Security Analysis of the Drone Communication Protocol: Fuzzing the MAVLink protocol

Karel Domin Eduard Marin Iraklis Symeonidis
KU Leuven
ESAT-COSIC and iMinds
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
karel.domin@student.kuleuven.be {first.surname}@esat.kuleuven.be

Abstract

The MAVLink protocol, used for bidirectional communication between a drone and a ground control station, will soon become a worldwide standard. The protocol has been the subject of research many times before. Through this paper, we introduce the method of fuzzing as a complementing technique to the other research, to find vulnerabilities that have not been found before by different techniques. The goal is to identify possible vulnerabilities in the protocol implementation in order to make it more secure.

1 Introduction

Currently, drones are used to support critical services such as forest fire and illegal hunting detection, search and rescue operations or to deliver medical supplies. For this purpose, they are often equipped with a navigation system (GPS), a camera and an audio interface. Furthermore, they have a radio that enables wireless communication with the Ground Control Station (GCS) or a remote control. Besides the clear benefits of using drones in all these services, they can also pose important security and privacy threats. The wireless communication channel opens up the door for several types of remote attacks. For example, adversaries could attempt to obtain sensitive data by eavesdropping the wireless medium, send malicious commands to the drone, or alter its software.

Previous research has focused on analysing the wireless communication protocols of commercial drones [3], and exploiting the lack of security measures in the communication channel [4, 5]. However, we are not aware of any security analysis of the drone's software. In this paper, we tackle this problem, and carry out a software security analysis of the MAVLink protocol, which is expected to become a world-wide standard within the DroneCode project [2]. More specifically, we investigate potential design or implementation protocol flaws using fuzzing techniques. The goal is to inject invalid or semi-invalid data to produce an unexpected software behaviour. We briefly formulate three different research questions that we would like to explore more in detail in the rest of this paper. This includes: (i) *how can we identify software security flaws in the MAVLink framework?*, (ii) *what are the consequences of exploiting these security flaws?* and (iii) *can we provide countermeasures to mitigate such issues?*.

2 Related Work

Most of the research conducted in the past years resulted in attacks against the security of drones. When a vulnerability was found, the vulnerability was exploited and the drone could be hijacked or could crash. Academic research had the goal of reproducing a certain attack, make a theoretical background or proof and try to come up with countermeasures to assure the security of the drone. A large portion of the

conducted research is about GPS spoofing. A GPS spoofing attack attempts to mislead the drone's GPS receiver by broadcasting fake GPS signals while pretending to be a legitimate GPS signal sent by a satellite. This attack can trick any device using GPS signals into changing its trajectory or make the device believe that it is at another location [11]. Other research focuses on the lack of security mechanisms like authentication and encryption. There is lot of bidirectional communication between the drone and the GCS. Many different types of communication channels can be used for this like WiFi, Bluetooth or Radio Channels. The major problem with these communication channels is that they are used without any form of encryption or that they are used together with weak encryption that can be cracked. Some research about the vulnerabilities of MAVLink includes adding encryption to the protocol [12, 5, 4, 13], but this has not yet been implemented in the MAVLink protocol. We want to look at the vulnerability analysis from another perspective. We want to search the space of software vulnerabilities of the MAVLink protocol. As far as we know, the fuzz-testing method has not yet been applied for analysis of the MAVLink protocol.

