POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Thesis

Security Analysis of the WPA2 KRACK patches

Supervisors
prof. Antonio Lioy
prof. Jan Tobias Muehlberg

Candidato
Graziano Marallo

Academic Year 2018-2019

Dedicated to my mother and father

Summary

Recently have been discovered that WPA2 is vulnerable to key reinstallation attacks (KRACKs). In response, software vendors patched their implementations to prevent key reinstallations. However, how can we be sure those patches are correct, and indeed prevent all key reinstallations? What if they are flawed, and it is still possible to attack implementations? The goal is address these questions and perform a security analysis of patches that are supposed to prevent key reinstallation attacks. Fuzzing technique will be applied in order to perform several analysis.

When connecting to a protected Wi-Fi network, a handshake is executed that provides both mutual authentication and session key negotiation. A recent discovery prove that this handshake is vulnerable to key reinstallation attacks. In response, vendors patched their implementations to prevent key reinstallations. 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, meaning some patches may be flawed in practice.

The goal of this thesis is to asses the correctness of patches. By doing that, different analysis will be done in order to find several bugs and possibile bug patterns in the 4-way Handshake implementation.

Acknowledgements

Opzionali, solo nel caso si sia ricevuto un aiuto speciale e particolarmente rilevante.

Contents

Chapter 1

History & Work

Questo capitolo e' stato inserito per tenere traccia delle modifiche apportate al documento durante il tempo e contenere i report quindicinali sul lavoro svolto.

Modifche rispetto alla versione precedente:

- Fix della bibliografia. (Bibtex)
- Fix di alcuni typos nel documento
- Inseriti dei placeholder nei capitoli ancora da scrivere
- Creato il capitolo introduttivo

1.1 Report

Lavoro svolto : Avendo terminato lo studio della letteratura (citata in bibliografia) che mi era stata fornita dal mio supervisor, ho concentrato la mia attenzione sulla parte pratica. Per effettuare una prima analisi del 4-Way Handshake sto utilizzando un'implementazione open source di un Wirless Daemon (IWD). Nello specifico i miei supervisor mi hanno suggerito di analizzare il codice sorgente contenuto nel file "iwd/unit/test-eapol.c". In quest'ultimo essenzialmente vengono effettuati una serie di test per verificare se ogni step del 4-Way Handshake viene completato con successo. Dopo un'iniziale analisi, ho individuato nel codice la chiamate alla funzione che testa lo scambio della PTK(Pairwise Temporary Key). Nella funzione ho individuato il punto in cui il primo pacchetto del messaggio 3 viene riempito con dati chiave dell' EAPOL frame, successivamente anche il secondo frame. Come punto di partenza ho pensato di sostituire ai dati che vengono passati normalmente dalla funzione originale, dati letti da un file. In questo modo facendo leggere i dati da un file e possibile sfruttare AFL e ottenere una mutazione dell'input sperando di incorrere in qualche bug da poter poi analizzare.

Per fare quanto detto è stato necessario creare un "harness code". Quest'ultimo sostanzialmente è una sorta di "imbracatura", o se vogliamo, un codice ausilario che permette al codice di essere sottoposto all'analisi del fuzzer. Le modifiche effettuate sono le seguenti:

1. modifica del main per leggere come primo argomento sulla command line un file in input
2. impedire la lettura del frame 1 del messagio 3 dalla struttura c predefinita;
3. fornire al fuzzer in input un file contenente dati chiave dell' EAPOL frame (usati originariamente dalla struttura);
4. effettuare controlli sul file per evitare di incorrere in errori indesiderati (i.e. file non trovato, lettura non riuscita);
5. ripetere gli step 3 e 4 per il frame 2 del messaggio 3;
6. rimpiazzare le chiamate alla funzioni originali con le funzioni ausilarie create;

Dopo aver effettuato qualche primo test, il fuzzer non riesce ad ottenere dei risultati validi anche dopo diverse ore. Per questo motivo ho pensato di semplificare parte dei test che vengono effettuati, in modo da evitare che vengano effettuati cicli inutili e modificare il codice in modo da poter ottenere risultati significativi.

Piano di lavoro :

- Semplificare unit test per ottenere performance migliori durante l'analisi.
- Selezionare uno dei test EAPoL/WPA2 4-Way Handshake all'interno del codice e renderlo il main target dell'analisi.
- Fare injection di bug all'interno del codice di libreria testata (branches and paths differenti). Per esempio pensavo di inserire qualche buffer overwrite e verificare se è possibile trovarli tramite il fuzzer.

