UNIVERSITÀ DEGLI STUDI DI MILANO

DIPARTIMENTO DI INFORMATICA

Corso di Laurea in
Sicurezza dei Sistemi e delle Reti Informatiche

Security Fuzzing of ADS-B, NextGen protocols and avionics devices

RELATORE
Prof. Valerio Bellandi

TESI DI LAUREA DI
Giulio Ginesi
Matr. 872795

Anno Accademico 2017/2018

Ai miei genitori

Ringraziamenti

Contents

Chapter 1

Introduction

T0D0 1: Short introduction

The rest of the thesis is organized as follows:

T0D0 2:

- *Chapter 2: Avionics Protocols*
- *Chapter 3: Software Testing and Fuzzing*
- *Chapter 4: Experimental Setup*
- *Chapter 5: Results Analysis And Conclusions*
- *Chapter 6: Future Work*

Chapter 2

Avionics Protocols

Avionics telecomunications have undergone a major upgrade in the last years which is still in progress as new standards are set to be fully deployed within the next decade. Such changes are made to bring the benefits of modern tecnology onboard of the aircrafts and in the **Air Traffic Management** (**ATM**) world.

Different organizations collaborate in defining the standards and guidance documents for aeronautics, the major are the Radio Technical Commission for Aeronautics (**RTCA**) in the United States, the European Organisation for Civil Aviation Equipment (**EUROCAE**) in Europe and the International Civil Aviation Organization (**ICAO**) which is United Nations organization. All this organizations are non governative therefore they produce standards, guidelines or rules that will then have to be adopted by the Federal Aviation Administration (**FAA**) in the united states, by the **EUROCONTROL**[1] and the single national aviation autority of the various european countries (*ENAC/ENAV*[2] in Italy). This fragmentation of the organizations, the absence of a global shared guidance on the standardization of the aviation sector (which is partially an **ICAO** task.[3]) and the developement of different standards to acomplish the same task brought to an unclear situation regarding standards and regulations, for this reasons some problems in the the definition of a globally accepted NextGen family of protocols rose in the latest years; this will be discussed in the appropriate section.

T0D0 3: something more

There are two way of identifying an aircraft:

- The classic radar (or Primary Radar), based on electromagnetic waves which is only able to give information on a position of an object in the sky. Such system gives a real-time image of the portion of sky that it is observing, this includes aircrafts, birds, clouds, and any other object in his field of

[1] Organization to provide unique centralized platform for civil and military aviation coordination in Europe.

[2] Ente Nazionale Aviazione Civile e Ente Nazionale Assistenza Volo

[3] Art 1. The contracting States recognize that every State has complete and exclusive sovereignty over the airspace above its territory.[1]

view making it a basic and farely unreliable system. Since this method requires no interaction with the aircraft that is beeing traced it is called **non-cooperative** system.

- Second Surveillance Radar (SSR) is an evolution of the above system which, in addition to the spatial position of the aricraft, requests additional information depending on his mode of operation. Since this system requires an active cooperation from the aircraft which must reply to the information requests send by the system it is called **cooperative** system.

Cooperative systems allow to have a bidirectional aircraft-ground as well as aircraft-aircraft data flow. This system stems from the military Identification Friend or Foe (**IFF**) one which was developed during the second World War. The actual civil system uses a 4 digit code called "Transponder Code" or **"Squawk"** to identify the aircraft, this communicates via a **transponder** which handles the incoming interrogations and the responses. In Figure 2.1 is an overview of all the protocols and their division between the old generation and the new generation which is still in deployiment.

