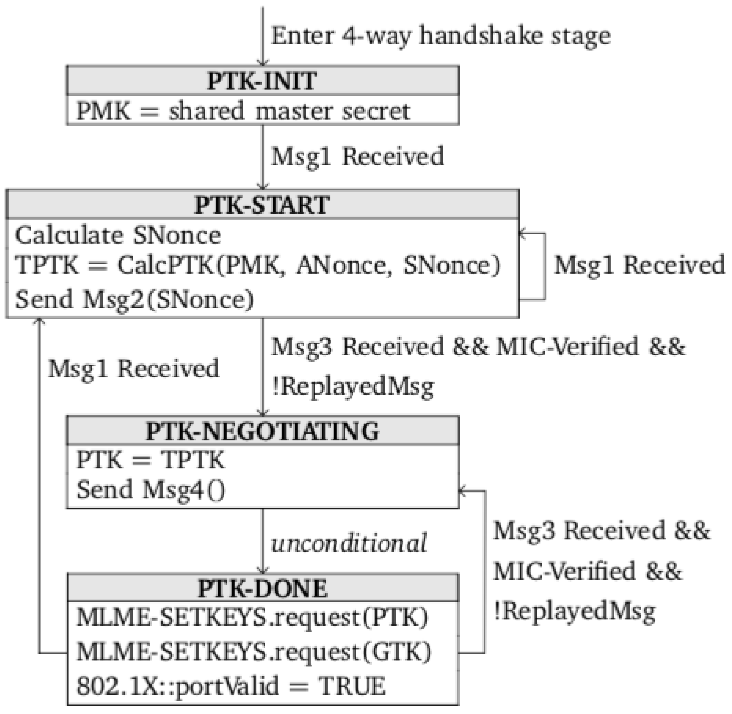

Enter 4-way handshake stage
PTK-INIT
PMK = shared master secret
Msg1 Received
PTK-START
Calculate SNonce
TPTK = CalcPTK(PMK, ANonce, SNonce)
Send Msg2(SNonce)
Msg1 Received
Msg1 Received
Msg3 Received && MIC-Verified && !ReplayedMsg
PTK-NEGOTIATING
PTK = TPTK
Send Msg4()
unconditional
Msg3 Received && MIC-Verified && !ReplayedMsg
PTK-DONE
MLME-SETKEYS.request(PTK)
MLME-SETKEYS.request(GTK)
802.1X::portValid = TRUE

Figure 2.5. Supplicant State Machine

2.4.2 Key Reinstallation Attack

After taking into account the functioning of the state machine it is now possible focus the attention on the attack itself. Since the supplicant accepts retransmissions of message 3, even when it is in the PTK-DONE stage, it is possible to force the reinstallation of the PTK. Specifically, the attacker first establishes a man-in-the-middle (MitM) position between the supplicant and the authenticator. Then, by exploiting his advantage, he can trigger the retransmissions of message 3 by preventing message 4 from correctly landing to the authenticator. As a consequence, it will be brought to retransmit message 3 and the supplicant will be forced to reinstall an already-in-use PTK. In turn, this will reset the transmit nonce and the received replay counter that is currently used by data-confidentiality protocol. Here, depending on which protocol is used, the attacker is allowed to replay, decrypt and/or forge packets freely.

Of course, there are some situations in which the attack will fail, briefly reported here: Since not all the Wi-Fi clients are properly using the state machine, i.e.

Windows and iOS 10, they do not accept retransmissions of message 3 at all. Hence the attack cannot be carried out successfully. The fact that those clients violate the standard itself resulted in invulnerability to the key reinstallation attack. However, they are still vulnerable to attacks against the group key handshake[15].

The second case is that, as it is necessary to establish a MitM position between supplicant and authenticator, it appears to be impossible to set up a rogue AP with a different MAC address. Since the session key is clearly based on the MAC address of the client and the AP this would lead to a different key being derived, thus causing the abortion of the handshake and the failure of the attack. Exploiting a channel-based MitM attack [16] would change the situation. Cloning the AP on a different channel, but with the same MAC address, will assure that both client and AP will surely derive the same session key, making the attack still possible.

Finally, certain implementations accept only frames that have been protected using the data-confidentiality protocol after the PTK has been installed. This could brought to a retransmission of message 3 without any encryption causing the drop of the message by the supplicant. Again, this could be by-passed with some technique [15].

Plaintext Retransmission of message 3

In the scenario presented above, if we consider the victim is still accepting the plaintext retransmissions of message 3 after it has already installed the session key, it is easy to understand how the key reinstallation attack is straightforward. In the initial step, the attacker can use a channel-based MitM attack through which he can manipulate the handshake messages [16]. Soon after this operation is set up, he can prevent message 4 from arriving to the authenticator, showed in figure [2.6]. As soon as message 4 has been sent the victim will install both PTK and GTK. At this point also the 802.1x port will be open. This will infer that, from now on, normal data frames can be transmitted. Then, in the third stage of the attack, the authenticator will retransmit message 3 because it did not receive message 4 since it has been intercepted by the attacker. At this point the latter forwards the retransmitted message 3 inducing the victim to reinstall the PTK and GTK. As result of this action, both replay counter and nonce used by data-confidentiality protocol are reset. It is important to underline here that the attacker cannot replay an old message 3 due to the fact that its EAPOL replay counter is no longer fresh and so it would be refused. In stage 4, the one in charge of completing the handshake from the authenticator side, since the victim has already installed the PTK the message is now encrypted making this step not so trivial. Plus, the authenticator will reject this message 4 due to the fact that its PTK is not installed yet.

In principle this should happen, but by carefully analysing the 802.11 standard has been noticed that the authenticator may actually accept *any* replay counter used in the 4-Way Handshake previously. Basically, some APs accept replay counters that were used in a message to the client, but were not yet used in a reply from the client. In that way those APs will accept the older unencrypted message 4 which has a replay counter r + 1, as shown in figure [2.6]. As result of this action, the PTK will be installed and the AP will start sending encrypted unicast data frames to the client. In the end, when the victim retransmits its next data frame, the data-confidentiality protocol will reuse nonces. The attacker has the faculty of managing both the forwarding time between messages and the amount of nonces that can be reused as well. Moreover, the client could be de-authenticated by the attacker, after which it will reconnect to the network performing a new 4-Way Handshake. In figure [2.6] it is possible to notice that, even though no attack has been mounted by third party, the KRACK attack can happen spontaneously if message 4 is lost due to background noise. Basically, this would happen when the client that accepts plaintext retransmissions of message 3 may already be reusing nonces with no one forcing it to do so [15].

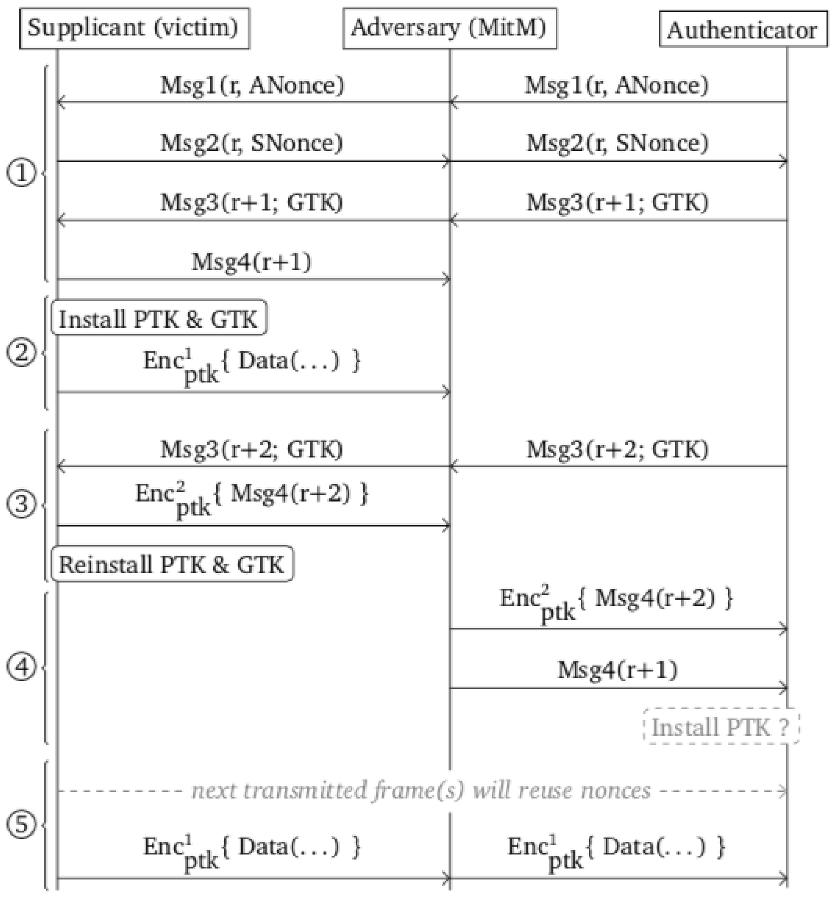


Figure 2.6. Key Reinstallation attack when the victim still accept message 3

2.4.3 All-zero key reinstallation

The focus should be brought on the behaviour that the attacks cause on the wpa_supplicant. In version 2.4 and 2.5 it is indeed possible to reinstall an all-zero encryption key when a message 3 has been received. The vulnerability seems to be caused by the fact that the standard 802.11 indirectly suggests to clear the encryption key from memory when it has been installed. However, when this bug has been patched, not all cases have been carefully considered. In fact, the only case that has been taken into account is a benign one, which is when message 3 is retransmitted because message 4 has been lost due to background noise. As the case of an attacker abusing this bug was not considered, an active adversary can abuse of the bug in order to force the reinstallation of an all-zero key. Nevertheless, it is possible to abuse the wpa_supplicant 2.6 by injecting a forged message 1 between the original and the retransmitted message 3 [17].

Chapter 3

Background on discovery techniques

3.1 Discovery techniques

As explained in the previous chapter, vulnerabilities in the last few years have become one of the main causes of threats in cyberspace security. A vulnerability is basically, as defined in RFC 2828[18], a flaw or weakness in a systems design, implementation, or operation and management that could be exploited to violate the systems security policy. Attacks on vulnerabilities, for example especially the ones made on zero day vulnerability, are seriously dangerous and sometime can be brought to critical situation as well.

Having in mind all the serious leaks of data and damages that have been witnessed over the years it appears clear why a lot of effort has been devoted to vulnerability discovery techniques in order to protect software and information systems. Various techniques can be used to achieve this goal such as static analysis, dynamic analysis, symbolic execution and fuzzing[19]. In this work the main focus will be on fuzzing because it requires less knowledge of the target and for this reason can be easily adapted and scaled to a large variety of situations and problems. As a matter of fact, fuzzing is the most popular vulnerability discovery solution.

3.1.1 Traditional discovery techniques

For the sake of completeness it is interesting to briefly go over all of the proposed techniques before getting into the fuzzing analysis details:

Static Analysis

Static analysis performs its work without actually executing the target programs. It is usually used to perform an analysis on the source code and seldom on the object code. Through the analysis of lexical, grammar, semantics features and data flow analysis, it is able to detect bugs that could be tricky to find in other ways. As its negatives, static analysis has a high detection speed and it could be able to quickly check the target code giving results. However, this speed brings with it a high false rate along with its hard-to-use model for vulnerability detection. The number of false positives is rather high and being able to identify the result of the analysis itself still remains hard work to perform.

Dynamic Analysis

As an opposite to the previous one, in the dynamic analysis of programs, the analyst has to execute the target problem in real systems or emulators. Dynamic analysis tools are able to detect program bugs neatly by monitoring the different running states and by performing an analysis on the runtime knowledge. One of the biggest advantages is the high accuracy rate that offers, but of course, on the other hand it requires a heavy human involvement both in terms of effort and in terms of knowledge and skills. So it is true that dynamic analysis presents a high speed, but low efficiency due to the lack of scalability and difficult to carry out large-scale testing. Both are serious problems to take into account.

Symbolic execution

Another vulnerability discovery technique that is to be considered promising is using symbols throughout the program inputs. The symbolic execution is able to maintain a set of constraints for each execution path. Eventually, constraint solvers will be used to resolve all the constraints that have been used and determine what the inputs are that have caused that execution. Such a technique has shown good effects in testing small programs but there are some limitations to take into account. For example, when the scale of the program tends to grow, the execution states explodes, causing problems to the constraint solvers that are not able to perform their job properly. Moreover, when the target program execution interacts with other components, that are out of the environment (i.e system calls, handling signal, etc.), some problems may arise. In other words, symbolic execution appears to still be difficult to scale up to large context and applications [20].

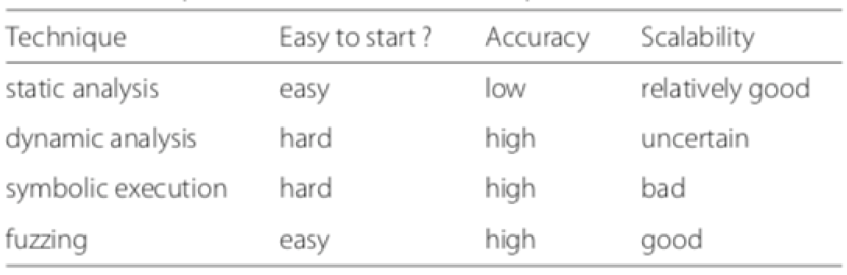
Fuzzing

The concept of fuzzing[1] was initially proposed in the 1990s[21], and even though the concept has remained unchanged, the way of how fuzzing is done has grown exponentially and evolved in the last few years. On the other hand, through the years it has been noticed that fuzzing tends to to find simple memory corruption bugs and cover a small part of the target code provided. This could result in a low efficiency while finding major or critical flaws. Using a combination of feedback-driven fuzzing modes and genetic algorithms can provide a more flexible and customisable fuzzing framework that makes the fuzzing process more clever and more efficient. For this reason, fuzzing is the most popular vulnerability discovery technique used

[1]Image source 3.1: https://medium.com/@nikhilh20/fuzzing-with-american-fuzzy-lop-quickstart-64d02579a6c5

Figure 3.1. Fuzzing

today. It starts with generating massive normal and abnormal inputs towards application and, at the same time, trying to detect exception by feeding generated inputs to the target applications and carefully monitoring each execution state. In comparison with other techniques, figure [3.2], fuzzing presents itself as easier in deploying and plus it allows us to perform analysis even without the source code. Another remarkable feature is that fuzzing tests are performed during real execution and for this reason it gains an higher accuracy. Moreover, it requires a very small knowledge of the target application, resulting in a scalability gain. Nevertheless, it presents disadvantages such as low efficiency and low code coverage, but this does not prevent it from still becoming the most efficient state-of-the-art vulnerability discovery technique. It is important to understand how the process works and how it is defined.