3 Background information

3.1 Fuzzing

Fuzzing is a technique for finding vulnerabilities and bugs in software programs and protocols by injecting malformed or semi-malformed data. The injected data may include minimum / maximum values and invalid, unexpected or random data. Subsequently the system can be observed to find any kind of unexpected behavior, e.g. if the program crashes. There are three main types of fuzzing variants that can be distinguished: Plain Fuzzing, Protocol Fuzzing and State-based Fuzzing [15, 16]. `Plain Fuzzing` is the most simple way of testing. The input data is usually made by changing some parts of correct input that has been recorded. It provides very little assurance on code coverage because it does not go very deep into the protocol [14]. In `Protocol Fuzzing`, the input is generated based on the protocol specifications like packet format and dependencies between field. This is called `Smart generation` and is able to create semi-valid input. The opposite, `dumb generation`, is the corruption of data packets without awareness of the data structure. Protocol fuzzers typically generate test cases with minimum values, maximum values [14, 17]. `State-based Fuzzing` is a fuzzing technique that does not try to find errors and vulnerabilities by changing the content of the packets, but instead attempts to fuzz the state-machine of the software [14]. The most common method is to start with a dumb and basic fuzzer and then increase the amount of intelligence when necessary to create a smarter fuzzer [18]. Depending on the availability of source code, we can also distinguish between `Black-box Fuzzing` and `White-box Fuzzing`. The actual techniques used for fuzzing are typically a combination of black-box or white-box fuzzing with dumb or smart fuzzing. Unlike black-box fuzzing, white-box fuzzing requires a greater testing effort, however it provides a better test coverage [19].

3.2 MAVLink

The Micro Air Vehicle Communication Protocol (MAVLink Protocol) is a point-to-point communication protocol that allows two entities to exchange information. It is used for bidirectional communications between the drone and the GCS. MAVLink is a part of the DroneCode project, governed by the Linux Foundation [20]. A MAVLink message is sent bytewise over the communication channel, followed by a checksum for error correction. If the checksum does not match, then it means that the message is

corrupted and will be discarded. Figure. 1 shows the structure of a MAVLink message. We will now give a brief description of the fields included in the message:

- ◇ *Magic*: indicates the beginning of a new messages.
- ◇ *Length*: indicates the length of the payload field.
- ◇ *Sequence number*: indicates the sequence number of the packet.
- ◇ *System ID*: ID of the sending system.
- ◇ *Component ID*: ID of the sending component.
- ◇ *Message ID*: ID of the message in the payload.
- ◇ *Payload*: payload of the packet, which contains the parameters of the message.
- ◇ *CRC*: checksum for validation.

MAVLink messages are handled by the `handleMessage(msg)` function. This function has a switch statement, handling the different message IDs [21].

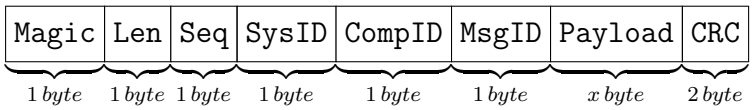

Figure 1: MAVLink packet structure

3.3 Fuzzing Methodology

For our experiments we built a fuzzer capable of creating custom MAVLink messages. Several strategies are applied to construct the messages that are sent to the drone. Initially, we started with a random dumb fuzzing to observe how the software handles invalid messages. We then made a smarter fuzzer which takes into account the message format, and constructs semi-valid messages. The techniques for constructing the payload of the messages are different for every test case.

4 Methodology

4.1 Lab Setup

Our laboratory setup employs the Software In The Loop environment (SITL) [7], which provides simulators for the ArduCopter, ArduPlane and ArduRover. We use the drone simulator for the ArduCopter [1]. The simulator is run on a Linux virtual machine and the fuzzer on a host machine. The host system is running OS X El Capitan (8gb RAM, 2,4 GHz Intel Core i5) and the virtual machine for the virtual drone is running Ubuntu 14.04 TLS 64-bit. The communication between both machines is via a TCP connection.

Configuration:

Name	Specification	Function	IP-address
Host System	Mac OSX	Host	192.168.56.1
System 2	VM2: Ubuntu	Virtual Drone	192.168.56.102

4.1.1 Case Studies

Our fuzzer, which is implemented in Python, is capable of constructing valid MAVLink messages. We now discuss how every field in the packet is constructed in our fuzzer. The `Magic` is a fixed value and is set to `"fe"`, whereas the *Length* field is set to the size of the payload field. In every transmitted message the `Seq` is increased by one, and it is reset to zero if it reaches the value of 255. The `SysID` and `CompID` are kept fixed, i.e. `"ff"` and `"00"`, respectively. The `MsgId` is a value in the range of `0-255`. The `payload` contains the parameters that are used internally, (e.g. the height of the drone), and is generated based on different strategies, which we will discuss more in detail for each experiment. The `CRC` value is generated using a CRC-16 function; its polynomial generator is 0x1021, the initial value is `FFFF`, the input data bytes are reversed, and the CRC result is reversed before the final XOR operation. The CRC parameters were obtained by looking at the available documentation and testing the generator on [23]. The input to the generation function is as follows: *Length+seq+SysID+CompID+MsgID+Payload+Seed*.
The seed is a x25 checksum generated over the message name, followed by the type and name of each field. This seed is used to capture changes in the XML describing the message definitions. This results in messages being rejected by the recipient if they do not have the same XML structure.

There are some properties that a fuzzer needs to have. A fuzzer must be able to record the test cases for reproduction. Therefore, every constructed message is written to a file before it is sent to the virtual drone. Another property is the ability of transmitting the test cases to the system under test. Since we are using SITL with a TCP connection therefore, the fuzzer initiates a three-way handshake with the virtual drone and a connection is established via sockets. The generated messages can now be sent to the drone over this socket. To observe the behaviour of the drone, we can use different approaches. A simple observation is to check whether or not the connection with the drone is still alive. To further investigate its behaviour, the virtual drone can be run inside gdb [22].

To start the virtual drone, the command in Listing 1 is used. This starts the Arducopter simulation for a quadcopter, at a certain location with all of the memory erased and faster operation. We define the following test cases.

Listing 1: Startup command

```
./arducopter.elf --home -35,149,584,270 --model quad
--speedup 100 --wipe
```

`Test Case 1` We establish a connection to the drone and start to send completely random data including numbers, letters and characters. The actual structure of a MAVLink message is ignored at the moment. The length of the data sent to the virtual drone ranges from 1 to 1000 characters. We do this to test how the software handles incorrect data.

`Test Case 2` For every message ID, we create a message with payloads of length ranging from the minimum length (i.e. 1 byte) to the maximum length (i.e. 255 bytes). The payload consists of completely random combinations of hexadecimal values. We repeat this test several times. This test is used to give an indication of how semi-valid messages are handled. This is important since these messages can go deeper into the software.

`Test Case 3` We also test the system's behaviour when messages without any payload are sent. For every message id, a new message (with no payload) will be constructed with the length set to zero. This test aims to find vulnerabilities that do not depend on the payload.

`Test Case 4` We construct payloads consisting entirely out of the minimum value. This test investigates how the implementation handles the minimum value `"00"`. This is done for payloads with a length ranging from 1 to 255 bytes.

`Test Case 5` following the previous test case, we do the same for the maximum value `"ff"`.

`Test Case 6` Within this test case, we do not send the messages byte per byte. An entire message with all the necessary fields is included in one TCP packet. We incrementally increase the length of the random payload from 1 to 255 bytes extending the length of the entire message.

`Test Case 7` Within this test case, we try to identify vulnerabilities depending on the value of the length field and the actual value of the payload. In a first run, we constructed messages with up to 5 bytes of payload and set the value of the length field to the length of the payload minus one. In the second run, we did the same, except that the value of the length field was set to the length of the payload plus one.

5 Results Obtained and Discussion

Resulting from the listed test cases, we were able to identify a few security flaws. Particularly, from the sixth test case, where the payload increased randomly, the fuzzing script was able to crash the virtual drone. The error caused by the fuzzing script can be seen in Listing. 2.

Listing 2: Flaoting Point Exception

```
ERROR: Floating point exception - aborting
Aborted (core dumped)
```

To investigate the cause of the exceptions we used the gdb debugger and the core dump of the memory when the kernel crash occurred. From an analysis we identified that errors corresponds to three specific functions. However, further investigation needs to be performed to identify the exact cause of the exceptions.

The next step of our work is to complete the entire range of the test cases aiming to gain more results identifying error flaws of the MAVLink software implementation. Moreover, we aim to further investigate the causes of the identified software flaws. However, we have to stress that fuzzing all possible test cases is a resource demanding operation. For instance, there is a limitation concerning to the memory usage. With the current setup, it is not possible to try all possible permutations of payloads, for all possible message and payload lengths. Currently, we are looking to further improve the fuzzing scripts aiming to make the fuzzing operations more memory efficient.