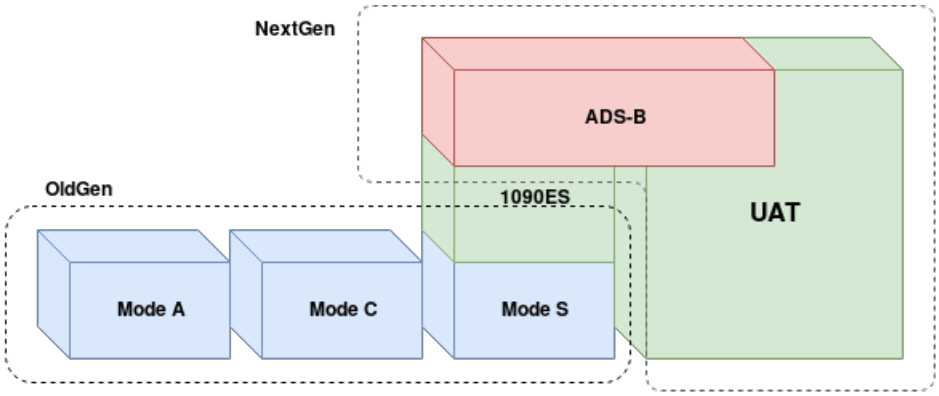

Figure 2.1: All generation of avionics protocols

Having a continuous data flow between the aircraft and the groud has many benefits:

- Flight controllers can obtain much more information about a single aircraft, his intentions, the status of the flight and real time weather information from the onboard sensors. Moreover satellite based ADS-B allows tracing aircrafts in areas where no radar coverage is available (e.g. over the ocean).

- The airline can keep track of his fleet in real time thus allowing a better planning of the turnover and quick deployiment of replacement aeroplane.

- The maintenance branch of the airline can track in real time any anomalies reported by the aircraft instruments and sensors performing a remote analysis and helping the pilots in the troubleshooting process helping them to take better decisions.

2.1 OldGen

T0D0 4: acars messages. explain how 1090mhz/1030MHZ protocols works.

The first generation of protocols is fairly simple, it uses the 1030MHz frequency for interrogation while the 1090MHz frequency is used for replies. The first two protocols are **Mode A** and **Mode C**.

Mode A is the simplest, it responds to an interrogation request just by broadcasting the **squawk** code which was previously assigned to the pilot by the controller. In this way the controller can identify the aircrafts on his screen by such code. In addition to this the pilot can manually generate a special response called *"Ident"* which is used to highlight the aircraft on the controller screen[4].

T0D0 5: images of the signal???

Mode C is an extention of the previous protocol which, in addition to the transponder code, sends information about the altitude and the pressure. This kind of transponders are often referred as **Mode A/C** transponder.

Mode S is the newest protocol and it is an hybrid between the two generations. It is backward compatible with the **Mode A/C** but at the same time it is also part of the NextGen as it can be seen in Figure 2.1.

T0D0 6: ads-b/ads-c in between old and next or actually is in both

2.2 NextGen

T0D0 7: explain how 978/nexrad/tis-b/fis-b protocols works

T0D0 8: The American Federal Aviation Administration (FAA) as well as its European EUROCONTROL named ADS- B as the satellite-based successor of radar.

T0D0 9: maybe explain backwards compatibility with 1090 using 1090ES (extended squitter) also put the theory about using fis-b/tis-b in uat and 1090mhz band. (Find some references and obtain some documents FREE)

[4] Tecnical specifications of such response are defined in ICAO Annex 10 Volume IV.

T0D0 10: Where mode-ac and mode-s are used now

Chapter 3

Software Testing and Fuzzing

3.1 Software testing methods

T0D0 11: Short introduction on the various testing methodologies beside fuzzing

3.1.1 Black Box

For this methodology no access to the source code or knowledge of the design specifications and internal mechanisms of the program is required. The test is based only on what can be observed which means giving to the target different inputs and monitoring his behavior. If we reflect on this definition for a moment it is one of the many ways to describe fuzzing. Although the definition of Black Box Testing fits quite well the one of fuzzing they are not the same thing in fact black box testing is widely used and includes all the testing techniques where the only available information comes from the user interaction with the application. One example among all is SQL injection where the test is conducted against a plain web page without any previous knowledge about logic of the servers and the various scripts. Only interacting with them allows the user to get an idea of the inner working mechanisms, the interaction can be direct with the user via a manual testing or using automated testing tools for example SQLmap[1]. Black Box testing has many advantages: it does not require access to the source code so it is always applicable and even in the presence of source code it can still be effective. Since it is not source code dependent an effective test case against one program (for example a zip extractor) can be easily reused to test any program that implements the same functionality. However due to his simplicity this method can only scratch the surface of the application being tested, achieving a extremely low code coverage. Moreover complex vulnerabilities which requires multiple attack vector will never be triggered and can only be discovered with a White Box approach.