Technique	Easy to start ?	Accuracy	Scalability
static analysis	easy	low	relatively good
dynamic analysis	hard	high	uncertain
symbolic execution	hard	high	bad
fuzzing	easy	high	good

Figure 3.2. Comparison table

3.1.2 Fuzzing work process

The main process of the traditional fuzzing process has four main stages, that can be summarized in this way, figure [3.3]:

- Test case generation stage
- Test case running stage

- Program execution stage monitoring
- Analysis of exceptions

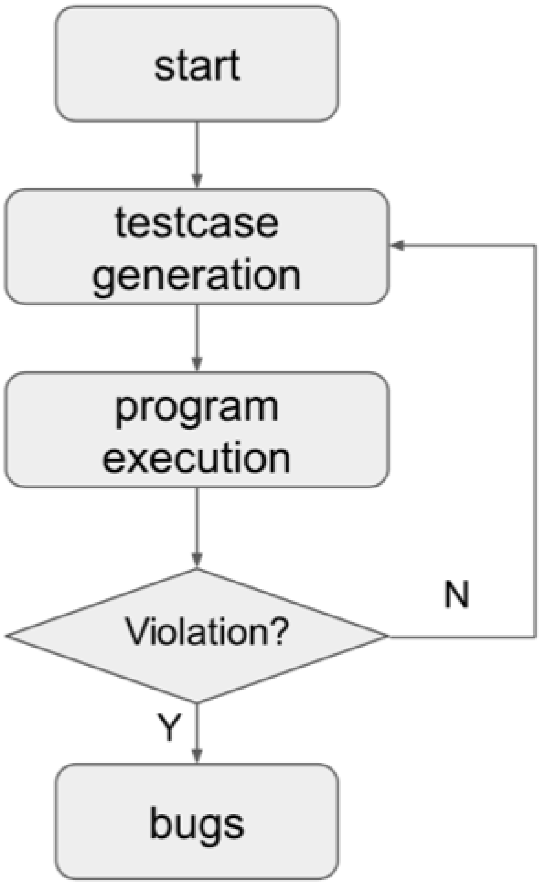


Figure 3.3. Fuzzing Stages

The fuzzing test is from the generation of several program inputs, usually defined as testcases. Of course this stage is relevant and very important since the quality of the generated testcases is directly linked with the effects of that tests performance. Provided inputs have to match the requirements of tested programs as much as possible. Nevertheless, they should be broken enough in order to make their processing good enough to force the program into a failure. Inputs can be of various types, e.g. different file formats, network communication data, executable binaries, etc. One of the main challenges resides in understanding how to generate broken enough testcases. In state-of-the-art fuzzers two kinds of generators are typically used and they are the generation based and mutation based.

Testcases are fed to target programs after they have been generated in the previous phase, then fuzzers automatically start the target program process until its end while driving the testcase to handle process of target programs. The analysts, before the execution takes place, are in charge of configuring the way the target programs start and finish and also pre-defining which parameters and environment variables have to be used. In a normal execution the fuzzing process should stop at a predefined time-out, program execution hangs or crashes. While running the fuzzer monitors the execution state during the execution of the program, seeking for exception and/or crashes. When some violation is captured by the fuzzer it stores it in the correspondent testcase result folder, that can be used later to replay the crash and perform further analysis. For some kinds of violations, that are not so straightforward to be analysed, it is possible to use several tools, i.e AddressSanitazer, DataFlowsanitazer, etc. In the subsequent analysis stage, the analyst could try to determine the location and also the root cause of the violations collected. This part of the job can be done by means of debuggers like GDB or otherwise.

As told before, a fuzzer can be classified in two big families:

- ***Generation Based***: with this kind of fuzzer, the knowledge of program input is strictly required, along with a configuration file that basically offer the guidelines to generate testcases. In fact, given a basic knowledge, the generated testcases are able to pass the validation of the programs easily, but it is essential to have a friendly document to do so, otherwise analysing the file format could become really tough work. This kind of fuzzer are easier to start and this makes them the more applicable and widely used today.
- ***Mutation Based***: requires a set of valid initial inputs so that testcases are generated throughout the mutation of the initial inputs and testcases generated during the first fuzzing process.

3.1.3 Types of Fuzzers

Another important aspect to consider is surely the type of fuzzers that exist. As mentioned, we have generation based and mutation based, but depending on the program source code and the degree of program analysis, fuzzers can be also divided and classified in white, grey and black boxes.

- White box are assumed to have all access to the source code and a lot of additional information can be gathered during the analysis on the source code and how test cases have affected the program running state.
- Black box perform fuzzing tests without any knowledge of the source code.
- Gray box also does not have source code awareness but the information is collected from a program analysis.

Another classification can be done based on the strategies adopted to explore programs. Thus we have directed fuzzing and coverage-based fuzzing.

- The goal of a directed fuzzer is to generate test cases that can cover the target code and the target paths of a program and lead to a faster test on programs;
- A coverage-based fuzzer tries to generate as much code of programs as possible and so expects a more thorough test along with discovering a bunch of bugs.

Still, both of them have a problem, a key challenge, that is how to extract information of executed paths. Fuzzers can be also classified as *dumb* or *smart*.

- Smart fuzzer are able to adjust the generation of testcases based on the collected information and how the testcases have affected the program behaviour. For example, considering a mutation based fuzzer, the feedback could be used to determine which and how to mutate part of the input in order to have better results.

- Dumb fuzzer does not require the input model and for this reason can be deployed to fuzz a wider variety of programs.

3.1.4 Key Challenges in Fuzzing

There are several challenges that todays fuzzer needs to face and solve:

- mutate seed input
- low code coverage
- passing the validation

The first one is generically solved by using mutation based generation, but in this case questions still arise. Where to mutate and how to do so? Exploiting only mutation on a few key points could affect the the control flow of the program, and how to locate this key "hot-spots" in test cases has importance. The way fuzzers mutate has an impacting importance, and so, for blind mutation it could end up seriously wasting testing resources while a more conscious mutation strategy could increase and improve the efficiency of testing significantly. In conclusion, by answering this question is possible to cover more program paths and is easier to trigger bugs. For the second challenge, the work done until now has proved that better coverage results in a higher probability of finding bugs. Despite that, most test cases only cover the same few paths, while most of the codes could not be reached and fairly analysed. So from this we can understand that it is not a sensible choice to achieve high coverage only exploiting a large amount of testcases generation and using them as testing resources. In order to solve this problem coverage-based fuzzers try to address this problem with the help of some program analysis technique, like code instrumentation. The last challenge is about validation. Programs, generally, validate inputs before parsing and handling. The validation works as a sort of guard of programs used to protect program against invalid input and damage that can be caused by maliciously constructed inputs. For example, invalid test cases are always discarded or simply ignored.

3.1.5 Coverage-Based Fuzzing - AFL

This kind of strategy is the most spread and used by state-of-the-art fuzzers and has been proved to be one of the more effective and efficient. The fuzzers try to traverse as many running states of the program as possible. Using this scheme it is still possible to have a loss of information and certain degree of uncertainty. AFL is one of kind of fuzzer that exploits this scheme of working. In the program analysis, the program itself is composed by basic blocks that are basically code snippets with a single entry and a single exit point. The instructions will be sequentially executed and executed only once. In code coverage measuring, state-of-art methods take basic block as the best achievable granularity. This is because of the fact that basic blocks are the smallest coherent units in a program execution. Measuring function would end up in an information loss or redundancy problem. Finally, basic block information can be easily gathered through code instrumentation. Nowadays, the two basic measurement choices based on basic block are either simply counting the executed one or counting the actual transition performed. In the latest method, what happens is that the program is seen and interpreted as a graph, where vertices are used to represent basic block while the edges turn to be transitions between basic block. In this way not only basic blocks are recorded, but also edges. This lead to a better coverage that could not be achieved with a simple basic block recording. AFL[2] is the first to introduce the edge measurement method into coverage-based fuzzing. AFL gains the coverage information via lightweight program instrumentation. Depending on the source code, AFL provides two instrumentation modes; the compile-in instrumentation and the external one. In the first one, AFL provides both gcc and llvm mode which will be able to instrument a code snippet while the binary code is generated. In the external mode, the qemu mode provided by AFL will instrument code snippets when basic block is translated to TGC block.

Working process

The algorithm in figure [3.4] shows how the coverage-based fuzzer operates. The test starts from the initial given seed input, if any, otherwise the fuzzer will construct one itself. In the main loop, interesting seeds will be chosen for the following mutation and testcases generation. The target program is guided to execute the generated testcases beneath the fuzzer monitoring. By doing so, all the testcases that will trigger crashes are going to be stored for future use, while other worthy ones are going to be added to the seed pool. The main loop stops at a pre-configured time-out chosen by the analyst, or a default one or when an abort signal is received. During the whole process, the fuzzer tracks the execution for two main reasons: code coverage and security violations. The first one is used to pursue a deep exploration of the various states of the program, while the latter is better for bug finding.

The target application is instrumented before execution for the coverage collection and this instrumentation can be done via either compile time, or as external, with gcc/llvm mode and qemu mode. After a seed has been provided to AFL, in the main fuzzing loop, the fuzzer chooses a favourite seed from its pool according to the seed selection strategy. Usually the smallest and fastest are preferred and picked up. After that seed files are mutated, according to the mutation strategy, and a certain number of test cases are generated. The last step is that test cases are executed and the execution is under tracking. Obviously, coverage information is collected in order to determine which are the interesting test cases (for example the ones that reach new control flow edges), to add them to the seed pool and exploit them for the next round of execution. The working process is shown in figure [3.5].

[2]American Fuzzy Lop

```
Algorithm 1 Coverage-based Fuzzing
Input: Seed Inputs S
T = S
Tx = ∅
if T = ∅ then
    T.add(emptyfile)
end if
repeat
    t = choose_next(T)
    s = assign_energy(t)
    for i from 1 to s do
        t' = mutate(t)
        if t' crashes then
            Tx.add(t')
        else if isInteresting(t') then
            T.add(t')
        end if
    end for
until timeout or abort-signal
Output: Crashing Inputs Tx
```

Figure 3.4. Fuzzing Algorithm

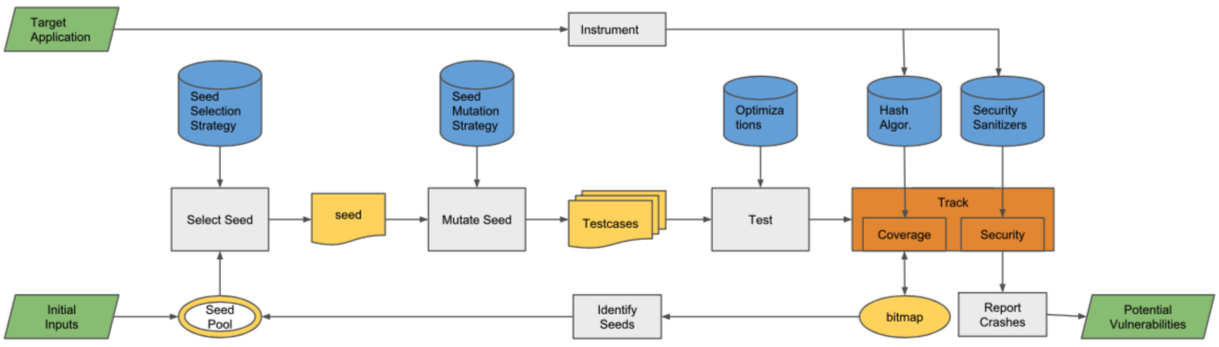


Figure 3.5. Fuzzing Working Process

3.1.6 Fuzzing of protocols

Fuzzing technique has been used to detect several vulnerabilities on different applications since it was invented. According to the different characteristics of the target application, it is possible to use different fuzzers that can apply a certain strategy to face different problems. Since in this work fuzzing is intended to be used against a protocol, it is useful to mark how it can be done in this context. In the recent past, lots of local applications have been transformed into network services in a B/S mode. In this configuration, services are deployed on a network and client applications can communicate with those exploiting network protocols. For this reason, security testing on network protocols became more and more crucial and a source of concern. Security flaws in protocols could be much worse than local applications flaws, in fact they can result in serious damages, like DoS (denial of service) attack, information leakage, defacement and so on. Several challenges are bound to fuzzing protocols, the first one is that services may define their own communication protocols - which are difficult to determine protocol standards. Plus, even for documented ones it is still hard to follow a specification such as a RFC document [22].

3.2 American Fuzzy Lop

As extensively discussed above, fuzzing is one of the most powerful and proven strategies for identifying security issues in real-world software and AFL is responsible for the vast majority of remote code execution and privilege escalation bugs found to date in security-critical software. A brief explanation on the main characteristics are given below.

Design goal of AFL-Fuzz

Speed : AFL is meant to let fuzzing most of the given targets at approximately their native speed.