Chapter 2

Introduction

In the last few years, we have witnessed a wide spread of the Wi-Fi networks. These are protected and secured by the WPA2. Hence, every router, smart-phone, laptop and smart device that is currently available on the market is using this kind of protocol to get connect to the internet and exchange data. Nowadays, those data are becoming increasingly crucial and important than in the past. It is sufficient to think to mobile banking, healthcare, automotive sector and so forth. All the aspects of our lives are driven and covered by those data that are exchanged on the network, so it's clear that this should be done in a secure way. This security has been provided for years by the 4-Way Handshake, a protocol thought and proved to be secure. Imagine if we find out that all our data are no longer exchanged in secure manner? What if a protocol thought be secure for over 14 years turns out be no longer secure? Are there any chance to understand where this protocol is flawed? In order to address these questions, it appears necessary to perform a deep analysis of the protocol itself along with the cause that brought us at this point. In this introductory chapter the 4-Way Handshake will be explored step by step, in order to better understand how the state-machine works, how messages are exchanged and where the flaw takes shape. The KRACK will be explained dropped in the context that is under analysis and the tool used to address this problem will be presented as well.

2.1 4-Way Handshake principles

As said previously, most of the Wi-Fi network that are used today are protected and secured by the WPA2, aka Wi-Fi Protected Access 2. Even the public hotspots that are wide spread around the world, are running under the WPA2 protocol even though, authentication is not performed for obvious reason but the encryption should be used instead. Such encrypted networks are so based on the Wi-Fi handshake that is able to securely negotiate a fresh pairwise key and exploit this key to ensure that the normal traffic exchange across the network is encrypted. An essential point of the work is for sure understand where the 4-Way Handshake is collocated and how it works during the establishment of secure connection between a generic client and a generic server.

2.1.1 Historical Background

At his creation, the 802.11 original standard supported a basic security protocol called WEP (Wired Equivalent Privacy), but unfortunately it was designed in a such way that a lot flaws were present and today is considered completely broken. For those reason the IEEE came up with two solution: 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 was that most part of the vendors implemented the cryptographic primitives of WEP in hardware and as a result an incompatibility issue arise and in fact older devices were unable to support CCMP. In order to solve this problem the short-term solution was designed and it was called (WPA-)TKIP protocol. Basically it's very similar to WEP but it is based on RC4 cipher and this means that the old devices can support TKIP using only firmware update. Once the protocol 802.11i was standardized, the WPA2 certification program started, requiring that a device had supporting CCMP and still able to retro-compatibility with the TKIP.

2.1.2 Functioning

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

Stage 1: ***Network Discovery*** Networks can be discovered essentially in two ways: passively listening for a beacons, or by actively sending a probe request. When the Access Point (AP) receives those 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 will be send. This will contain several useful element such as contains the supported pairwise cipher suites of the network, the group cipher suite being used, and the security capabilities of the AP. After the client has found the a valuable 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*** Here the supplicant will start authenticating itself with the authenticator and vice versa, and after this action has been completed the supplicant will continue by associating with the network. There are four authentication methods defined in 802.11i standard but here we will consider just one of them: the Open System authentication, that allows any supplicant to successfully authenticate. 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, pairwise and group chiper suite are chosen and so that have to been used by the participants.

Stage 3: ***802.1x Authentication*** This stage is optional and it can be left out in this work.

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 function on possible downgrade attack and negotiates a fresh session key called Pairwise Transient Key (PTK). The PTK is derived from the Authenticator Nonce, Supplicant Nonce, and the MAC address of both supplicant and authenticator. The 4-Way Handshake generally provides also a Group Temporal Key from the authenticator to the supplicant. Figure 2.1 shows how an Eapol frame is structured.

Message 1 : sent by the authenticator and containing randomly generated ANonce of the authenticator itself. Here the two info flags need to be set and they are Pairwise and Ack, represented by label Msg1. It's important to notice that at this level there is no protection by a MIC (Message Integrity Check), hence for an attacker could be possible 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 supplicant, and is protected by a MIC. In this message the key info flags to set are Pairwise and MIC, represented by label Msg2. Here it's important to say that in the Key Data Field of the EAPOL message is stored the RSNE element, 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 contains again the ANonce and secure by the MIC as well. The required key info flag here are Pairwise, MIC and Secure (labelled in Msg3), while the Key Data field will contain the supported cipher for the AP. When the supplicant receives this message, ad done previously by the authenticator, the RSNE is checked with the one received when the protocol takes place for the first time, and if the the two RSNE differ then a downgrade attack has been detected and the handshake aborted.

Message 4 : this is the last message and is sent by the supplicant in order to inform the authenticator that the handshake has been completed in correct 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.