[1] http://sqlmap.org/

3.1.2 White Box

This methodology is nothing more than a source code review or analysis, it can be done either by a human or by automated tools. Human analysis in not always feasible and might be inaccurate since, especially in large projects, it is easy to miss one line or make simple mistakes. For this reasons automated source code analysis is the way to go in a white box approach. This method has many advantages mainly regarding code coverage since it is possible to test all the paths for vulnerabilities. However, source code rarely is available for projects outside the open source community and this approach still requires a human interaction to review the results and identify the parts that are actually related to security problems. Moreover the review might require significant time to complete, effectively bringing the performance down to the level of the BlackBox approach.

3.1.3 Gray Box

The definition of this technique is quite trivial because a lot of different methods can be seen as Gray Box approach. This takes the Black Box method as a starting point augmenting it with the use of common reverse engineering and binary analysis techniques using disassemblers, debuggers and similar tools. The majority of those tools still requires human interaction and reverse engineering a program can often be a tedious and hard job. However there are some automated tools that aims to facilitate this analysis, for example one of them is Ropper[2]. For this reasons the Gray Box technique is able to efficiently improve the coverage produced by the classic Black Box approach still requiring access only to the compiled application. However reverse engineering is a complex task and requires specific set of skills and tools which might not always be available.

3.2 The fuzzing world

"Fuzzing is the process of sending intentionally invalid data to a product in the hopes of triggering an error condition or fault." [2]

Fuzz Testing dates back to 1989 when it started as a project of the operating systems course at the University of Wisconsin-Madison. At his first stages it was nothing more than a simple black box technique to test the robustness of some UNIX utilities. The first two fuzzers *fuzz* and *ptyjig*, developed by two students, where incredibly successful in finding previously unknown vulnerabilities: they where able to crash around 30% of the considered utilities using the random inputs generated by the fuzzers[3]. It quickly become clear that this was a powerful and strong technique to test the robustness of a piece of software.

During the years fuzz testing evolved and developed gaining popularity and expanding to more and more fields (networks, files, wireless and more) until it became a proper field of research and engineering. As for today this field is quite vast and, using this kind of tools in the software testing phase, is starting to be

[2]https://github.com/sashs/Ropper

a best practice. However it is still uncommon to find fuzzing outside the security world, for this reason is hard to give a precise and unique definition for this technique. I will now try to give an overview of the modern fuzzing world with particular reference to the local fuzzers.

First of all they can be divided in two main categories:

- **Mutation Based (or Dumb)**: apply random mutations to the given input without any knowledge of the data that are being manipulated. Common mutations include bitflipping, shortening or expanding and similar. This method is really simple and is a pure brute force approach, it can be applied to user provided inputs as well as to automatically generated ones to progressively generate a valid input.

- **Generation Based (or Smart)**: requires some knowledge of the data or protocol that is being fuzzed, this information are then used to generate proper inputs. For example generating valid CRC codes for mutated strings or keeping the correct structure for particular files (jpg). Obviously this approach has some advantages but in some cases a mutation based approach can give better results as it can test programs even outside their functioning boundaries.

As always the best sits in the middle so a fuzzer that combines the two methods is the one that can give the most comprehensive results.

We can then relate fuzzers to the previously mentioned testing methods, in this way fuzzers can be further differentiated by the approach that they have to the problem and the information they require to test the binary. This divides the fuzzing world in three theoretical categories: **BlackBox Fuzzing, WhiteBox Fuzzing** and **GrayBox Fuzzing**. However fuzzing is not an exact science, in the real world choosing the right approach can be trivial and usually there is not a clear distinction between the different categories.

T0D0 12: Fuzzing approaches.