Rock-solid reliability : it is designed to be just a very good traditional fuzzer with a wide range of interesting and useful strategies.

Simplicity : AFL is designed to allow a better understanding of the testing framework. It is possible to manipulate the output file, the memory limit and the ability to override the default time-out. In case of failure warning messages are shown to the tester outlining the possible causes and workarounds.

Chainability : AFL allows testers to use more lightweight targets in a way to create a small corpora of test cases that can be fed into a manual testing or a UI harness.

Approach to the problem

American Fuzzy Lop is a **brute-force** fuzzer coupled with a simple, yet solid, instrumentation-guided genetic algorithm. It uses a modified form of edge coverage to effortlessly pick up subtle changes that are possible in a program control flow. The algorithm basically works in 6 steps, that are the following:

1. Load initial test cases provided by the user into the execution queue,
2. Retrieve the input file from the queue,
3. Attempt to trim the test case to the smallest size in a way that does not alter the behaviour of the target program,
4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,
5. If any of the generated mutations ended in a new state transition recorded by the instrumentation the mutated output is then added as a new entry in the queue and later processed.
6. Go to 2.

After test cases have been discovered some of them can be pulled out and create room for newer ones. As a side result of the process, the tool is able to create a small self-contained corpus of interesting testcases. These are very useful in order to seed other.

3.2.1 Instrumentation

The instrumentation can be carried out by a companion tool that works as replacement for *gcc* or *clan* in any standard build process for code deriving from a third party. Of course this can be done when the source code is available for the analyst. The instrumentation presents a humble performance impact and, along with other optimizations made by *afl-fuzz*, a consistent number of programs can be fuzzed as fast, or sometimes even faster, than is possible with traditional tools. In case the source code is not available, the fuzzer also offers an experimental support for a fast and on-the-fly instrumentation of black-box binaries. This can be done by the exploiting of Quick EMUlator (QEMU).

3.2.2 Initial test case selection

In order to get valuable results, the fuzzer could require one or more starting input file that should contain a well-formed good example of the input data. Data are normally expected to be read from the target application. Generally, it is advised to keep the files small and to avoid using multiple testcases unless they are functionally different from each other.

3.2.3 Fuzzing binary

The fuzzing process itself is carried out by the afl-fuzz utility. This program requires a read-only directory with initial test cases, a separate place to store its findings, plus a path to the binary to test. AFL gives the possibility to read input data both from stdin and from file. Clearly, this choice is left to the analyst and it depends on which kind of target program is under analysis. Is also possible to modify the timeout and the memory limit for the process in execution, giving a certain degree of freedom in order to be scalable.

3.2.4 Understand output

Once afl-fuzz is on execution, the process will continue until an abort signal is sent from the user. It is advisable to let the fuzzer complete, at the very least, one queue cycle. This could take anywhere between a couple of hours and up to a week. It is up to the analyst to decide when a process may be considered completed. Three subdirectories are created within the output directory and updated constantly in real time:

queue : testcases for each distinct execution path, in addition to all the starting files provided by the user.

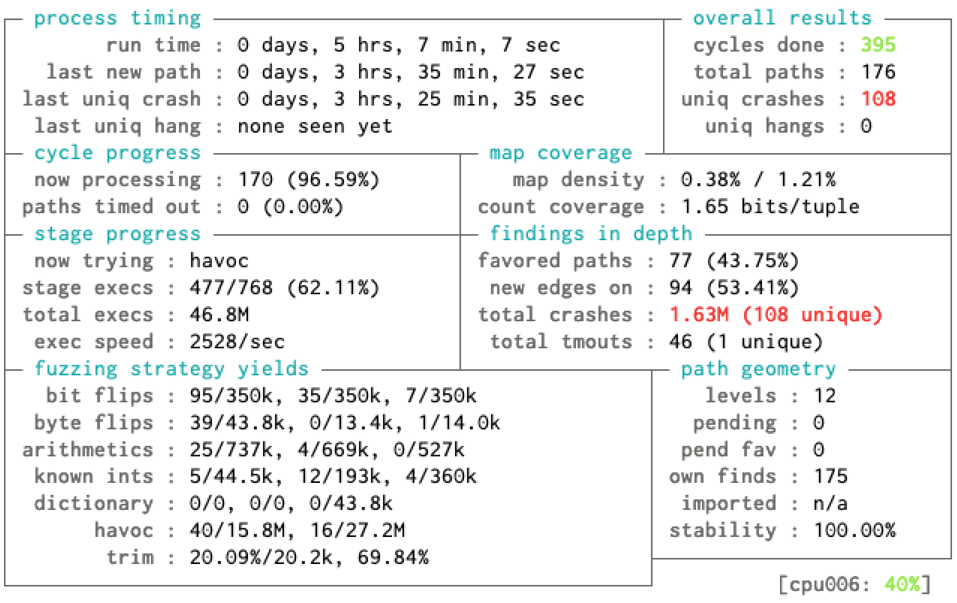
crashes : unique testcases that have caused a fatal signal received in the target program. These entries are grouped by the received signal.

hangs : unique testcases that have caused the target program to time out. The default time out limit before something falls under the classification of hang is the larger of 1 second and the value of the provided time parameter.

If there is a state transition that has not been recorded in a previous execution as a fault then both crashes and hangs have to be considered as *unique*. It could happen that a single bug can be reached in multiple ways. This will cause a count inflation in the process in the early stages, but should quickly be resolved and the number should decrease. The file names generated for crashes and hangs are obviously related to their parent file, in this way it is possible to speed up and also help the debugging process.[23]

3.2.5 Status screen

The status screen that is available for the tester during the process is represented in figure [3.6]. It is divided in 8 boxes, each one showing useful informations with respect to the current execution.



```
┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 5 hrs, 7 min, 7 sec        │  cycles done : 395    │
│   last new path : 0 days, 3 hrs, 35 min, 27 sec      │  total paths : 176    │
│ last uniq crash : 0 days, 3 hrs, 25 min, 35 sec      │ uniq crashes : 108    │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 170 (96.59%)      │    map density : 0.38% / 1.21%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.65 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 77 (43.75%)            │
│ stage execs : 477/768 (62.11%)      │  new edges on : 94 (53.41%)            │
│ total execs : 46.8M                 │ total crashes : 1.63M (108 unique)     │
│  exec speed : 2528/sec              │  total tmouts : 46 (1 unique)          │
├─ fuzzing strategy yields ───────────┴───────────┬─ path geometry ────────────┤
│   bit flips : 95/350k, 35/350k, 7/350k          │    levels : 12             │
│  byte flips : 39/43.8k, 0/13.4k, 1/14.0k        │   pending : 0              │
│ arithmetics : 25/737k, 4/669k, 0/527k           │  pend fav : 0              │
│  known ints : 5/44.5k, 12/193k, 4/360k          │ own finds : 175            │
│  dictionary : 0/0, 0/0, 0/43.8k                 │  imported : n/a            │
│       havoc : 40/15.8M, 16/27.2M                │ stability : 100.00%        │
│        trim : 20.09%/20.2k, 69.84%              ├────────────────────────────┘
└─────────────────────────────────────────────────┘          [cpu006: 40%]
```

Figure 3.6. Status screen

Processing timing : shows how long the fuzzer has been running and how much time has elapsed since its most recent finds.

Overall results : shows the number of times the fuzzer went over all the interesting test cases found so far, successfully fuzzed them and looped back.

Cycle progress : shows how far along the fuzzer is with the current cycle along with the test case ID.

Map coverage : shows the number of branch tuples hit and variability.

Stage progess : gives an in-depth look on what the fuzzer is doing at the moment and which strategy is under usage.

Findings in depth : shows metrics about the number of paths that are mostly liked by the fuzzer and number of test cases that have reached a better edge coverage.

Fuzzing strategy yields : shows how many paths have been explored with respect to the number of attempts for each strategy.

Path geometry : shows the path depth reached, how many inputs are still queued, how many are favourites, the number of new paths found imported from parallel fuzzing and the stability of the process

Chapter 4

Fuzzing IWD

In this chapter it will be discussed how it is possible to prepare and adapt an Inet Wireless Deamon to be analysed through fuzzing. For instance, a brief introduction on IWD architecture will be given. Subsequently the focus will be on how choose and generate a suitable input seed for the fuzzer, followed by the methodology on how to turn unit tests into test harness codes able to be fuzzed by AFL.

4.1 iNet Wireless Daemon

iNet Wireless Daemon (also known as IWD) is a project aiming to provide a comprehensive Wi-Fi connectivity solution for Linux based devices. The ultimate goal of the project is to somehow optimize resource utilisation such as storage, runtime memory and link-time costs. In fact, this is accomplished by not depending on any external libraries and utilises features provided by the Linux Kernel. This results in a self-contained environment being developed which only depends on the Linux Kernel and the runtime C library[24]. In this work we are going stress part of this library, specifically the implementation related to 4-Way Handshake implementation.

4.1.1 IWD Architecture

The IWD architecture is rather big and complex due to the fact that crucial operations are handled in order to assure security robustness. For sake of completeness a high-level structure of the architecture is reported below, showing where IWD is placed [4.1].

For our analysis, the main focus is on a specific section of the entire implementation. What is considered to be interesting for this research can be found in the *src* and *unit* directories of the IWD library. The implementation that involves the cryptography, the EAPOL frame handling and the 4-Way Handshake management (in other word the supplicant itself) resides in the *src* directory. In the *unit* directory, as the name suggests, the unit tests are developed to ensure the correct functioning is implemented. As it is, this code could be analysed by the fuzzer

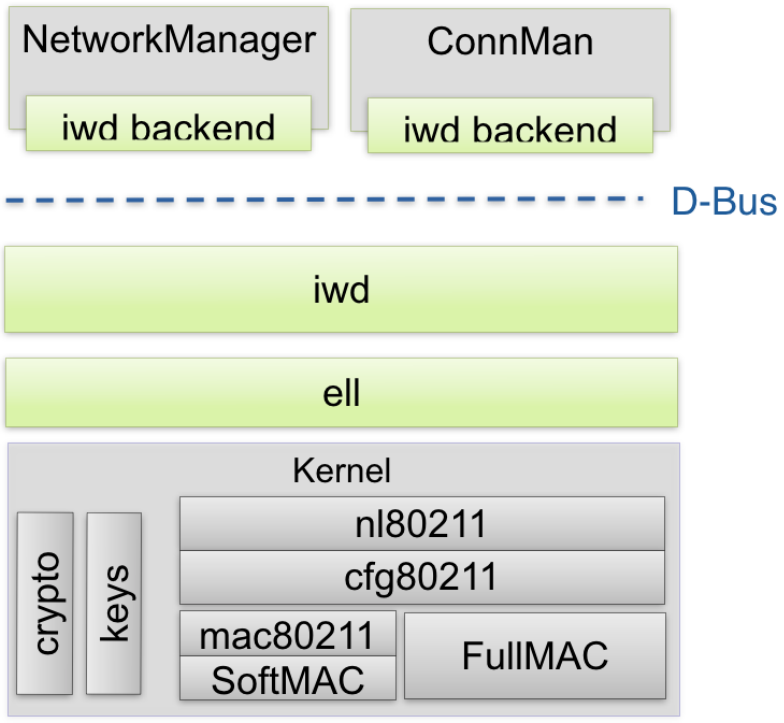


Figure 4.1. IWD Architecture

using the provided binary mode execution. Nevertheless, the likelihood of finding actual vulnerabilities is very low. This is due to the fact that this library has been developed with the intention of being highly secure and patching all the problems discovered lately. Therefore, the chance of finding software vulnerabilities is indeed very low. Fortunately, the source code is an open-source software, hence we have access to it, meaning that it is possible to modify. This allows for the creating of a harness code and to perform a more deep and precise analysis.

The general idea is to take the unit tests as a starting point and develop a test harness to automatised software vulnerabilities discovery through the use of a fuzzer. The unit tests are various, from MIC computation to the several cases of the 4-Way Handshake. Three significant tests have been chosen and analysed with the intention of assessing the correctness of the implementations. Dealing with these unit tests and making them able to be fuzzed requires a certain number of steps to be applied before the fuzzer can function properly.

4.2 Preparing IWD to Fuzzing

The first necessary step to understand, to prepare the IWD source code to be fuzzed, is to gain a general understanding of the source code available. Starting from this point, the main focus is to identify possible target functions where the message exchange of the 4-Way Handshake is performed, and build a *harness code* that can guide the fuzzer over the target program. A test harness is generally used to enable the automation of tests. It can provide stubs and drivers which can be small programs, or code snippets, that interact with the software currently under test. Usually test harnesses execute tests by exploiting a test library and generating a final report. In our context, we can consider the used stub code as an entry point that allows the fuzzer to provide its own input data and explore the targeted code. On top of that, the final goal to achieve is to inject malformed payloads during the message exchange and verify how the underlying structure reacts to them. Applying this strategy, we will try to trigger eventual vulnerabilities that would otherwise be hard to spot.

4.2.1 Input seed selection

One of the first challenges that must be faced is the input seed selection. The importance of this process is essential. The quality of test cases affects the efficiency and the effectiveness of fuzzing testing. A good test case can explore more program execution states and cover a reasonable amount of code in a shorter time. Besides, using good test cases could target potential vulnerable locations and provide a faster discovery of programs bugs. As a matter of fact, selecting an input that is already known to be a cause of crashes for the target program has no meaning, and the fuzzer will not accept it in any case. On the other hand, selecting a meaningless input could lead to a inconclusive analysis since the path coverage achieved will never trigger potential bugs or locations of the code where a bug could reside.

The size of the input file is also an aspect that must be carefully taken into account. A huge input could not be processed correctly, leading to a dead end, and in most of the cases AFL will not allow it to be used. So, how can this issue be tackled? The general answer behind this question is to collect as much information as possible from the target program and devise a possible action plan. As we are interested in understanding if and how it could be possible to find software bugs in this context the idea is to identify an entry point where it is possible to inject an unexpected, or malformed, data input and monitor how the target code will react to it.