6 Conclusion

This work aims to identify software vulnerabilities, by using the technique of fuzzing. Currently we focused our research on the MAVLink protocol. MAVLink is used as a communication protocol between a drone and a ground control station. The protocol is actively developed by the community and aims to become one of the drone communication standards. Our aim is to contribute to the identification of the security flaws of the protocol and to help the development community to mitigate these flaws. At the same time, we want to proof the suitability of fuzzing techniques for discovering vulnerabilities in the implementation of the protocol. Currently, we have identified software vulnerabilities that we are further investigating.

Many fuzzing platforms do already exist and can be used for the software analysis of the MAVLink protocol. Our next step is to further research and extend our fuzzing scripts aiming to generate more complex fuzzing scenarios.

7 Acknowledgements

This work was supported in part by the Research Council KU Leuven (C16/15/058).

References

[1] ArduCopter, https://www.dronecode.org/, 24 03 2016.

[2] DroneCode, http://ardupilot.org/copter/index.html, 24 03 2016.

[3] B. Hond, *Fuzzing the GSM Protocol*, Radboud University Nijmegen, Netherlands, 2011.

[4] J. A. Marty, *Vulnerability Analysis of the MAVLink Protocol for Command and control of Unmanned Aircraft* Air Force Institute of Technology, USA, 2014.

[5] N. Butcher, A. Stewart and Dr. S. Biaz , *Securing the MAVLink Communication Protocol for Unmanned Aircraft Systems*, Appalachian State University, Auburn University, USA, 2013.

[6] Sulley Manual, http://www.fuzzing.org/wp-content/SulleyManual.pdf, 24 03 2016.

[7] Software In The Loop, http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html, 24 03 2016.

[8] AR drone that infects other drones with virus wins dronegames, http://spectrum.ieee.org/automaton/robotics/diy/ar-drone-that-infects-other-drones-with-virus-wins-dronegames, 03 05 2016.

[9] SkyJack, https://github.com/samyk/skyjack, 03 05 2016.

[10] Maldrone, http://garage4hackers.com/entry.php?b=3105, 03 05 2016.

[11] GPS Spoofing, https://capec.mitre.org/data/definitions/628.html, 03 05 2016.

[12] Thomas M. DuBuisson, Galois, Inc.1 , *SMACCMPilot Secure MAVLink Communications*, Galois, Inc.1, 2013.

[13] MAVLink 2.0 packet signing proposal, https://docs.google.com/document/d/1ETle6qQRcaNWAmpG2wz0oOpFKSF_bcTmYMQvtTGI8ns, 03 05 2016.

[14] B. Hond, *Fuzzing the GSM Protocol*, Radboud University Nijmegen, 2011.

[15] A. Takanen, C. Miller, J. DeMott *Fuzzing for Software Security Testing and Quality Assurance*, ARTECH HOUSE, INC., 2008.

[16] A. Greene, M. Sutton, P. Amini *Fuzzing Brute Force Vulnerability Discovery*, Addison-Wesley, 2007.

[17] OWASP Fuzzing, https://www.owasp.org/index.php/Fuzzing, 03 05 2016.

[18] 15 Minute Guide to Fuzzing, https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/, 03 05 2016.

[19] J.eystadt, *Automated Penetration Testing with White-Box Fuzzing*, Microsoft Corporation, 2008.

[20] MAVLink Protocol, `http://qgroundcontrol.org/mavlink/start`, 03 05 2016.

[21] S. Balasubramanian, MAVLink Tutorial for Absolute Dummies (part-I), `http://dev.ardupilot.com/wp-content/uploads/sites/6/2015/05/MAVLINK_FOR_DUMMIESPart1_v.1.1.pdf`, 03 05 2016.

[22] GDB: The GNU Project Debugger , `https://www.gnu.org/software/gdb/`, 03 05 2016.

[23] CRC Generator, `http://www.zorc.breitbandkatze.de/crc.html`, 03 05 2016.