3.3 State of the art

There are many software available to do a multitude of fuzzing jobs. The choice is vast and goes from the simple hobby project to the more complex and advanced commercial software however,

T0D0 13: choosing a good fuzzer is a crucial step for the success of the task.

The following two software represents the state of the art in term of binary fuzzers.

T0D0 14: describe afl, afl-unicorn and peach.

3.3.1 American Fuzzy Lop

We ended up choosing `American Fuzzy Lop` or `AFL` [3] as it represents the state of the art in this category, it has a high rate of vulnerabilities discovery as stated int the "bug-o-rama trophy case" section of the website, it is widely used and in active development.

AFL has many different features that makes it a quite sophisticated software. It is a gray box fuzzer which require access to the source code in order to perform what it calls "instrumentation". This process consist in adding some custom code to the program that is being compiled; this automatic operation, performed by the custom compilers `afl-gcc` and `afl-clang-fast`, adds small code snippets in critical positions without weighting down the program but allowing the fuzzer to obtain information on the execution flow at run time. However, even without access to the source code, it can leverage the QEMU emulator to obtain information about the execution state of the binary though this slows down the fuzzing speed.

The information gathered by the fuzzer will then be used to mutate the input: the tool uses a genetic algorithm approach keeping a queue of interesting inputs that will then be mutated and appending a file to the queue if it triggers a previously unknown internal state. This has been empirically proven to be a successful approach in sintetizing complex files or sentences form scarce or null information[**aflblog**].

One of the best characteristic of afl is that is open source meaning that every person can improve the code and modify it to meet different needs. For example one really active researcher in the fuzzing field, Dr. Marcel Böhme[**mbhome**], created some really interesting variations of afl trying to achieve better code coverage[**aflfast**][**greybf**] and better path discovery[**pythia**].

afl-unicorn

T0D0 15: review

`afl-unicorn` is a fork of afl that was build to leverage the power of afl and fuzz trivial binaries *"For example, maybe you want to fuzz a embedded system that receives input via RF and isn't easily debugged. Maybe the code you're interested in is buried deep within a complex, slow program that you can't easily fuzz through any traditional tools."*[**aflunicorn**]. In particular it uses the unicorn engine[**unicorn**], which is a CPU emulator, to run the extracted context of a previously running instance of the same program making it architecture independent and easy to selectively fuzz. It might sound hard to understand but it is not: the unicorn engine allows us to fuzz a program in the exact same way as before but, in addition to this, we now

[3] http://lcamtuf.coredump.cx/afl/

have control over the memory, the CPU registers, and the code of the program. This is useful in many different situations for example when you do not have access to the source code or when you have the sources but you want to fuzz a precompiled binary, which is our case. Moreover `afl-unicorn` allow the user to specify portions of code to be fuzzed, for example a single function, and the fuzzing takes place directly inside the emulated memory manipulating the user defined region. It is also possible to write custom "hooks" when certain functions are called or particular addresses are reached so to skip particular functions or to trigger a defined behavior, such as a crash, if the program reaches a particular portion of the code.

While it is true that this approach is very powerful the template for the unicorn engine must be hand written and the process in quite time consuming (we wrote a tool to automatically generate it, more details in Section **??**). Moreover, we had some troubles getting the unicorn engine to work on the multicore server: while on two different laptops it has the exact same behavior when we try to run the engine on the server it has a completely different behavior.

3.3.2 Peach

T0D0 16: more detailed description

Another very popular fuzzer and "competitor" of afl is `Peach`, this is a bit different from afl since is capable of fuzzing different targets such as **network protocols, device drivers, file consumers, embedded devices, external devices** natively. Another difference is that while afl requires just an input file to start the fuzzing `Peach` in addition to this it requires a *"Peach Pit"* which is a file describing the protocol or the data format that you want to fuzz[**peach**]. Moreover as it is stated on the website: "Peach is not open source software, it's FREE software. Peach is licensed under the MIT License which places no limitations on it's use. This software license was selected to guarantee that companies and individuals do not have to worry about license tainting issues."[**peachlicense**]. For this reasons and since we believed that afl represented more the state of the art we did not choose `Peach`.