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 used to protect from broadcast and multicast traffic. The procedure is also used to periodically renew the group key. The figure 2.2 shows the steps that regulate the 4-Way Handshake during the connection establishment.[1]

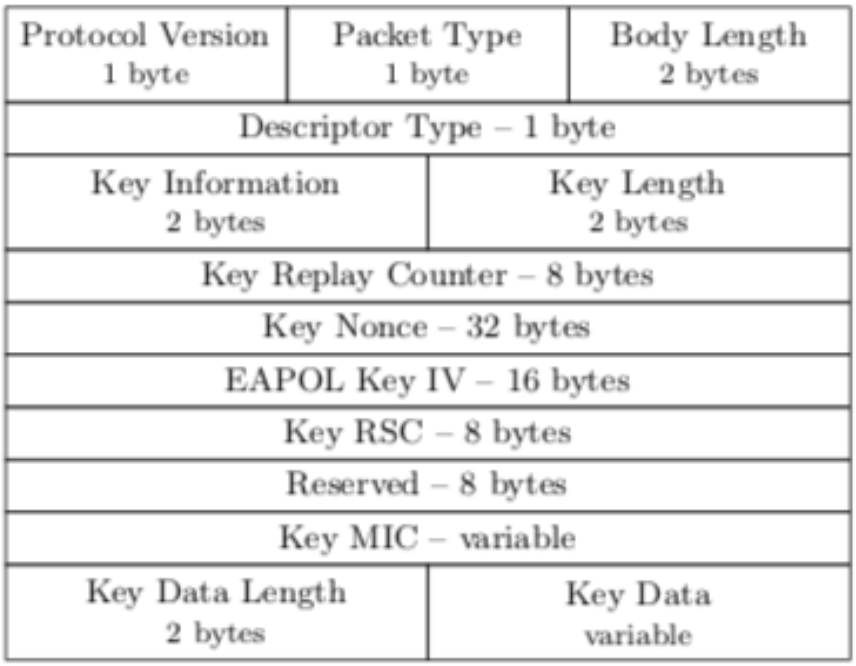

Figure 2.1. Layout of Eapol-Key frame

2.2 The KRACK attack

After a brief overview on how the 4-Way Handshake protocol works, now it's important to understand in which way is possible to attack and exploit its vulnerabilities. The Key Reinstallation Attack (KRACK) abuses of a 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 time and in addition to that it was proven to be secure. But in this kind of attack, the adversary can trick a victim in reinstalling an already in-use key without be conscious of it.

Nowadays all the Wi-Fi networks that are used, have a the WPA2 protection as defence mechanism against possible attack. This kind of technology relies obviously on the 4-Way Handshake defined in the standard 802.11i. As said 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, and since its introduction in the 2003 no breaches in the protocol have been found, apart from the weaknesses encountered by using TKPI (that was uniquely intended a short term solution). The idea behind the attack actually is more trivial that one can imagine, and can be sketched in the following way. Let's consider a client joins a network, 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's important to notice the fact that messages can be lost or even dropped, so the AP, in this case, is in charge to retransmit the message if do not receive a proper response as acknowledgement from

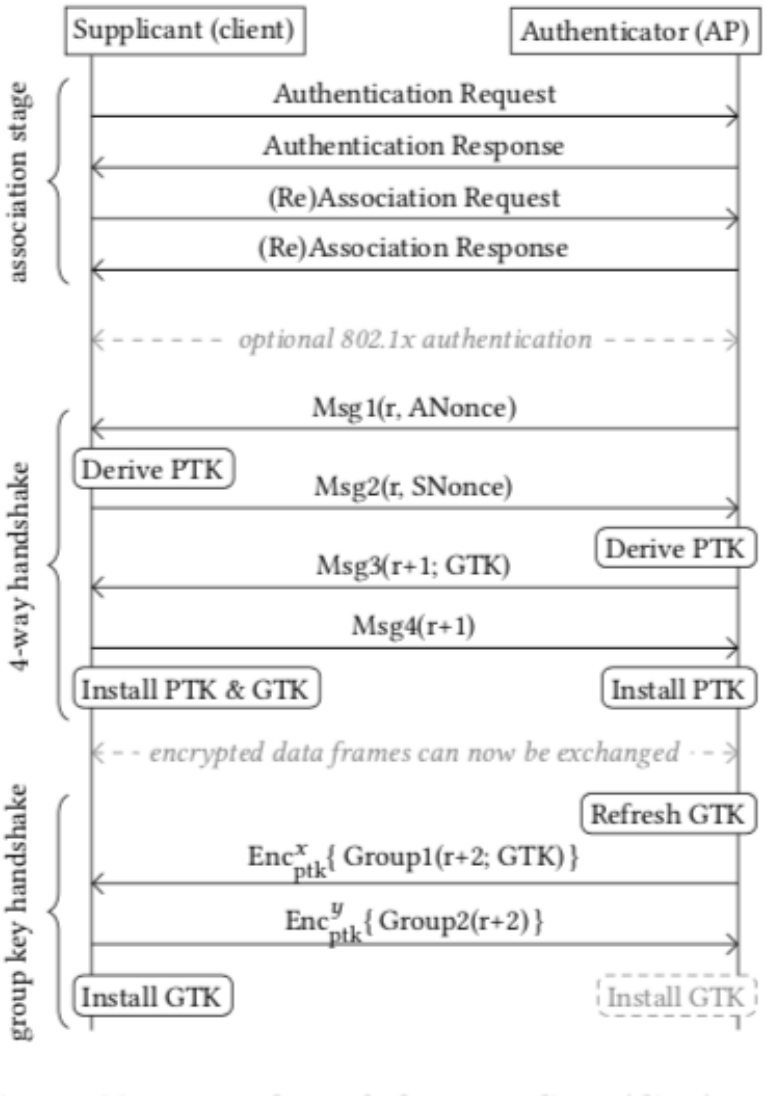

Figure 2.2. Messages exchanged when a supplicant connects to an authenticator.

the supplicant. Since this mechanism is easily understandable that message 3 can be received multiple times, and each time it will reinstall the same session key, but resetting the nonce and the receive replay counter used by the data-confidentiality protocol. Now what an attacker can do is forcing these nonce resets by collecting and replaying transmission of message 3, and 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 above described turned out to be devastating against version 2.4 and higher of "wap_supplicant", that is one the Wi-Fi most commonly used in Linux system. Since Android uses a modified version of the wpa_supplicant, all the Android version from the 6.0 to the newer one are effected.

2.2.1 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 spot the weaknesses of the protocol itself. The 802.11i amendment does not contain a formal state machine that explain how the supplicant has to be implemented, it just provide a pseudo-code that describe how, but not when, a certain message should be processed and handled. In one of it's extension, precisely the 802.11r, the 4-Way Handshake protocol come along with a detailed state machine of the supplicant (show in figure 2.3). When a connection to a network took place 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 occurs in two situation: when connecting to a network for the very first time or when the session key is being refreshed after an already terminated 4-Way Handshake. As soon as the supplicant enter in the PTK-START stage generates a random SNonce, compute the Temporary PTK and then sends its nonce to the authenticator by using message 2. At this point the authenticator will reply with message 3, which will be accepted by the supplicant only if the MIC and the reply counter are valid. If its the case, the supplicant passes 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 it's opened and the supplicant is able to receive and send normal data frames. Here it's 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.

2.2.2 Key Reinstallation Attack

After taking into account the functioning of the state machine, it's now possible dive into the attack itself. Since the supplicant accepts retransmissions of message 3, even when it is in the PTK-DONE stage, it's possible to force the reinstallation of th PTK. Specifically, the attacker first establish a man-in-the-middle (MitM) position between the supplicant and the authenticator, and then by exploiting his position can trigger the retransmissions of message 3 by preventing message 4 from correctly arriving to the authenticator itself. As a consequences, it will retransmit message 3 and the supplicant will forced to reinstall an already-in-use PTK. In turn, this will reset the transmit nonce and the receive replay counter that is currently used by data-confidentiality protocol. Notice here that depending on which protocol is used, the attacker is allowed to replay, decrypt and/or forge packets freely.

2.2.3 Plaintext Retransmission of message 3

In the scenario presented above, if we consider the victim still accepting the plaintext retransmissions of message 3 after it has already installed the session key, it's easy to understand how the key reinstallation attack is straightforward. In the initial step,

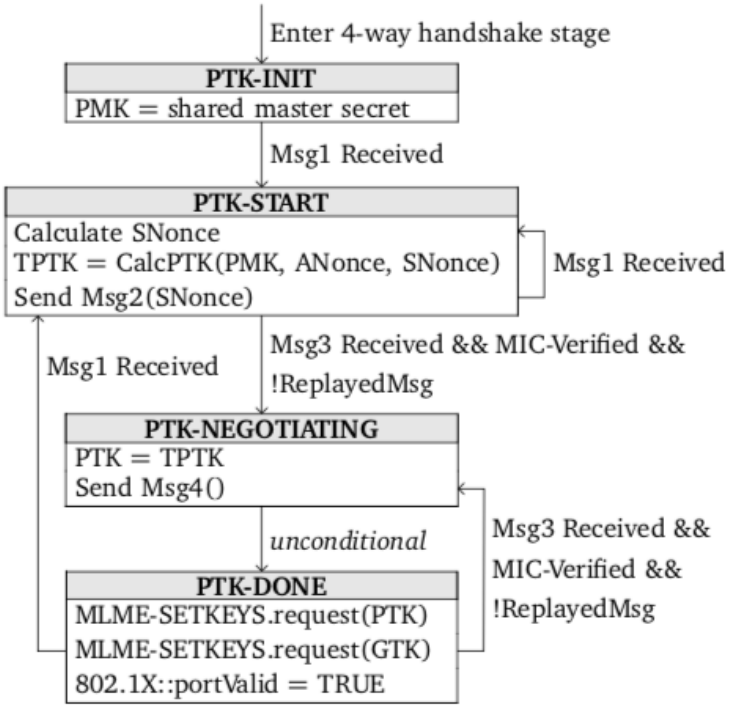

Figure 2.3. Supplicant State Machine

the attacker can use a channel-based MitM attack in which way he can manipulate as he desire the handshake messages. Soon after this operation is been set, he can prevent message 4 from arriving at the authenticator, showed in figure 2.4. As soon as message 4 has been sent the victim will install both PTK and GTK, and 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 didn't receive message 4 since it has been intercepted by the attacker. At this point the latter forwards the retransmitted message 3 to the victim 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's 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. Let's omit temporary the stage 4 and skip forward. In the end, when the victim retransmit its next data frame, the data-confidentiality protocol will reuse nonces. The attacker have the faculty of manages 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 2.4, 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 would happen when the client that accepts plaintext retransmissions of message 3 may already be reusing nonces without no one forcing it to do so.

Go back to the stage 4, that is the one in charge of completing the handshake at the authenticator side. Since the victim has already installed the PTK, the message is now encrypted making this step no so trivial; plus the the authenticator will reject this message for due to the fact that its PTK is not installed yet. In principle this should happen but 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 showed in figure 2.4. As result of this action, the PTK will be installed and the AP will start sending encrypted unicast data frames to the client. As said before, tha attacked has been confirmed works against *wpa_supplicant* that is used by Linux and Android.[2]

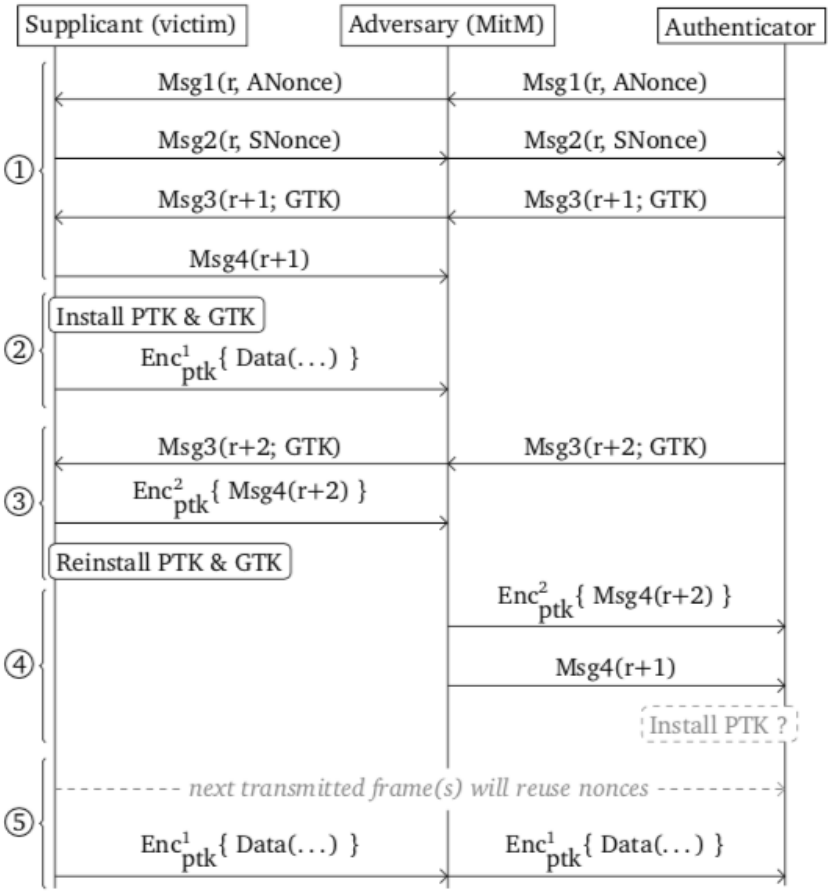

Figure 2.4. Key Reinstallation attack when the victim still accept message 3

Vulnerabilities, in the last few years, have become one of the main cause of threats in cyberspace security. A vulnerability is basically, as defined in RFC 2828, a flaw or weakness in a systems design, implementation, or operation and management that could be exploited to violate the systems security policy. Since they are so important and meaningful, a lot of work has been done in order to fight and in a certain way stem the attacks.

Various technique can be used like static analysis, dynamic, symbolic execution and fuzzing. Our main interest is on the latter. Fuzzing, basically, requires less knowledge of the target, and for this reason can be easily adapted and scaled to a large variety of situation and problem. Due to this reason is the de facto the most popular vulnerability discovery solution.

2.3 Fuzzing

2.3.1 General background

Attack on vulnerabilities, especially the ones made for example on zero day vulnerability, are seriously dangerous and sometime can brought to critical situation as well. So much of the effort of the research carry out nowadays is done on vulnerability discovery techniques towards software and information system. One of the main technique on which the attention has been focused on without any doubt, is fuzzing. But what is fuzzing and how it works?

The concept of fuzzing was initially proposed in the 1990's, and even though the concept has remained unchanged, the way of how fuzzing can be done has grown exponentially and evolved as well in the last few years. On the other hand, through the years has been noticed that fuzzing tends to to find simple memory corruption bugs and cover a small part of the the target code provided, and so as a result a low efficiency in findings major or critical flaws. Using a combination of feedback-driven fuzzing mode and genetic algorithms can provide a more flexible and customizable fuzzing framework that makes the fuzzing process more clever and in a certain way more efficient as well.

2.3.2 Fuzzing process

Fuzzing is basically the most popular vulnerability discovery technique used today. It stars with generating massive normal and abnormal inputs towards application and at the same time try to detect exception by feeding generated inputs to the target applications and carefully monitoring each execution states. In comparison with other techniques, fuzzing present itself as easier in deploying and plus it allows to perform analysis even without the source code. Another remarkable feature is that fuzzing tests is 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 a scalability gain. Nevertheless, on the other hand, presents any disadvantages such as low efficiency and low code coverage, but this not prevent it to be become the most efficient state-of-the-art vulnerability discovery technique. Is important to understand how the process work and how it is defined.

The main process of the traditional fuzzing process has four main stages, that can be summarized in this way(Figure 2.5):

- Test case generation stage

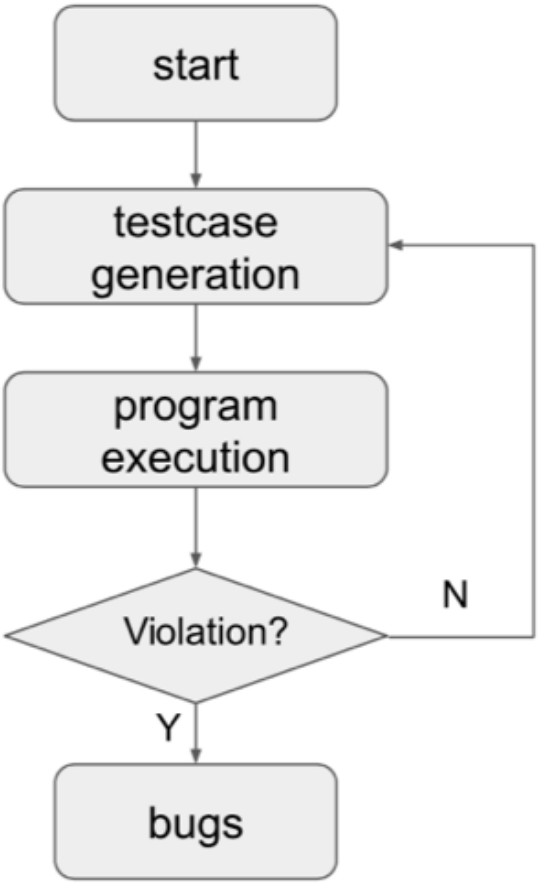

Figure 2.5. Fuzzing Stages

- Test case running stage
- Program execution stage monitoring
- Analysis of exceptions

After a bunch of program inputs have been generated (called from here on test case), these so generated inputs have to match the requirements of tested program as much as possible. Inputs can be of various type, e.g. different file formats, network communication data, executable binaries and so forth. In state-of-art-fuzzers are used essentially two kind of generators and they are the generation based and mutation based.

Test cases are given to target programs after they have been generated in the previous phase, than fuzzers automatically start and finish the target program process and drive the test cases handling process of target programs. Before the execution itself, analysts are able to configure the way the target programs start and finish, and also predefine which parameters and environment variables have to be used. In a normal execution, the fuzzing process should stops at a predefined time-out, program execution hangs or crashes. While running, fuzzer monitor the execution state during the execution of the program, seeking for exception and/or crashes. When some violation is captured by the fuzzers store it in the

correspondent test case for latter replay and analysis. In the subsequent analysis stage, the analyst could try to determine the location and also the root cause of the violations collected. This job can be exploited by means of debuggers like GDB or similar.

- ***Generation Based***: this kind of fuzzers the knowledge of program input is strictly required, along with a configuration file that basically offer the guide lines to generates test cases.
- ***Mutation Based***: requires a set of valid initial inputs and test cases are generated throughout the mutation of initial input and test cases generated during the first fuzzing process.

2.3.3 Types of Fuzzers

Another important aspect to consider is surely the type of fuzzers that exist. As said above 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 box. 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 source code and also how test cases have affected the program running state. Black box perform fuzzing test without any knowledge of source code while gray one also does not have source code awareness but the information are collected essentially from program analysis. Another classification is direct fuzzing and coverage-based fuzzing. The goal of a direct fuzzer is to generate test cases that can cover the target code and the target paths of a program and expect a faster test on programs; while a coverage-based fuzzers try to generate as much code of programs as possible and so expect a more thorough test along with a bunch of bugs discovery. Still, both of them have as a problem, or better as a key challenge, how to extract information of executed paths. The last classification is dumb or smart fuzzer according to weather there is a feedback between the monitoring of a program execution state and test case generation.

2.3.4 Key Challenges in Fuzzing

There are several challenges that today's fuzzer need 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 the 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 a more than fairly importance.

The way how fuzzers mutate has an impacting importance, and so, for what concern blind mutation, it could end up in a serious waste of testing resource while a more conscious mutation strategy could increase and improve the efficiency of testing significantly. In conclusion, by answering this questions is possible to cover more programs paths and is easier to trigger bugs. For what concern the second challenge, the work done till now has proved that better coverage results in a higher probability of finding bugs. Despite of that, most test cases only cover the same few paths, while most of the code could not be reached and fairly analysed. So from this we can understand that it's not a so sensible choice to achieve high coverage only exploiting a large amount of test cases generation and use 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. Actually, this kind of technique is used by AFL[1] and will be discussed further on. The last challenge is about validation. Programs, generally, validate inputs before parsing and handling, and the validation works as a sort of guard of programs used to protect program against invalid input and damage that can be caused by malicious constructed inputs. For example invalid test cases are always discarded or simply ignored.

2.3.5 Coverage-Based Fuzzing - AFL

This kind of strategy is the mostly spread and used by state-of-the-art fuzzers and has been proved to be one the more effective and efficient. The fuzzers try to travers as many as running state of the program as possible. Using this such of scheme is still possible to have a loss of information and certain degree of uncertainty. AFL is one of the kind of fuzzers that exploit this scheme of working. In the program analysis, the program itself is composed by basic block, 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. That is because of the fact that basic block is the smallest coherent units in a program execution, measuring function would end up in an information loss or redundancy problem, and 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 is happen 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, even edges. This lead to a better coverage that could not be achieved with a simple basic block recording. AFL is the first to introduce the edge measurement method into coverage-based fuzzing. Since AFL has been used to perform security analysis in this thesis work, here we show how coverage-based fuzzers gain coverage information during the fuzzing process. AFL gains the coverage information via lightweight program instrumentation. Depending on the source code, AFL provides

[1] American Fuzzy Lop

two instrumentation mode, the compile-in instrumentation and the external one. In the first one, AFL provides both gcc and llvm mode, which will instrument code snippet when binary code is generated. While in the external mode, basically the qemu mode provided by AFL, it will instrument code snippet when basic block is translated to TGC block. An example of how the instrumentation of the code is applied is shown in figure 2.6.

Algorithm 1 Coverage-based Fuzzing

Input: Seed Inputs S

```
1:  T = S
2:  Tx = ∅
3:  if T = ∅ then
4:      T.add(emptyfile)
5:  end if
6:  repeat
7:      t = choose_next(T)
8:      s = assign_energy(t)
9:      for i from 1 to s do
10:         t' = mutate(t)
11:         if t' crashes then
12:             Tx.add(t')
13:         else if isInteresting(t') then
14:             T.add(t')
15:         end if
16:     end for
17: until timeout or abort-signal
```

Output: Crashing Inputs Tx

Figure 2.6. Fuzzing Algorithm

2.3.6 Working process

The algorithm showed in figure 2.7, shows the general working process of a coverage-based fuzzer. The test starts it's execution from an initial given seed inputs, nevertheless if no input is provided then the fuzzer is able constructs one itself. In the main fuzzing loop, the fuzzer repeatedly chooses an interesting seed for the following mutation and test case generation, then the target program is driven to execute the generated test cases under the monitoring of fuzzer. The resultant test cases that trigger crashes will be collected, and other interesting ones will be added to the seed pool in order to be analysed in a second moment. For a coverage-based fuzzing, all test cases that are able to reach a new control flow edges are considered to be interesting and yet to be used. The main fuzzing loop stops at a pre-configured time-out or an abort signal given by the user itself. During its execution, fuzzers tend to track the execution in different ways. The execution need to be tracked basically for code coverage and security violations. The first one is used in order to

pursue a meticulous program state execution, while the second one is used in order to have a better way to trigger and find bugs.

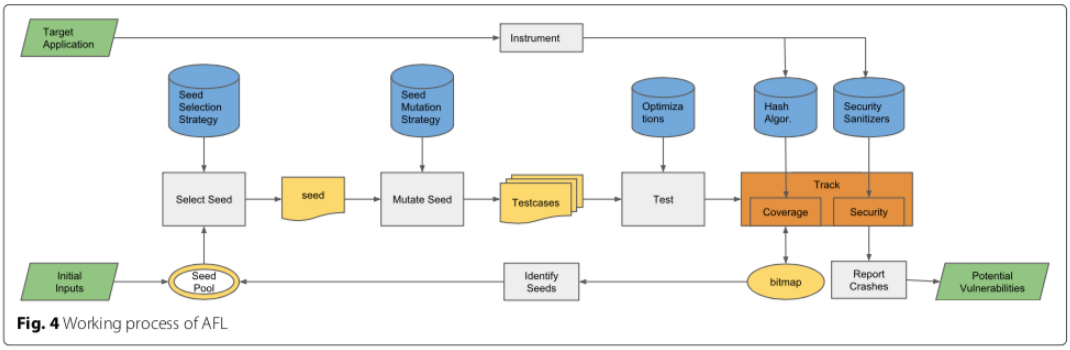

Figure 2.7. Fuzzing Working Process

It's also useful to show how an AFL process works. The target application is instrumented before execution for the coverage collection, and this instrumentation can be done 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, and usually the smallest and fastest are preferred and so picked up. After that seed files are mutated according to the mutation strategy, and a bunch of test cases are generated. As last step 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), in order to add them to seed pool and exploits them for the next round of execution.

2.3.7 Fuzzing of protocols

Fuzzing technique has been used to detect several vulnerabilities on different application since it was invented. According to the different characteristics of the target application, it's possible to use different fuzzers tha can apply a certain strategy to face different problem. Since in this work fuzzing is intended to be used against a protocol, it's useful to mark how can be done in this context. In the recent past, lots of local application have been transformed into network service in a B/S mode. In this configuration, services are deployed on network and client applications can communicate with those exploiting network protocols. For this reason, security testing on network protocols become more and more crucial and source of concern. Security flaws in protocols could be much worse than a local applications flaws, in fact they can result in serious damages, like DoS attack, information leakage, defacement and so forth. Several challenges are bond to fuzzing protocols, first one is that services may define their own communication protocols, which are difficult to determine protocol standards. Plus, even for documented one is still hard to follow specification such a RFC document. By the way, exploiting AFL, we're going to try exploit the flaws found in the 4-Way Handshake protocol. [3]

2.4 American Fuzzy Lop

As mainly discussed above, fuzzing is one of the most powerful and proven strategies for identifying security issues in real-world software; it is responsible for the vast majority of remote code execution and privilege escalation bugs found to date in security-critical software.

2.4.1 Approach to the problem

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

1. Load user-supplied initial test cases into the queue,
2. Take next input file from the queue,
3. Attempt to trim the test case to the smallest size that doesn't alter the measured behaviour of the program,
4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,
5. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.
6. Go to 2.

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

2.4.2 Design goal of AFL-Fuzz

Speed : afl-fuzz is meant to let fuzzing most of the given targets at approximately their native speed. On top of this, the tool leverages instrumentation to reduce the amount of work essentially in two ways, i.e. skipping non-functional but non-trimmable regions of the input file or by systematically trimming the corpus

Rock-solid reliability : most of the approach used today based on symbolic execution or similar are unreliable with real world targets, whilst afl-fuzz is designed to be rock solid. In fact it is designed to be just a very good traditional fuzzer with a wide range of interesting and useful strategies to go by.

Simplicity : AFL is designed to allows a better understanding of the testing framework even though there is not a complete knowledge of the tool itself. Essentially the three knobs you can play with are the output file, the memory limit, and the ability to override the default time-out. All the other settings are supposed to be work fine already. When this not apply, warning message are showed to the tester outlining the possible causes and workarounds.

Chainability : in order to avoid the creation of custom in-process fuzzers or exploit a massive CPU power again targets that are resource-hungry, AFL tries to elude ingeniously this problem by allowing testers to use more lightweight targets in a way to create small corpora of test cases that can be fed into a manual testing or a UI harness.

Chapter 3

Critical Analysis

This chapter/section has to be write yet.

Questo capitolo fornisce, se necessario, un'analisi critica dei lavori precedenti sul tema trattato nella tesi

Chapter 4

Design

This chapter/section has to be write yet.

Discutere in questo capitolo come ? stata progettata la soluzione al problema trattato nella tesi, indicando anche se sono stati valutati vari possibili approcci o soluzioni pre-esistenti e giustificando le proprie scelte. Descrivere quindi la soluzione vera e propria.

Nel caso sia stato sviluppato del software non triviale, è buona norma dedicargli tre sezioni:

- architettura dell'applicazione (interazioni con gli utenti e con altri sistemi, moduli logici, flussi dati interni ed esterni);
- manuale dello sviluppatore (descrizione dei moduli, degli algoritmi, delle interfacce e delle strutture dati);
- manuale utente (come installare ed usare il programma, interfacce, comandi, dati in input ed in output).

Nel caso di software molto voluminoso, queste tre sezioni possono diventare tre capitoli separati.

Chapter 5

Results

This chapter/section has to be write yet.

Inserire in questo capitolo i risultati conseguiti, cercando di analizzarli – se possibile – in modo quantitativo.

Chapter 6

Final Conclusions

This chapter/section has to be write yet.

Qui si inseriscono brevi conclusioni sul lavoro svolto, senza ripetere inutilmente il sommario. Si possono evidenziare i punti di forza e quelli di debolezza, i possibili sviluppi futuri o attivit? da svolgere per miglioreare i risultati.

Bibliography

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

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

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