The starting point is, without any doubt, *test-eapol.c* file, in which unit tests are implemented. Going through this code it is possible to gain a useful knowledge and a general overview of how the tests are executed, how the EAPOL-Key frame is constructed and then eventually define how a suitable candidate input seed for the fuzzer could be generated. In the first section of this code EAPOL-frames are statically defined and each one of them is specifically bound to a test case each aiming to test a different behaviour. An example of how Eapol data frames are structured is shown below:

```
  static const unsigned char eapol_key_data_*[] = {
      0x01, 0x03, 0x00, 0x5f, 0x02, 0x03, 0x0a, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x9e, 0x57, 0xa4,
      0xc0, 0x9b, 0xaf, 0xb3, 0x37, 0x5e, 0x46, 0xd3, 0x86, 0xcf, 0x87, 0x27,
      0x53, 0x00, 0x00
};
```

Listing 4.1. Example of eapol data frame

The Eapol data frame seems to be a good starting point in order to produce a suitable input candidate. In the code, 32 different Eapol key-data frames are defined, each one of them has a different payload along with all the necessary information required to simulate a complete 4-Way Handshake protocol.

```
static struct eapol_key_data eapol_key_test_* = {
        .frame = eapol_key_data_4,
        .frame_len = sizeof(eapol_key_data_4),
        .protocol_version = EAPOL_PROTOCOL_VERSION_2001,
        .packet_len = 117,
        .descriptor_type = EAPOL_DESCRIPTOR_TYPE_80211,
        .key_descriptor_version = EAPOL_KEY_DESCRIPTOR_VERSION_HMAC_SHA1_AES
    ,
        .key_type = true,
        .install = false,
        .key_ack = false,
        .key_mic = true,
        .secure = false,
        .error = false,
        .request = false,
        .encrypted_key_data = false,
        .smk_message = false,
        .key_length = 0,
        .key_replay_counter = 0,
        .key_nonce = { 0x32, 0x89, 0xe9, 0x15, 0x65, 0x09, 0x4f, 0x32, 0x9a,
                       0x9c, 0xd5, 0x4a, 0x4a, 0x09, 0x0d, 0x2c, 0xf4, 0x34,
                       0x46, 0x83, 0xbf, 0x50, 0xef, 0xee, 0x36, 0x08, 0xb6,
                       0x48, 0x56, 0x80, 0x0e, 0x84, },
        .eapol_key_iv = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                          0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
        .key_rsc = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
        .key_mic_data = { 0x01, 0xc3, 0x1b, 0x82, 0xff, 0x62, 0xa3, 0x79, 0xb0,
                          0x8d, 0xd1, 0xfc, 0x82, 0xc2, 0xf7, 0x68 },
        .key_data_len = 22,
};
```

Listing 4.2. Eapol key data structure example

All this information is stored in a C structure, shown in [4.2], that is retrieved when the correspondent unit test needs to start the protocol and simulate the packets exchange. Our analysis involves the payload of the Eapol key-data frames making it possible to reuse all the other information without any further modification. That being said, we should be able to modify this payload with the help of AFL, applying an instrumented-guided genetic algorithm, mutating the entire data. This should be able to disrupt the normal behaviour and alter the usual control flow of the target. Once our input has been correctly identified, it is necessary to devise a solution to allow the fuzzer to process it in the best manner. In the next paragraph we will discuss how this can be done.

4.2.2 Input File Creation

AFL provides two different approaches for reading input data: it can receive an input either directly from *stdin* or read it from a file. The first option may be considered in a situation where the input to provided is fairly short, such as a string or similar. For our purpose the second choice is the most appropriate and reasonable. This means we will have to deal with an input that is made of more than a single data frame, which can be fairly long, just as it is. Since the message exchange between the supplicant and the AP involves more than one packet using files is the only sensible approach. Another issue to take into account here is the file structure to be used. AFL requires one or more starting files which should contain a good example of the input data that the target program would normally expect. Keeping that in mind the provided files should be fairly small and the usage of multiple test cases is encouraged only if they are functionally different from each other. AFL is able to read a single file for each execution, hence we should be able to store more than one data frame in the same file, in order to provide the expected data to the target program. The file needs to store both the data frame and its length since during the mutation the data frame is altered both in content and in size. Ideally the file can be structured in different way but here we have considered two different approaches, which both seem reasonable. Being d1 and d2 the data bytes of the payload, respectively for the first frame and for the second one, we can write those values as in figure [4.2]:

- A d1 data bytes followed by a delimiter, i.e. "===", and then d2 data bytes.
- The size of the first frame followed by the d1 data bytes, then the size of the second frame followed by d2 data bytes.

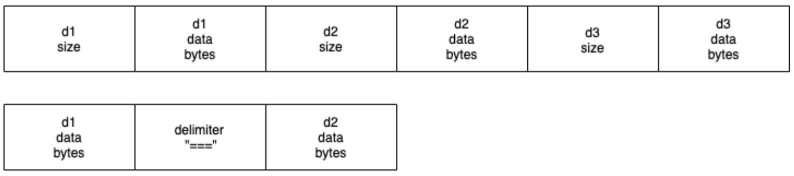


Figure 4.2. File format

Both these solutions are to be considered valid although in this work the second approach has been selected. In order to write the input binary file, an external utility code has been developed. This code simply allows us to re-use the data frames stored in the original source code and write them in a binary file that respects the constraints defined by AFL. The size of the frame is computed starting from the provided data frame array [4.1], and written in the file as type *size_t* as defined by the Eapol implementation. Subsequently the data frame array is iterated and each value contained into it is written to the file as an *uint8_t* type. The utility code can easily be adapted to handle more than two data frames. As a matter of fact, this will be used further on in this work to allow fuzzing on key reinstallation issues, where using three data frames is strictly necessary.

4.2.3 Harness Code

That being said, it is possible to start developing the harness code. As mentioned, the starting point is the *test-eapol.c* file which requires several modifications. Of course, the *main()* function is not implemented in a way that allows us to read any arguments from the command line due to the fact that all the data used by the unit tests are hard-coded. This kind of implementation is not suitable for our purpose so it is necessary to adapt the code and provide this feature along with other functions able to parse the data received from the outside. As a first step then, we need to modify the *main()* so that it can receive one parameter from the command line, that will be used by AFL, to provide the name of the file that must be used during each execution. The second step is to identify which functions could be targeted by the fuzzer. One of the functions that has been selected and tested is **eapol_sm_test_ptk()**. A code snippet of the function, reporting only the interesting part, is shown below

```c
static void eapol_sm_test_ptk(const void *data){
    [...]
    hs = test_handshake_state_new(1);
    sm = eapol_sm_new(hs);
    eapol_register(sm);

    eapol_sm_set_protocol_version(sm, EAPOL_PROTOCOL_VERSION_2001);

    handshake_state_set_pmk(hs, psk, sizeof(psk));
    handshake_state_set_authenticator_address(hs, aa);
    handshake_state_set_supplicant_address(hs, spa);

    r =  handshake_state_set_supplicant_rsn(hs,eapol_key_data_4 + sizeof(struct eapol_key))
  ;
    assert(r);

    handshake_state_set_authenticator_rsn(hs, ap_rsne);
    eapol_start(sm);

    __eapol_set_tx_packet_func(verify_step2);
    __eapol_rx_packet(1, aa, ETH_P_PAE, eapol_key_data_3,sizeof(eapol_key_data_3), false);
    assert(verify_step2_called);

    __eapol_set_tx_packet_func(verify_step4);
    __eapol_rx_packet(1, aa, ETH_P_PAE, eapol_key_data_5,sizeof(eapol_key_data_5), false);
    assert(verify_step4_called);

    eapol_sm_free(sm);
    handshake_state_free(hs);
    [...]
}
```

Listing 4.3. Code snippet of the original function

As suggested by the name, this function is used to test the installation of the Pairwise Transient Key, which is computed in the second and third step of the 4-Way Handshake during the authentication. This function will:

- initiate the state-machines, putting them in a queue;
- retrieve the supplicant nonce;
- set-up the the address for both the authenticator and supplicant;

- create a new handshake test and an Eapol state machine;
- simulate the message exchange involved in the communication.

The function *__eapol_rx_packet()* is the one which we are interested in. The call to this function will trigger a certain number of operations (i.e. key handle, MIC computation and so forth), to be discussed later on. The third and fourth parameter accepted by this function are indeed the data frame and size of the latter, which are no longer correct for our analysis. Both line 20 and 24 must be modified so that they can use the newly read data generated by the fuzzer. However, this modification is not sufficient. In fact, altering these packets will imply also that the messages sent back and forth from the authenticator and the supplicant cannot be verified properly anymore. The control for this is made on line 19 and 23 with the function *__eapol_set_tx_packet_func(verify_step*)*. The latter is in charge of simulating the expected response upon receipt of WPA frame 1 and WPA frame 3. A code snippet for one of the functions is reported below, to show how this is done.

```
static int verify_step2(uint32_t ifindex,
      const uint8_t *aa_addr, uint16_t proto,
      const struct eapol_frame *ef, bool noencrypt,
      void *user_data)
{
  const struct eapol_key *ek = (const struct eapol_key *) ef;
  size_t ek_len = sizeof(struct eapol_key) +L_BE16_TO_CPU(ek->key_data_len);

  assert(ifindex == 1);
  assert(proto == ETH_P_PAE);
  assert(!memcmp(aa_addr, aa, 6));
  assert(ek_len == expected_step2_frame_size);
  assert(!memcmp(ek, expected_step2_frame, expected_step2_frame_size));

  verify_step2_called = true;

  return 0;
}
```

Listing 4.4. Code snippet for verify_step2

In [4.4], we can see that the protocol, the address of the authenticator and both the frame and its size are compared with the one expected to be received. In case of any miss-match this function returns earlier with an error. Since the injection of malformed payload will cause these miss-matches in most cases, it is necessary to handle this case differently. We still need to check if a reply has been received but we cannot verify that is the exact one. Instead of using the asserts, that in case of failure will generate an error and return, we can substitute them with if-else statements. In doing so it will be still possible to check when the message has been received but in case of miss-match we keep the protocol alive and monitor in which way this is handled by the protocol.

AFL Entry Point

Once we gathered a sufficient knowledge about the target it is possible to devise and develop the entry point for AFL. The injection of our payload will have to define a new function that must be implemented and integrated in the source file of the target program. This newly created function will be used to:

- verify the existence of the file;
- restore the hard-coded parameters whenever the input file is missing;
- read the size of the frame from the file and store it into a variable;
- perform boundaries checks and fail early, aborting the test, for malformed files;
- read the data frame storing it into a variable;
- return to the calling function in case of success.

The code snippet below shows a small section of the newly created function [4.5].

```c
void * __afl_get_key_data_ptk( ){
    [...]
    ret = fread(&len_frame1, sizeof(size_t), 1, fp);
    if(ret <= 0 ){
  printf("\n Read len failed!\n");
      exit(EXIT_FAILURE);
  }
    [...]

    __afl_key = (uint8_t *)malloc(sizeof(uint8_t)* len_frame1);
    for (i=0 ; i <= len_frame1; i++){
        ret = fread(&data, sizeof(uint8_t), 1, fp);
       if(ret <= 0 ){
  printf("\n Read data failed!\n");
      exit(EXIT_FAILURE);
  }
   __afl_key[i] = data;
    }
    [...]
}
```

Listing 4.5. Code snippet of the new function

On return of this function, both the size and data frame will be read and stored in the correspondent variable ready to be used by the targeted function. The last step that is still missing is making our target function capable of processing the new input. Therefore we have modified the code showed in [4.3] by adding a call to new function. In the *__eapol_rx_packet()* function, the fourth and fifth parameters must be changed accordingly. Instead of using the EAPOL key frames defined in the code, the function has to use the data read from the file and stored into *_afl_key* and *len_frame*. Once all of these modifications have been done the fuzzing process can start.

4.3 Fuzzing for memory safety vulnerabilities

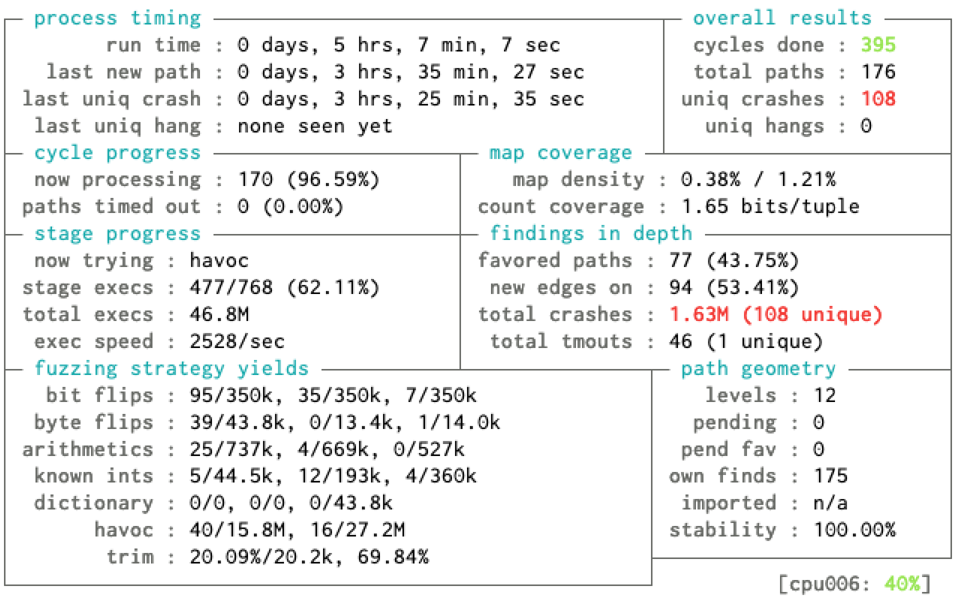
The target program is now almost ready to be fed to the fuzzer. Prior to execution, the IWD library needs to compiled by using a wrapped GCC compiler, known as **afl-gcc**. By doing so the whole code is both compiled and instrumented by AFL.

During the execution, AFL will provide its own UI status-screen, similar to the one shown in figure [3.6], enabling the monitoring of the performance of the fuzzer. Several runs can be done in parallel in order to gain as many results as possible, providing a full spectre analysis. The time that we have reserved for each run is variable and it goes from 4 hours up to 1.5 days.

There is no general rule for when we should interrupt the fuzzer, but in general referring to the "path geometry" and the colour-coded info in the "overall results" box, it is indeed possible to derive a good timing on whether to stop the analysis or keep it alive. For instance, when no pending paths will be shown and the "cycles done" information is green it is unlikely that any other new path will be found, thus indicating that the analysis is completed.

4.3.1 Crash Exploration

After collecting a sufficient amount of results, we begin to explore the crashes found. The first phase of analysis shows that, on average, the unique crashes found were between 80 and 110, depending on the time frame reserved for each execution. An example run is shown below in figure [4.3]:



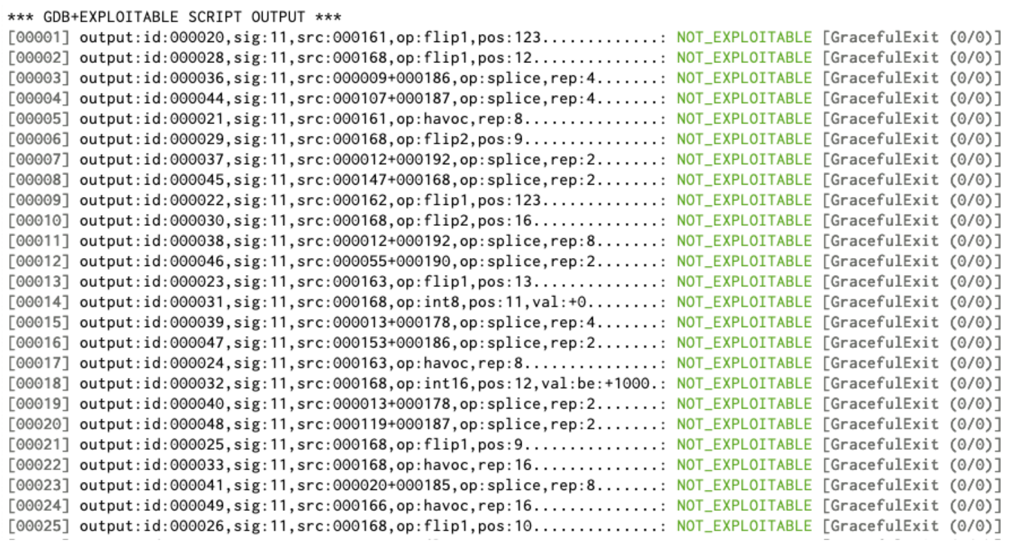
```
┌─ process timing ─────────────────────────────────────┬─ overall results ──┐
│        run time : 0 days, 5 hrs, 7 min, 7 sec         │  cycles done : 395 │
│   last new path : 0 days, 3 hrs, 35 min, 27 sec       │  total paths : 176 │
│ last uniq crash : 0 days, 3 hrs, 25 min, 35 sec       │ uniq crashes : 108 │
│  last uniq hang : none seen yet                       │   uniq hangs : 0   │
├─ cycle progress ──────────────────┬─ map coverage ─┴────────────────────┤
│  now processing : 170 (96.59%)    │    map density : 0.38% / 1.21%      │
│ paths timed out : 0 (0.00%)       │ count coverage : 1.65 bits/tuple    │
├─ stage progress ──────────────────┼─ findings in depth ─────────────────┤
│  now trying : havoc               │ favored paths : 77 (43.75%)         │
│ stage execs : 477/768 (62.11%)    │  new edges on : 94 (53.41%)         │
│ total execs : 46.8M               │ total crashes : 1.63M (108 unique)  │
│  exec speed : 2528/sec            │  total tmouts : 46 (1 unique)       │
├─ fuzzing strategy yields ─────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 95/350k, 35/350k, 7/350k            │    levels : 12         │
│  byte flips : 39/43.8k, 0/13.4k, 1/14.0k          │   pending : 0          │
│ arithmetics : 25/737k, 4/669k, 0/527k             │  pend fav : 0          │
│  known ints : 5/44.5k, 12/193k, 4/360k            │ own finds : 175        │
│  dictionary : 0/0, 0/0, 0/43.8k                   │  imported : n/a        │
│       havoc : 40/15.8M, 16/27.2M                  │ stability : 100.00%    │
│        trim : 20.09%/20.2k, 69.84%                ├────────────────────────┘
└───────────────────────────────────────────────────┘          [cpu006: 40%]
```

Figure 4.3. Fuzzer execution on eapol_sm_test_ptk

These numbers are extremely high for a security critical code. Consequently, an accurate investigation has been carried out. Clearly, the number of crashes found cannot be considered an accurate indicator of the fact that all the failures identified are indeed real vulnerabilities. It could be possible that during its execution the fuzzer may have encountered false positives and identified them as possible points of exploitation. Relaying on all the collected test cases produced by AFL it is possible to reproduce any single crash in a debug environment. The crashes and hangs can be manually examined within the output folders, where AFL has stored them, with the help of the GNU Debugger. On the other hand, this manual labor is neither fast nor very effective, hence we have made use of an AFL companion utility.

For instance, the *afl-utils* provides several shortcuts to render crash exploration easier and more automatic. The *afl-collect* tool allows us to copy all crash sample files from a specific AFL directory into a single location, providing an easy access for further crash analysis. Beyond that, afl-collect comes with advanced features, like invalid crash samples removing, as well as generating and executing, gdb scripts by using the **Exploitable** tool [25]. The latter is a GDB extension that is able to classify Linux application bugs by severity. What Exploitable does is to inspect the state of a Linux application that has crashed and outputs a summary of how difficult it might be for an attacker to exploit the software bug and gain control over the system [26].

To keep track of the different crash samples gathered during the several analyses, a database for each tested function has been created. Using the so obtained *crash database* we have been able to store all sample test cases and remove the ones already encountered during previous executions. By doing so, we have narrowed down the number of crash samples to verify. This ensures that only unique samples have been tested. After processing the available sample test cases, none of them have been marked as "exploitable"[4.4].



```
*** GDB+EXPLOITABLE SCRIPT OUTPUT ***
[00001] output:id:000020,sig:11,src:000161,op:flip1,pos:123.............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00002] output:id:000028,sig:11,src:000168,op:flip1,pos:12..............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00003] output:id:000036,sig:11,src:000009+000186,op:splice,rep:4.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00004] output:id:000044,sig:11,src:000107+000187,op:splice,rep:4.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00005] output:id:000021,sig:11,src:000161,op:havoc,rep:8...............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00006] output:id:000029,sig:11,src:000168,op:flip2,pos:9...............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00007] output:id:000037,sig:11,src:000012+000192,op:splice,rep:2.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00008] output:id:000045,sig:11,src:000147+000168,op:splice,rep:2.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00009] output:id:000022,sig:11,src:000162,op:flip1,pos:123.............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00010] output:id:000030,sig:11,src:000168,op:flip2,pos:16..............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00011] output:id:000038,sig:11,src:000012+000192,op:splice,rep:8.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00012] output:id:000046,sig:11,src:000055+000190,op:splice,rep:2.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00013] output:id:000023,sig:11,src:000163,op:flip1,pos:13..............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00014] output:id:000031,sig:11,src:000168,op:int8,pos:11,val:+0........: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00015] output:id:000039,sig:11,src:000013+000178,op:splice,rep:4.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00016] output:id:000047,sig:11,src:000153+000186,op:splice,rep:2.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00017] output:id:000024,sig:11,src:000163,op:havoc,rep:8...............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00018] output:id:000032,sig:11,src:000168,op:int16,pos:12,val:be:+1000.: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00019] output:id:000040,sig:11,src:000013+000178,op:splice,rep:2.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00020] output:id:000048,sig:11,src:000119+000187,op:splice,rep:2.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00021] output:id:000025,sig:11,src:000168,op:flip1,pos:9...............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00022] output:id:000033,sig:11,src:000168,op:havoc,rep:16..............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00023] output:id:000041,sig:11,src:000020+000185,op:splice,rep:8.......: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00024] output:id:000049,sig:11,src:000166,op:havoc,rep:16..............: NOT_EXPLOITABLE [GracefulExit (0/0)]
[00025] output:id:000026,sig:11,src:000168,op:flip1,pos:10..............: NOT_EXPLOITABLE [GracefulExit (0/0)]
```

Figure 4.4. GDB exploitable results

Despite the fact that the evaluation made by the tool was inconclusive, we considered it appropriate to study in-depth the obtained results. With the help of GDB we have manually tested the crashes sample one by one. Starting from the collection of samples stored in the database it is possible to run each of the samples against the targeted code and re-create the crashes generated. All the crashes were leading to a double free error in the code related to the state machine used during the 4-Way Handshake test. As a matter of fact, whenever an expected message is received, the *__eapol_rx_packet()* function is forced to interrupt the 4-Way Handshake and free the memory previously accommodated by the in-use state machine. Nevertheless, the unit test is also inducted to free the same state machine upon completion. To overcome this problem, a new function in the eapol.c source file has been implemented. The latter deals with this issue preventing the double free whenever a malformed packet cannot be processed. Upon completion of this task, further tests have been run to assess whether or not the fuzzer was actually executing correctly all the possible paths.

4.3.2 Understanding Fuzzer Performances

To better understand how the fuzzer was actually working on its target, we decided to add several *assert()* functions in different branches and paths to determine the effectiveness of the analysis. All the possible failure points of the functions involved during the messages exchange have been identified and instrumented as well. Indeed new tests have been performed to acquire and analyse new data. From the different executions we have been able to trace a list of all the crashes generated by our asserts and we have defined the causes as well. The table [**??**] shows our results:

Function	**Reason of the failure**
eapol_handle_ptk_1_of_4()	Verification of the PTK failed
	Key installation failed
eapol_calculate_mic()	MIC calculation failure
	Cipher suite cannot be computed
__eapol_rx_packet()	Packet size out of bound
_eapol_verify_mic	Invalid checksum computed for Key Descriptor
eapol_decrypt_key_data	Decryption of key data failed
eapol_key_handle()	Creation of PTK failed
eapol_eap_complete_cb()	Eapol time-out elapsed without any reply
__eapol_rx_packet()	Size of the frame out of bounds
	Cannot identify protocol version in use
	Not configured for EAP, NAK sent
	Unhandled EAP packet received, method miss-configuration

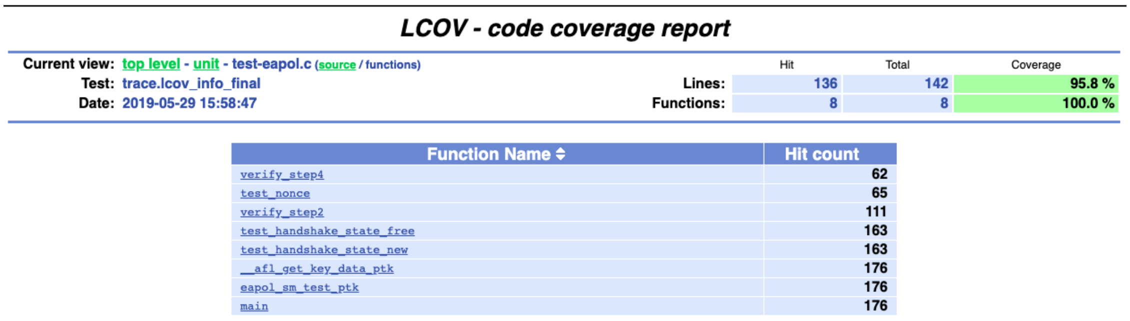
Table 4.1. List of identified crashes

From these crashes we can infer that fuzzer is correctly looking for vulnerabilities where they are expected to be. This enables us to perform now a complete analysis to assess if and where possible software vulnerabilities are present in the code. Anyway, the latest analyses performed on the unit test *eapol_sm_test_ptk()* have reported 0 crashes after several hours of fuzzing, which leads us to hypothesize that the code concerning this part of the 4-Way Handshake is indeed free from any software vulnerabilities.

4.4 Code Coverage & Metrics

On the top of what has been said, we can take it a step further and have a general view about metrics, specifically related to the code coverage. It can be defined as the number of lines of a source code executed during a given test suite for a program and usually expressed as a percentage. Code coverage brings with it plenty of benefits when performing code analysis and, above all, a better understanding on how we are dealing with the tested code. How can we do that based on AFL results? A companion tool can address this kind of issue. For instance, *afl-cov* uses test case files that have been produced by the fuzzer to generate the so called **gcov code coverage results** for a targeted binary. Code coverage is interpreted as a chain, so from one case to the next, in order to determine which new functions and/or lines have been hit by AFL with each new test case. It allows for specific lines or functions to be searched for within coverage and on any new match found the corresponding test case is displayed. Using this tool we can understand which AFL test case is the first to exercise a particular line or function. In the end, several reports will be produced, including also the zero coverage that will contain all the functions that were never executed while the fuzzer was up and running [27]. Producing code coverage data for AFL test cases is extremely important since it can help to try and maximize code coverage improving the effectiveness of AFL itself.

For all these reasons, code coverage analysis has been performed on all test cases produced by AFL. We report, as an early example, the code coverage obtained in one of the executions of the fuzzer to show the effectiveness of the analysis [4.5].


LCOV - code coverage report

Current view: top level - unit - test-eapol.c (source / functions)
Test: trace.lcov_info_final
Date: 2019-05-29 15:58:47

	Hit	Total	Coverage
Lines:	136	142	95.8 %
Functions:	8	8	100.0 %

Function Name ⬍	Hit count
verify_step4	62
test_nonce	65
verify_step2	111
test_handshake_state_free	163
test_handshake_state_new	163
__afl_get_key_data_ptk	176
eapol_sm_test_ptk	176
main	176

Figure 4.5. Code coverage

As shown in the picture reported above, we are able to obtain a high percentage of coverage on the targeted function. The numbers of lines that are executed is almost near to the total number of the lines of the target, while we reach a complete coverage for the number of functions. This actually indicates that the harness code is able to stress and trigger all the possible paths of the targeted function leading to a precise analysis. The hit count for each function of test-eapol.c is reported in table [4.4] below:

For what concern the eapol.c file, the coverage obtained is 83.2%. We report the hit count of the functions in table [4.4].

Function Name	Hit count
__afl_get_key_data_ptk	176
eapol_sm_test_ptk	176
main	176
test_handshake_state_free	163
test_handshake_state_new	163
test_nonce	63
verify_step2	111
verify_step4	62

Table 4.2. Hit count functions in test-eapol.c

From both tables it is possible to see that the number of hits for each function is pretty high. These results enforce the belief that the code is indeed explored in a suitable and accurate way, and the presence of software vulnerabilities is very low, if not completely absent.

4.4.1 Metrics

We can go a step further and more statistics related to the executions can be retrieved as well. One common way to do so is in a graphic view. Plotting the results based on three different metrics can give an further insights about the behaviour of the fuzz testing. The following plots are divided according to three different metrics:

- execution speed/second;
- path coverage;
- crashes and hangs found.

Then the statistics reported below are referred to the example execution in [4.5]:

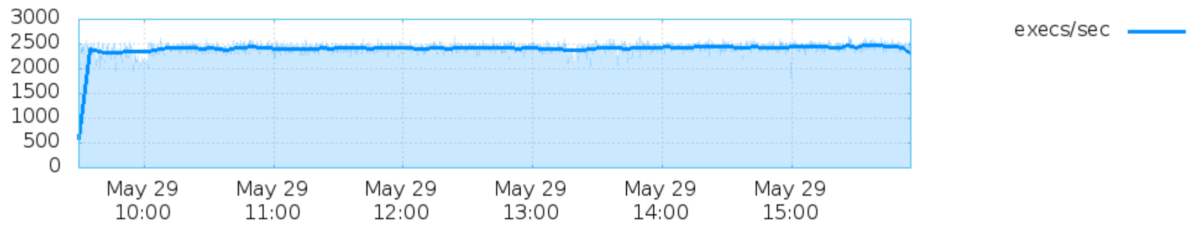


Figure 4.6. Execution speed plot

In figure [4.6] it is possible to evaluate the execution speed over the time frame reserved for the run. From this statistic we can see how the execution speed remains

Function Name	**Hit count**
eapol_create_ptk_4_of_4	3
eapol_find_rsne	3
eapol_handle_ptk_3_of_4 3	3
eapol_install_gtk	163
eapol_verify_ptk_3_of_4	3
eapol_decrypt_key_data	7
eapol_verify_mic	8
eapol_eap_msg_cb	79
eapol_create_ptk_2_of_4	91
eapol_handle_ptk_1_of_4	91
eapol_verify_ptk_1_of_4	91
eapol_calculate_mic	94
eapol_create_common	94
eapol_exit	163
eapol_frame_watch_add	163
eapol_frame_watch_free	163
eapol_frame_watch_remove	163
eapol_register	163
eapol_sm_destroy	163
eapol_sm_free	163
eapol_sm_new	163
eapol_sm_set_protocol_version	163
eapol_start	163
is_freed	163
__eapol_tx_packet	173
eapol_sm_write	173
eapol_init	176
eapol_key_handle	190
eapol_frame_watch_match_ifindex	309
eapol_rx_packet	309
__eapol_rx_packet	326
__eapol_set_tx_packet_func	326

Table 4.3. Hit count functions in eapol.c

constant for the entire time frame reaching a very high level, implying the efficiency of the evaluation.

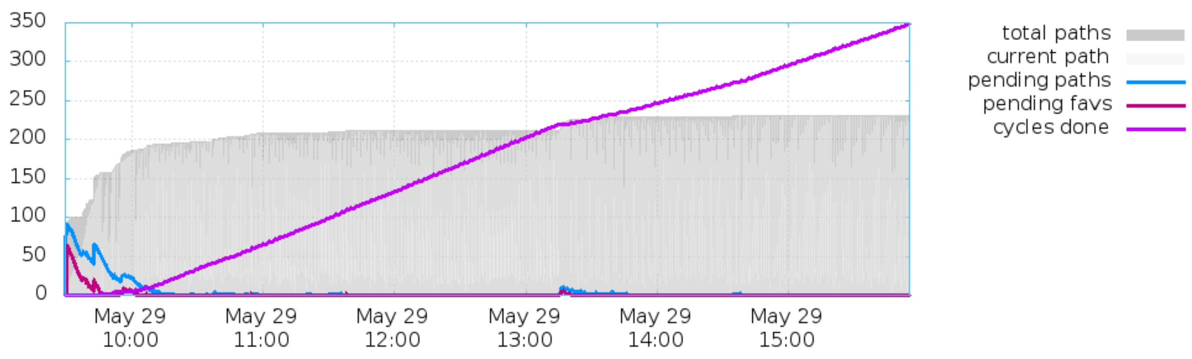


Figure 4.7. Path coverage plot

The figure [4.7] instead, shows that after an initial peek of both favourites and pending paths founds, they become much more stable. Nevertheless, the fuzzer continues to explore new paths even after almost 4 hours of execution. An another interesting value that we want to highlight is the number of cycles done. For the first four hours of execution the number of cycles grows along with the number of total paths found. After this period, the number of found paths become stable without any new discovery, while the number of the cycles continue to grows very fast.

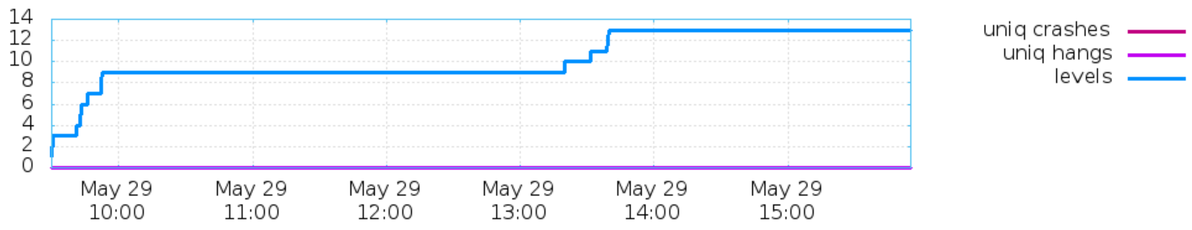


Figure 4.8. Crashes and hangs plot

Finally, the figure [4.8] reports the numbers of crashes and hangs found. As previously said, no crashes have been found, but still we can retrieve an interesting statistic value. The number of levels reached is 13, meaning that the fuzzer has generated plenty of different test cases starting from the one provided in the first place. This can be considered as a good index for the evaluation of our analysis.

4.5 Fuzzing for other unit tests

Following the same criteria that have been explained so far, we have tested two other functions: *eapol_sm_test_igtk()* and *eapol_sm_test_wpa2_ptk_gtk()*. The first function is very similar to the previously tested, although the packets are different and it also tests the installation of the IGTK, that is supposed to be installed as soon as the PTK is correctly installed. Minor changes to the test harness code have been made to allow the fuzz testing on this new function. For instance, the input provided to the fuzzer is different in terms of EAPOL-frames, hence this has required the creation of a new input file to be able to start from the correct input and generate valid test cases. Accordingly, as we expected, even this function does not trigger any software vulnerabilities, or at least not any easily detectable ones. The code coverage and the metrics gathered from the various analysis have showed, as for the first function analysed, n high rate of success in terms of path explorations. The results will be presented in the related Chapter 6.

The last function that has been tested, *eapol_sm_test_wpa2_ptk_gtk()*, is used to verify that both PTK and GTK are correctly installed by the two recipients. Generally speaking, the approach devised for the previous analysis remains unchanged, yet some new adjustments are required. As a matter of fact, this function needs 3 EAPOL-frames to test the messages exchange. Therefore, the test harness code must be modified accordingly. The entry point function should now accept three different EAPOL-frames and store them in the correspondent variable to be used by the tested function. The new input file required for this test can be generated using the utility code developed in the first place, allowing the write for a third frame. How to create a new input file to accommodate three different frames is discussed in more detail in the next chapter. For further information, refer to Chapter 5. The results obtained from this last analysis will be discussed in the related Chapter 6 as well.

Chapter 5

Fuzzing for KRACK

In this chapter it will be discussed how it is possible to find protocol-level vulnerabilities like KRACK through a fuzzing analysis. For instance, we will discuss how the harness code used for finding general software vulnerabilities can be modified and adapted to perform a slightly different analysis. It will be demonstrated how to simulate a key reinstallation attack on IWD library and how a fuzzer can discover it.

5.1 Preparing IWD for KRACK

Fuzzing for protocol-level vulnerabilities, like KRACK, presents some new challenges due to the fact that cryptography can affect performances negatively. As a matter of fact, being able to get crashes beyond the various authentication checks that are put in place in the code can present serious difficulties. Indeed, trying to do so would normally boil down to reversing a cryptographic function, which is meant to be *NP hard* problem.

To overcome this issue we should be able to find a way to revert most of the authentication checks performed allowing the fuzzer to easily go over them and render it able to look for a key reinstallation. The IWD code, developed with the intent of being unaffected by key reinstallation is indeed robust to KRACK. For this reason, to establish whether or not it is possible to detect this kind of flaw, in applying software fuzzing testing the general idea is to revert the patches implemented by the developers. Throughout several changes in the current implementation the fuzzer should be able to identify correctly whenever a key reinstallation is performed.

5.1.1 Input seed selection and file creation for KRACK

Dealing with a new problem, of course, requires a different approach with respect to file creation. On the other hand, the input seed selection is now easier. Already identified in the previous analysis, we are sure that the input provided is a suitable one. The file creation instead is different. In the previous analysis it was only necessary to fuzz on two EAPOL frames that, in this new scenario, is clearly not enough. Indeed, reading only two data frames is not correct anymore. Enabling the fuzzer to process three different frames requires us to store in the input file the same number of frames that the target program is expecting. Specifically, we need to read the second EAPOL frame twice, which is basically a duplicate of message 3. Using the utility code developed for the first analysis it is possible to add a minor change to create the new input file where three data frames are stored. Being d1, d2, d3 the data bytes of the payload respectively for the first frame, the second one and the third frame, we can write these data into a binary file as represented in figure [5.1]:

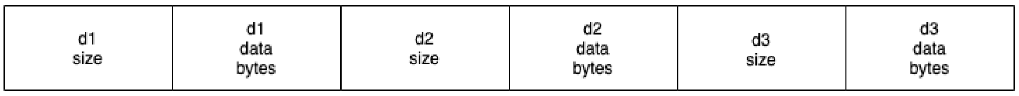


Figure 5.1. File format for KRACK

With this configuration, the input file will store the size of each frame followed by data frame itself, as explained in section 4.2.1,. In the picture above, d2 and d3 represent identical EAPOL frames. However they will be both mutated by AFL and used against the target program.

5.2 Harness Code for KRACK

The harness code previously developed cannot be used to find KRACK-like vulnerabilities due to the fact that we want to perform a different case analysis. First of all, in the previous implementation, we were only able to handle two data frames that, clearly, are not sufficient to simulate a key reinstallation attack. This means that two more variables for storing the third frame and its size are needed. The function showed in [4.5] can be taken as a starting point but it requires some changes. The third frame must be read as done for the other two, along with all the boundary checks needed to avoid malformed input, and eventually discard it in case of failure. Upon completion, the function will be able to return to its caller with the three data frames correctly stored in the corresponding variables and ready to used by the target function.

Detect key reinstallation function

The detection of a key reinstallation is the new problem that is introduced in this case. The test harness must be updated in such a way that this issue can be addressed. A new function must be developed in order to acknowledge whether the injection of second message 3 has lead to a key reinstallation. The code snippet below shows the implementation[5.1]:

```c
static void detect_key_reinstallation(struct handshake_state *hs,const uint8_t *tk, uint32_t
    cipher){
  static uint8_t prev_key[16] = {0};

  if (memcmp(tk, prev_key, 16) == 0) {
        printf("===> Key reinstallation detected!\n");
        assert(0);
    }

    memcpy(prev_key, tk, 16);
}
```

Listing 5.1. detect_key_reinstallation implementation

The function accepts as parameter a pointer to the current state of the handshake and a pointer to the key installed and the cipher suite used. Defined in the *prev_key* as an all-zeroed key, we compare the actual key in use with the defined one. If the comparison ends up with a positive result we have indeed detected a key reinstallation attack; otherwise we can simply copy back the key and the execution can go on without any problems. At this point in the tested function, a call to ita__handshake_set_install_tk_func is performed before the simulation of the messages exchange, providing as only parameter our *detect_key_reinstallation*. The test harness for the unit test is now completed. Nevertheless, we still have to deal with all the functions in charge of processing the messages. The next section will face this topic.

5.3 Revert authentication checks

The various authentication checks involved cannot easily be fuzzed. This implies that the *eapol.c* file has to be changed as well. The first function that we should handle is *eapol_verify_mic()*. In this function checksums, signature checks and message authentication checks are performed on the incoming packets. Hence, it is necessary to revert all of them. After our modification, as soon as the function is invoked by the supplicant, no checks are performed at all. This provides a return to the caller with a **true** value, ensuring that in each case the MIC calculation returns even though no computation is actually performed at all.

The next function to take care of is *eapol_handle_ptk_3_of_4()*, where the receipt of message 3 of the 4-Way Handshake is handled. In fact, on reception of message 3, the supplicant silently discards the message of the Key Replay Counter field value, once it has already been used, or if the ANonce value in message 3 differs from the ANonce value in message 1. Another check is made on the same function later on, where the **ptk_complete** value is set, meaning that the PTK has been correctly installed and it is checked whether a re-transmission of message 3 has occurred. In fact, if the the previously cited value is set, then it is certain that message 3 has been received again. We can be sure of that since the Anonce would not match otherwise and there would be no chances of reaching that point in the code. Both of the cases described above are shown below in [5.2]:

```
if (memcmp(sm->handshake->anonce, ek->key_nonce, sizeof(ek->key_nonce))){
    printf("Message discarded. Key replay counter already used or Anonce in msg3 differs
      from Anonce msg1\n");
    return;
    }
[...]
if (sm->handshake->ptk_complete)
```

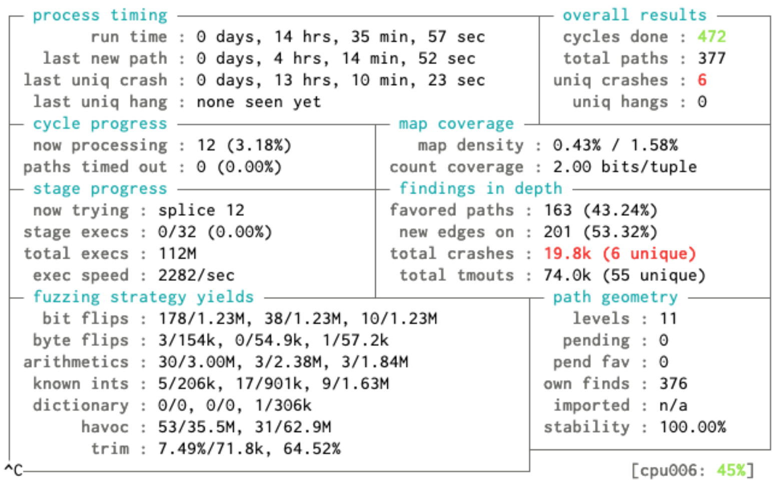
```
  return;
[...]
```

Listing 5.2. Reverting Anonce check

The last function that requires attention is *eapol_key_handle()*. Due to the fact that the supplicant and authenticator shall track the key replay counter for security association, the latter shall be initiated to zero on re-association. The authenticator will then increment the value of the replay counter on each successive EAPOL-key frame. The supplicant should also use the key replay counter and ignore EAPOL-Key frames with a key replay counter field value smaller, or equal to, any received in a valid message. Whenever the key replay counter is checked and is found to be valid the supplicant is allowed to update it. Otherwise, this should be avoided. Since we still want to fuzz the replay counter we have left this check unaltered.

5.4 Fuzzing for KRACK-like Vulnerabilities

Once all the required modification steps have been performed, the target program has been fed to fuzzer. In fact, a key reinstallation should now be triggered when AFL is fuzzing on our input file. The main thing that the fuzzer should do is to flip the correct bits/bytes in the second message 3. By flipping the correct portion of data it should find a valid replay counter and install the zeroed key. As done for the previous analysis, we have tested the target program extensively with a sufficient amount of tests in order to gather an accurate data set of results. With our test harness, AFL is indeed able to discover the key reinstallation after a few minutes of execution. The image below shows an example of one execution [5.2]:



```
┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 14 hrs, 35 min, 57 sec     │  cycles done : 472   │
│   last new path : 0 days, 4 hrs, 14 min, 52 sec      │  total paths : 377   │
│ last uniq crash : 0 days, 13 hrs, 10 min, 23 sec     │ uniq crashes : 6     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ───────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 12 (3.18%)       │    map density : 0.43% / 1.58%        │
│ paths timed out : 0 (0.00%)        │ count coverage : 2.00 bits/tuple      │
├─ stage progress ───────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 12            │ favored paths : 163 (43.24%)          │
│ stage execs : 0/32 (0.00%)         │  new edges on : 201 (53.32%)          │
│ total execs : 112M                 │ total crashes : 19.8k (6 unique)      │
│  exec speed : 2282/sec             │  total tmouts : 74.0k (55 unique)     │
├─ fuzzing strategy yields ──────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 178/1.23M, 38/1.23M, 10/1.23M        │    levels : 11         │
│  byte flips : 3/154k, 0/54.9k, 1/57.2k             │   pending : 0          │
│ arithmetics : 30/3.00M, 3/2.38M, 3/1.84M           │  pend fav : 0          │
│  known ints : 5/206k, 17/901k, 9/1.63M             │ own finds : 376        │
│  dictionary : 0/0, 0/0, 1/306k                     │  imported : n/a        │
│       havoc : 53/35.5M, 31/62.9M                   │ stability : 100.00%    │
│        trim : 7.49%/71.8k, 64.52%                  ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu006: 45%]
```

Figure 5.2. AFL fuzzing for KRACK

5.4.1 Crash Exploration

Applying a crash exploration, with the help of GDB and afl-utils, all the test cases that have been able to recreate a key reinstallation have been collected and analysed. Executing the test cases against the target program, after the PTK has been correctly installed, the injection of a second message three mutated by AFL can trigger our detection function. This means that the key has been reinstalled with all zeros. The evaluation of the results for this case will be given in the related Chapter 6.

Chapter 6

Results Evaluation

In this chapter we will provide a formal evaluation of the results obtained from our analysis. The results, that have been partially introduced in both Chapter 4 and Chapter 5 will be discussed in more detail.

For sake of completeness, we report the specification of the machine that has been used during our tests. All the experiments performed were run on virtual machine with the following features [6.1]:

OS	Linux Ubuntu
Version	16.04.6 LTS (Xenial Xerus)
Architecture	x86_64
ArchiteCPU op-mode(s)	64-bit
CPU(s)	16
Thread(s) per core	1
Model name	Intel Core Processor (Broadwell, IBRS)
CPU MHz	2399.994

Table 6.1. System specification

6.1 Results evaluation from Fuzzing for memory safety vulnerabilities

The results will be reported for each of the three functions that have been tested. It is important to highlight that after approximately 30 runs, executed separately for the three functions analysed, we report an estimation based on the average results. Due to the fact that no software vulnerabilities have been found by the fuzzer, we have assessed the quality of the execution based on the percentage of code coverage obtained, the number of hits for the functions involved and the number of paths found.

6.1.1 Unit test: eapol_sm_test_ptk

	Hit	*Total*	*Coverage*
Lines	136	142	95.8%
Functions	8	8	100%

Table 6.2. Coverage for test-eapol-ptk.c

	Hit	*Total*	*Coverage*
Lines	421	1132	37.2%
Functions	32	77	44.2%

Table 6.3. Coverage for eapol.c

Table [6.2] shows the code coverage report for test unit eapol_sm_test_ptk. On average, the fuzzer is able to cover the totality of the functions gaining a function coverage of 100%. While for the line coverage it can reach 95.8%. From these results we can tell that the fuzzer has successfully explored completely the unit test that has been fed to it. The value reported in [6.3] shows a lower coverage both in lines and functions.

6.1.2 Unit test: eapol_sm_test_gtk

Table [6.4] reports the code coverage for the function eapol_sm_test_gtk(), which can be considered a good result based on the values obtained. Similarly to the previous function, the fuzzer has managed to cover almost the totality of the functions ensuring a certain degree of performances. In table [6.5] the coverage with the respect to the eapol.c file is shown.

	Hit	*Total*	*Coverage*
Lines	135	141	95.7%
Functions	8	8	100%

Table 6.4. Coverage for test-eapol-ptk.c

	Hit	*Total*	*Coverage*
Lines	431	1132	38.2%
Functions	35	77	45.5%

Table 6.5. Coverage for eapol.c

6.1.3 Unit test: eapol_sm_test_wpa2_ptk_gtk

Table [6.6] shows the code coverage for the last function used as a target by the fuzzer. It reports good results, as for the previous two functions. Instead table [6.7] shows the results for the eapol.c file with respect to the eapol_sm_test_wpa2_ptk_gtk() target function.

	Hit	*Total*	*Coverage*
Lines	180	188	95.7%
Functions	9	9	100%

Table 6.6. Coverage fort est-eapol-ptk-gtk.c

	Hit	*Total*	*Coverage*
Lines	471	1132	41.6%
Functions	36	77	46.8%

Table 6.7. Coverage for eapol.c

Nevertheless, an important remark is required here. Evaluating the results obtained for the code coverage with respect to the *eapol.c* file, we can notice that the code coverage is much lower compared to the unit tests. Indeed these results have been gathered using all the functions that are implemented in eapol.c. The latter contains all the functions needed to perform the total number of unit tests originally stored in the test-eapol.c file. Due to the fact that the analysis has been restricted only to the unit tests where the 4-Way Handshake was involved we can recompute these values. As a matter of fact, if we leave out the unused functions, we obtain a coverage that can be approximate to 72.4% for the lines and 86.1 % for the functions.

The tables [6.8][6.9], report an estimation of the numbers of hits for the functions presented above:

Function Name	**Hit count**
__afl_get_key_data_ptk	236
__afl_get_key_data_gtk	228
__afl_get_key_wpa_ptk_gtk	332
eapol_sm_test_ptk	234
eapol_sm_test_gtk	228
eapol_sm_test_wpa2_ptk_gtk	332
main	332
test_handshake_state_free	253
test_handshake_state_new	221
test_nonce	96
verify_step2	128
verify_step4	67

Table 6.8. Hit count functions eapol_sm_test_ptk

Function Name	**Hit count**
eapol_create_ptk_4_of_4	3
eapol_find_rsne	3
eapol_handle_ptk_3_of_4	30
eapol_install_gtk	3
eapol_verify_ptk_3_of_4	30
eapol_decrypt_key_data	99
eapol_verify_mic	84
eapol_eap_msg_cb	129
eapol_create_ptk_2_of_4	91
eapol_handle_ptk_1_of_4	245
eapol_verify_ptk_1_of_4	91
eapol_calculate_mic	232
eapol_create_common	231
eapol_exit	163
eapol_frame_watch_add	309
eapol_frame_watch_free	308
eapol_frame_watch_remove	307
eapol_register	309
eapol_sm_destroy	309
eapol_sm_free	309
eapol_sm_new	309
eapol_sm_set_protocol_version	163
eapol_start	309
is_freed	309
__eapol_tx_packet	359
eapol_sm_write	359
eapol_init	332
eapol_key_handle	591
eapol_frame_watch_match_ifindex	789
eapol_rx_packet	789
__eapol_rx_packet	927
__eapol_set_tx_packet_func	927

Table 6.9. Hit count functions in eapol.c

From both tables, it is possible to see that the number of hits for all the functions involved is quite high. This proves the fact that the fuzzer has explored and performed its research in a correct and exhaustive way. From a graphical point of view, we have reported the statistical value of one execution that can be taken as an example of the average results that we have obtained in all the testing analysis

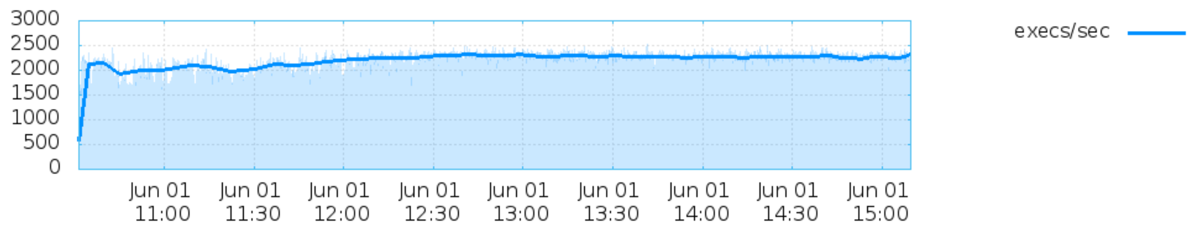


Figure 6.1. Execution speed plot

Picture [6.1] represents the average execution speed/seconds, which was around 2000 and 2300 for the most part.

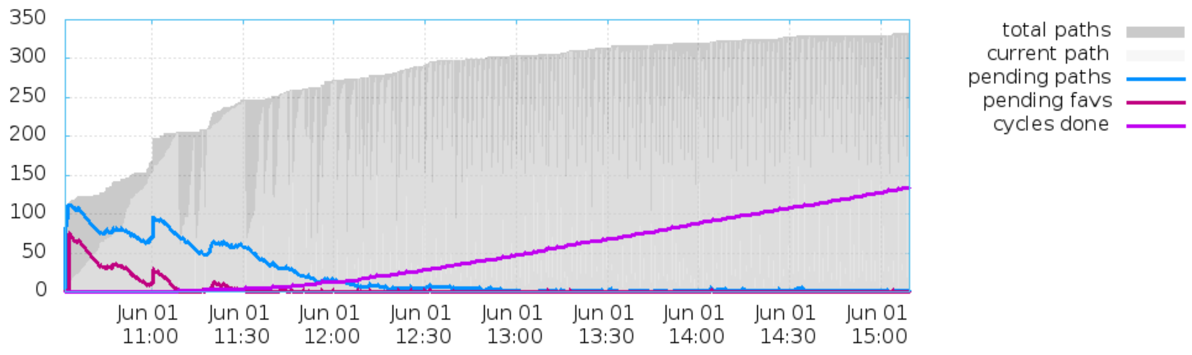


Figure 6.2. Path coverage plot

A more interesting statistic is plotted in figure [6.2], where it is possible to see that after an initial peak of paths found the discovery becomes more stable until the number of possible new paths runs out. At last, the number of levels reached by the fuzzer can be attested at around 12. From this value it is possible to assess that the generation of test cases produced by AFL, starting from the provided input, is indeed quite good [6.3].

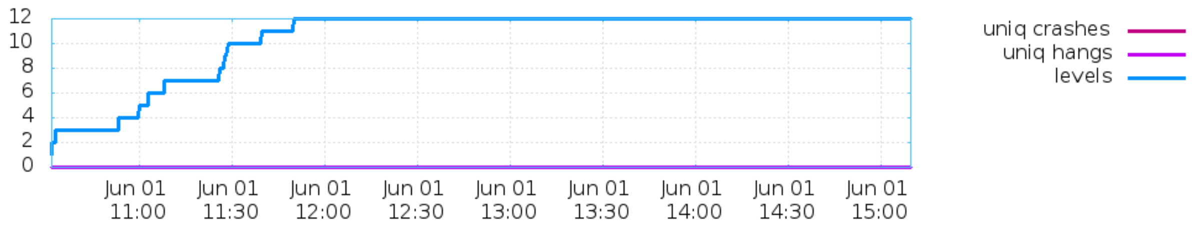


Figure 6.3. Crashes, hangs, levels plot

From our evaluation we can infer that the current implementations of IWD is not subject to memory safety vulnerabilities. In fact, we would not expect to find common vulnerabilities in a security critical code that is designed to be robust and safe. Nevertheless, during our analysis, as discussed in Section 4.3, the injection of malformed payload during the 4-Way Handshake could lead to a denial of service attack (DoS). In fact we have demonstrated, instrumenting the code, that whenever an EAPOL-frame cannot be correctly processed the handshake fails and the process is stopped. As a direct consequence, if an attacker manages to set up a MitM position they can start sending plenty of malformed packets preventing the completion of the protocol.

6.2 Results from Fuzzing for KRACK

After the evaluation of the findings gathered using AFL to find KRACK-like vulnerabilities, it possible to say that software fuzzing can be used as a valid technique in this context. In table [6.10] the coverage for the target function is reported, while in table [6.11] the values for the eapol.c file are reported.

	Hit	*Total*	*Coverage*
Lines	163	174	93.1%
Functions	9	9	100%

Table 6.10. Coverage for test-eapol-ptk-krk.c

	Hit	*Total*	*Coverage*
Lines	471	1093	43.6%
Functions	37	77	48.1%

Table 6.11. Coverage for eapol.c

The total code coverage, as well as for the previous tests, shows a success rate in analysis of the targeted program. Both lines and functions are almost totally covered, showing that AFL has explored all the possible paths. The number of hits for the function has been reported below in table [6.2]:

Function Name	**Hit count**
detect_key_reinstallation	83
__afl_get_key_data_ptk	337
eapol_sm_test_ptk	377
main	377
test_handshake_state_free	224
test_handshake_state_new	354
test_nonce	221
verify_step2	191
verify_step4	244

Table 6.12. Hit count functions eapol_sm_test_ptk_krk

The following statistics represent the plotted results for this analysis.

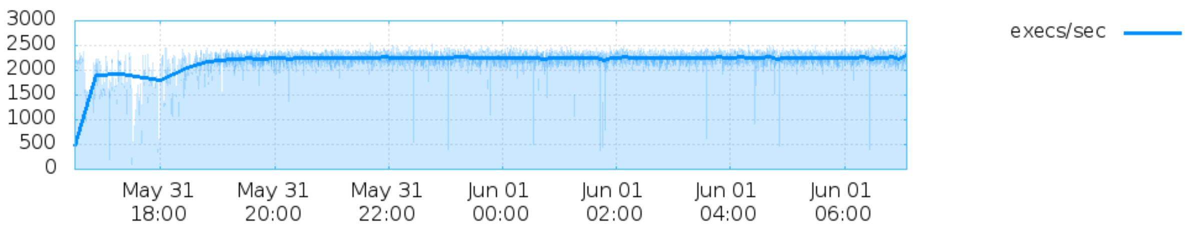


Figure 6.4. Execution speed plot

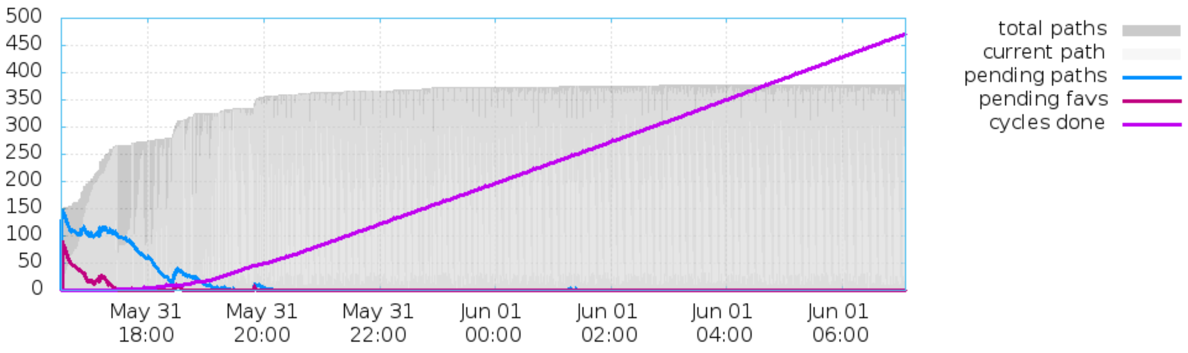


Figure 6.5. Path coverage plot

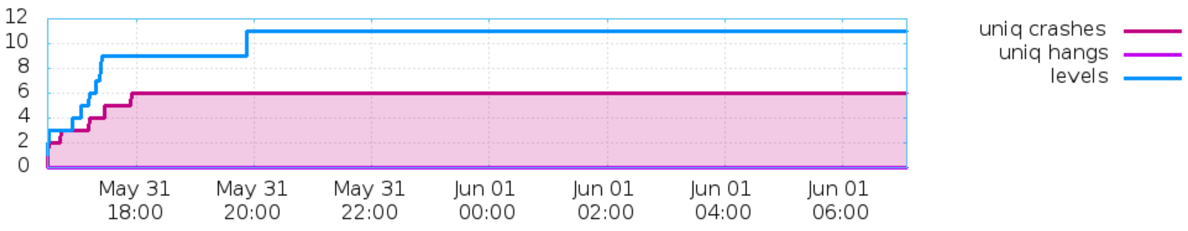


Figure 6.6. Crashes, hangs, levels plot

In conclusion, it can be inferred that it is possible to devise a methodology to applying software fuzzing in order to find software vulnerabilities in a protocol implementation. By turning unit tests into test harnesses for the fuzzer it is possible to discover memory safety vulnerabilities. Using a similar approach, although reverting authentication checks manually, it is also possible to spot KRACK-like vulnerabilities in a protocol-level implementation.

Chapter 7

Conclusions and Future works

Our first goal was essentially to define whether finding software vulnerabilities in a protocol implementation was possible using a systematic approach and in an automated way. Conclusively, we can say that it is possible to achieve this goal by going through several challenges that the problem inherently presents. Acquiring an adequate knowledge of the targeted program, understanding how data are treated and processed and how to alter it is a primary step in achieving the goal. The identification of suitable input candidates able to generate testcases that could reach deeper and harder to reach paths is indeed a variable that we have taken into account. Starting from a valid EAPOL-frame and exploiting AFL's features, the 4-Way Handshake protocol implementation provided by Intel's IWD has been stressed and analysed.

The creation and implementation of a test harness code, lying between the target program and the fuzzer itself, represents the entry point for our analysis. Throughout the test harness, a bunch of malformed and unexpected EAPOL-frames have been sent against the protocol in order to allow the fuzzer to analyse any possible paths and look for possible software vulnerabilities. As a consequence of the analysis carried out in this work, the results have shown that the target programs do not present any easily exploitable vulnerabilities since no crashes were detected during the testing phase. The validation of the results has indeed shown that the fuzzer has correctly analysed almost a total amount of lines of code, reporting a high coverage in each one of the tests executed. Although, one remark can be made; even if we did not find any software vulnerabilities in a high-critical security implementation, we have noticed that the injection of malformed packets can lead to a stalemate in the successful completion of the 4-Way Handshake. For instance, an attacker that manage to set up a MitM position may prevent the correct landing of messages between the supplicant and the authenticator. This can lead to a possible DoS (Denial of Service) attack.

The second main goal of this work was to determine the possibility of making use of the fuzzer to find protocol-level implementations, like KRACK. Along the lines of the work done on software vulnerabilities discovery, using the same approach but with some variations we have been able to simulate and detect a key reinstallation in the 4-Way Handshake protocol. In order to do so, some considerations have been made. For instance, the capability of the fuzzer to get through the various authentication checks would normally boil down to reversing a cryptographic function. This is usually considered an NP hard problem. Therefore, reverting manually the authentication checks made by the protocol has allowed us to fuzz the target program and detect a case of a key reinstallation attack within a few hours of execution.

On top of what has been discussed so far, we have implicitly answered the third research question proposed. As a matter of fact, with the implementation of a test harness, able to interact with the underlying protocol-level code, we can apply automated vulnerability detection to cryptographic protocol implementations.

Despite the results that we have obtained being quite satisfactory there is certainly room for improvements. Unit tests are surely a good starting point for constructing a fuzzing harness. It is indeed true that creating a suitable test harness specifically developed for a certain targeted unit test allows us to obtain a reasonable line coverage and path exploration. Yet, clearly there are some disadvantages. The effort required to go through each unit test, understanding the scope and how it can be turned into a test harness may be considered time consuming and often difficult. A possible future work could be to devise a new approach to combat these issues. For instance, the fuzzer should be aware of the protocol that it is going to have as a target, being able to automatically generate malformed payloads to detect potential flaws. An example could be to make the fuzzer able to decide in which order to send the packets, instead of automatically following the specifications of the protocol as it is done in the unit tests. By doing so the likelihood of finding interesting new behaviours would increase. In the same way, it could be possible to extend the analysis to other 4-Way Handshake and make the fuzzer able to perform its analysis on a larger scale, keeping a certain degree of accuracy and coverage.

Bibliography

[1] Wikipedia, https://en.wikipedia.org/wiki/Transport_Layer_Security

[2] Ubuntu manuals, http://manpages.ubuntu.com/manpages/cosmic/man8/wpa_supplicant.8.html

[3] The OpenSSL project, http://heartbleed.com/

[4] Wikipedia, https://en.wikipedia.org/wiki/Cloudbleed

[5] Wikipedia, https://en.wikipedia.org/wiki/Fuzzing

[6] Kaspersky Lab Encyclopedia, https://encyclopedia.kaspersky.com/knowledge/software-vulnerabilities/

[7] Ã. Erlingsson, Y. Younan, and F. Piessens, “Low-level software security by example”, Handbook of Information and Communication Security, pp. 633–658, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010

[8] L. Vaas, “The final ’final’ nail in wep’s coffin?(wired equivalent privacy)”, eWeek, 2007

[9] S. Fluhrer, I. Mantin, and A. Shamir, “Weaknesses in the key scheduling algorithm of rc4”, 2001, pp. 1–24

[10] A. Stubblefield, J. Ioannidis, and A. Rubin, “A key recovery attack on the 802.11b wired equivalent privacy protocol (wep)”, ACM Transactions on Information and System Security (TISSEC), vol. 7, no. 2, 2004, pp. 319–332

[11] M. Vanhoef, D. Schepers, and F. Piessens, “Discovering logical vulnerabilities in the wi-fi handshake using model-based testing”, 2017, pp. 360–371

[12] E. Tews and M. Beck, “Practical attacks against wep and wpa”, Proceedings of the second ACM conference on wireless network security, 2009, pp. 79–86

[13] K. Paterson, B. Poettering, and J. Schuldt, “Plaintext recovery attacks against wpa/tkip”, 2015, pp. 325–349

[14] “Iso/iec international standard - information technology telecommunications and information exchange between systems local and metropolitan area networks specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications amendment 6: Medium access control (mac) security enhancements (8802-11, second edition: 2005/amendment 6 2006: Ieee std 802.11i-2004)”, IEEE, 2004, ISBN: 0738140732

[15] M. Vanhoef and F. Piessens, “Key reinstallation attacks: Forcing nonce reuse in wpa2”, 2017, pp. 1313–1328

[16] M. Vanhoef and F. Piessens, “Advanced wi-fi attacks using commodity hardware”, Proceedings of the 30th Annual Computer Security Applications Conference, 2014, pp. 256–265

[17] Jouni Malinen, http://lists.infradead.org/pipermail/hostap/2017-October/037989.html

[18] Shirey RW (2000), https://tools.ietf.org/html/rfc2828

[19] B. Liu, L. Shi, Z. Cai, and M. Li, “Software vulnerability discovery techniques: A survey”, 2012 Fourth International Conference on Multimedia Information Networking and Security, 2012, pp. 152–156

[20] Bowne S (2015), https://samsclass.info/127/proj/p18-spike.htm

[21] S. Wang, J. Nam, and L. Tan, “Qtep: quality-aware test case prioritization”, Proceedings of the 2017 11th Joint Meeting on foundations of software engineering, 2017, pp. 523–534

[22] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey”, Cybersecurity, vol. 1, no. 1, 2018, pp. 1–13

[23] Michal Zalewski, http://lcamtuf.coredump.cx/afl/

[24] Denis Kenzior, https://iwd.wiki.kernel.org/

[25] Utilities for automated crash sample processing/analysis, https://github.com/rc0r/afl-utils

[26] The ’exploitable’ GDB plugin, https://github.com/jfoote/exploitable

[27] Michael Rash, https://github.com/mrash/afl-cov