Chapter 4

Experimental Setup

4.1 Hardware Setup

T0D0 17: Describe the hardware used, why we used that hardware and why it can be useful to test also on such a hardware. Getting dedicated hw is difficult and expensive. This part is explained really well in the paper.

4.2 Software Setup

T0D0 18: Explain why we used what we used. Same as before, copy from paper and final report.

4.3 Tools

During the project I wrote a set of tools and scripts that played a major role in automating and simplifying some of the data processing and fuzzing tasks. Such tools are divided in two categories: **Data Tools** used to manage, modify and create the datasets. **Fuzzing Tools** used to facilitate and speed up the fuzzing process. The details and descriptions of the tools are as follows.

Data Tools:

- **converter.sh** and **runner.sh** this scripts are meant to interact with the data provided by the DO-358 zip file.This archive contains different files which are designed to be used with a dedicated test tool for real avionics hardware, for this reason there are 18 subfolders (called groups) and for each one of them there are 3 files: `TestGroupXX Procedures.doc`, `TestGroupXX Stimulus.csv` and a `bin` folder containing the actual data files. Each one of this folders contain many different files, each representing a unique type of information which has already been demodulated and is stored in a binary format. Since the program that we want to test only accepts data in a uplink(or downlink) format we wrote **converter.sh** to convert one single file

or an entire directory in the appropriate format. Every file contains just one encoded type of data and with many files feeding them into a program will be long and tedious. The **runner.sh** script will feed each file contained inside a specified directory through the specified program either interactively, namely stopping after each file and asking if the user wants to continue, or simply running all files at one.

- **message_generator.py** is a simple script that given the first part of a demodulated Mode-S message will calculate a correct CRC, it can do this virtually for every random string with a correct length. This is only for test purpose since we wanted to see what would happen when the test program is fed with messages composed by all 1 or all 0 having a valid CRC.

Fuzzing Tools:

- **start.sh**: afl can be parallelized on many cores, to do that a different command with slightly different parameters must be issued for every instance that you want to spawn. For this reason I wrote **start.sh** which takes as input the number of cores that the user wants to use and the other parameters required by the fuzzer. The script will then generate the required directories and then start the chosen number of fuzzers: 1 Master and $n - 1$ Slave.
- **unicorn_template_generator.py**: `afl-unicorn` requires quite some time to set up all the environment and gather all the information needed to properly write the template for the Unicorn engine. For this reason we wrote this script which will create the template for the Unicorn engine and populate it with proper addresses. The script is designed to be sourced into GDB while the debugger is on a breakpoint inside a function, it will not work if called inside the main since what we want to test is usually a particular function.
- **extract_from_memory.py** will dump the specified memory region from a running program in gdb so it can be used as input to `afl-unicorn`. More information on how `afl-unicorn` works and why this scripts are needed can be found in 3.3.1.
- **afl utility scripts**:
  - **killem_afl.sh** → stop a running instance of afl.
  - **wazzup_afl.sh** → get information and statistics on the running instances of the specified fuzzer.
  - **afl_noroot.sh** → run afl on a system where the user does not have root privileges.

4.4 Proceedings

T0D0 19: which test we run, why and the result

Chapter 5

Results Analysis And Conclusions

5.1 Caveats

T0D0 20: Put all the problems we had here

5.2 Results

T0D0 21: Put the significative Results here and the state of the fuzzers

5.3 Conclusions

T0D0 22: I can think about doing a separate chapter for the conclusion

Chapter 6

Future Work

T0D0 23: Future work and proprietary fuzzer developement

Bibliography

[1] International Civil Aviation Organization (ICAO). *Convention on International Civil Aviation (doc. 7300/9)*. 1944. URL: https://www.icao.int/publications/Documents/7300%5C_cons.pdf.

[2] Pedram Amini Michael Sutton Adam Greene. *Fuzzing: Brute force vulnerability discovery*. Addison Wesley, 2007.

[3] Bryan So Barton P. Miller Lars Fredriksen. "An Empirical Study of the Reliability of UNIX Utilities". In: (1990). URL: ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf.