KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

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

Karel Domin

Thesis submitted for the degree of Master of Science in Engineering: Computer Science, specialisation Secure Software

Thesis supervisors:
Prof. dr. ir. B. Preneel
Prof. dr. ir. F. Piessens

Assessors:
Prof. dr. T. Holvoet
dr. Ir. M. Mustafa

Mentors:
Ir. E. Marin
Ir. I. Symeonidis

Academic year 2015 – 2016

Preface

I would like to thank my mentors for guiding me through the research and writing process, proof reading my text and providing me with useful, constructive feedback. My sincere gratitude also goes to my friends and family, for supporting me throughout the entire process.

Karel Domin

Contents

Abstract

In our current society, commercial drones have become common. They are used by hobbyists as well as by professionals to support critical services. Even though drones can provide many benefits, they can also pose serious security threats. Several attacks against drones have emerged the last few years. Together with the attacks, a considerable amount of research is performed trying to establish secure drone operations. An important part of the drone security research is focused on the wireless communication channel. This channel is vulnerable because the lack of security measures. Research did focus on providing a secure wireless communication channel. However, securing the wireless channel alone is not enough, there is also a need for a secure implementation. In our research we focussed on the MAVLink protocol because this is trying to become a worldwide standard. Our goal is to carry out a security analysis of the MAVLink protocol and investigate potential design or implementation flaws in the protocol. For this, we use fuzzing techniques. In our work we created a fuzzer capable of generating random and semi-valid MAVLink messages which can be injected in the protocol. Our experiments resulted in the generation of a floating point exception, which made the virtual drone crash. After analysing the exception, other test cases were constructed, further refining our test method. This led to the identification of 15 vulnerable implementation parts were the floating point exceptions can be thrown. Further analysis of the results indicated that a certain combination of sent messages with random data posed a vulnerability for the drone. We were able to identify a certain pattern in the implementation that is vulnerable to this attack.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

UAV	Unmanned Aerial Vehicle
GCS	Ground Control Station
GPS	Global Positioning System
MAVLink	Micro Air Vehicle Communication Protocol
SITL	Software In The Loop
EEPROM	Electrically Erasable Programmable Read-only Memory,

Chapter 1

Introduction

This section introduces the conducted research. We start with the motivation for the research. This will be followed by the related work, our research goals, research questions and our approach.

1.1 Motivation

Unmanned aerial vehicles (UAVs), also known as drones, are vehicles that are controlled remotely from a Ground Control Station (GCS) or autonomously by a programmed mission. There are two main purposes of drones that can be identified, drones for military usage and drones for commercial usage. In our work, we focus on the commercial drones.

Currently, commercial 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 a GCS or a remote control. Besides the clear benefits of using drones, they can also pose important security and privacy threats. In particular, 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 even alter its software. If a drone is hijacked it can pose serious threats.

1.2 Related Work

This section starts with a short overview of projects that resulted from experimenting with drones. Subsequently a typical drone setup is discussed with the associated security risks. We take a closer look at the GPS spoofing attack and the security of the wireless communication channel.

In the last decade, the amount of commercial drones sold increased enormously. Drones are getting cheaper and people start experimenting with them, resulting in interesting projects. For example, `VirusCopter` consists of a virus, acting as a worm, capable of infecting AR Drones [2]. Similarly SkyJack is a drone engineered to autonomously find drones within flying distance and take full control over them. The attacker drone disconnects the target drone from its wireless connection with the GCS and then connects with the drone, pretending to be its owner [3]. `MalDrone`, which can be used in combination with SkyJack, can be installed on the drone as a backdoor for the parrot AR.Drone, killing the current autopilot system and take over control [4].

Figure 1.1 gives an overview of the setup for drone operations and their possible vulnerabilities. This setup consists of a drone, a GPS module and a GCS or remote control, which provide the basis for vulnerabilities. For instance, one can alter or find vulnerabilities in the software of the drone or exploit the internal state-machine. Subsequently, the communication channel between the drone's GPS module and the GPS satellite makes the drone vulnerable to GPS spoofing attacks, whereas the channel between the drone and the remote control or GSC allows for attacks such as eavesdropping or replay attacks.

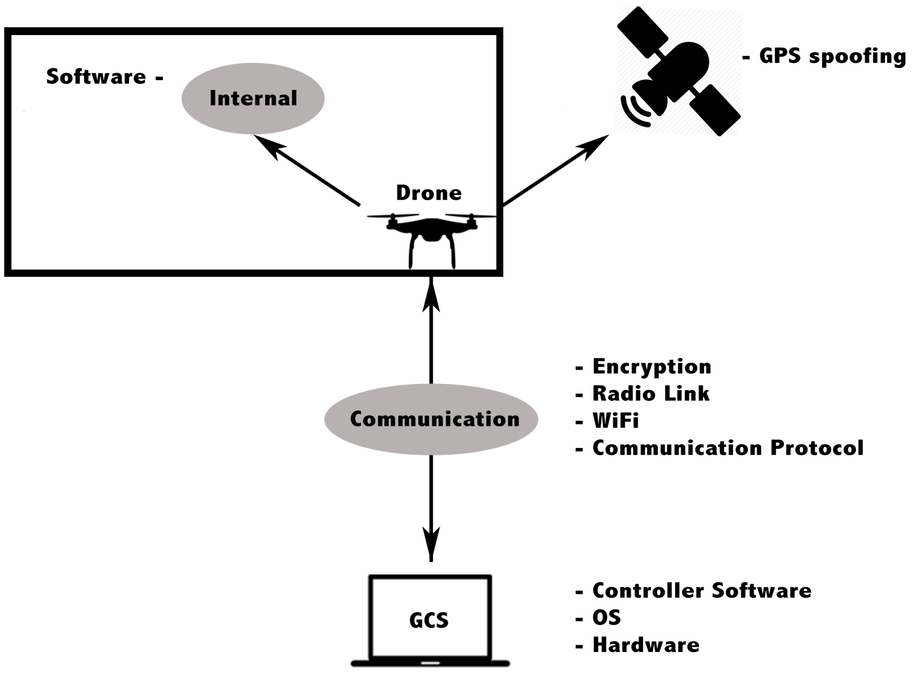


Figure 1.1: Drone Vulnerabilities

Several articles have conducted research on 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 [5]. Tippenhauer et al. studied the requirements for successful GPS spoofing attacks on individuals and groups of victims with civilian or military GPS receivers [6].

Other research focuses on the communication channel between the drone and the GCS. 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. For example, in 2009, militants in Iraq used a 26 dollar software called SkyGrabber to intercept live video feeds from U.S. Predator drones, lacking secure communication channels. Pleban et al. described the security problems of the Parrot AR.Drone 2.0 quadcopter. They found that after powering up the drone, it sets up an open, unencrypted, WiFi access-point. By scanning all ports they discovered that ports 21 (FTP) and port 23 (Telnet) are open. Since the FTP port is not password protected, they can access confidential data. The Telnet port can provide access to a root shell, allowing attackers to perform malicious actions [7]. Adding encryption to the communication channel is a simple way to solve the problem. However, this was rejected because of the extra costs [8].

A possible communication protocol for bi-directional communication between the drone and a GCS is the MAVLink protocol and is attempting to become a worldwide standard. The MAVLink protocol does not provide any security measures. Research by J. Marty proposed a methodology to quantify the cost of securing the MAVLink protocol through the measurement of network latency, power consumption, and exploit success [9]. In addition, the research of Thomas M. Dubuisson introduces a light-weight encapsulation format that can be used with MAVLink to protect against forgery, replay attacks, and snooping. This is accomplished by using cryptographic solutions, and adds an additional 16 bytes to each message [10]. Subsequently, N. Butcher et. al studied the possible wireless attacks on MAVLink, and proposed the RC5 encryption algorithm as a countermeasure to these attacks and studied how encryption affects the performance of the drone [11]. We can conclude that adding encryption to the protocol is needed to protect sensitive data, but until today this has not yet been implemented in the official distribution of the MAVLink protocol.

However, securing the wireless channel alone is not enough, we also need a secure implementation. In this work we want to search the space of software vulnerabilities of the MAVLink protocol and look at the same security problems from a different perspective. We need a combination of both strong security measures and a secure implementation.

1.3 Our Approach

Our goal is to carry out a software security analysis of the MAVLink protocol and investigate potential design or implementation protocol flaws. For this, we use fuzzing techniques. The goal of fuzzing is to inject invalid or semi-invalid data to produce an unexpected software behaviour or crash the software. To the best of our knowledge, the fuzz-testing technique has not yet been applied for analysis of the MAVLink protocol. In addition, we want to study the suitability of the fuzzing technique as complementing to the standard test methodologies for identifying possible security flaws in the implementation of the MAVLink protocol. For our research, the MAVLink protocol specifications such as the message structure and message handling methods have been examined in detail and based on these specifications a fuzzer was constructed. The fuzzer is able to create MAVLink messages in an automated manner.

We briefly formulate three different research questions that we would like to explore more in detail in the rest of our work. This includes: (i) *how can we identify software security flaws in the MAVLink framework?*, (ii) *Is the fuzzing technique suitable for discovering implementation flaws in the MAVLink protocol?* and (iii) *what are the consequences of exploiting these security flaws?*.

1.4 Organization of the Thesis

Chapter 2 provides an introduction to the Software In the Loop environment for simulating a drone and describes the concept of fuzzing. Since the MAVLink protocol is the main subject of our research, it is explained in detail in chapter 3. Chapter 4, explains the methodology of the research containing the lab setup, the fuzzing methodology and the different case studies. Chapter 5, gives an overview of the results of the case studies, together with a brief analysis and further refined case studies. Chapter 6 discusses the results of the previous analysis and chapter 7 states the lessons we have learned throughout the research and all the problems and limitations that were encountered. We conclude with final remarks and suggestions for future work in chapter 8.

Chapter 2

Background Information

In this section, the background information needed to understand our research is given.

We will discuss the Software In The Loop environment as a simulator for the drone. Subsequently, the concept of fuzzing will be explained.

2.1 Software In The Loop (SITL)

Software In The Loop (SITL), provides simulators for the ArduCopter, ArduPlane and ArduRover from the DroneCode project (see chapter 3). The SITL simulator allows to examine the behaviour of the drone without needing to have the hardware. It is a build of the drones operation system using a C++ compiler. Because ArduPilot is a portable autopilot, it can run on many different platforms like Linux and Windows [1]. Furthermore, SITL provides access to development tools, such as interactive debuggers, static analyzers and dynamic analysis tools. This makes developing and testing in ArduPilot much simpler. An overview of the SITL architecture which shows how the simulator is connected to the GCS, is shown in Fig 2.1.

2.2 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 or maximum values and invalid, unexpected or random data. After the data is injected, the system can be observed to find any kind of unexpected behaviour, e.g., if the program crashes or failing built-in code assertions, that other testing techniques missed. The technique of fuzzing is illustrated in Fig 2.2.

There are three main types of fuzzing variants that can be distinguished: Plain Fuzzing, Protocol Fuzzing and State-based Fuzzing [12, 13].

`Plain Fuzzing` is the most simple way of testing. The input data can be completely random or constructed by changing some parts of correct input that has been recorded. These types of fuzzers, also called `dumb fuzzers`, require almost no

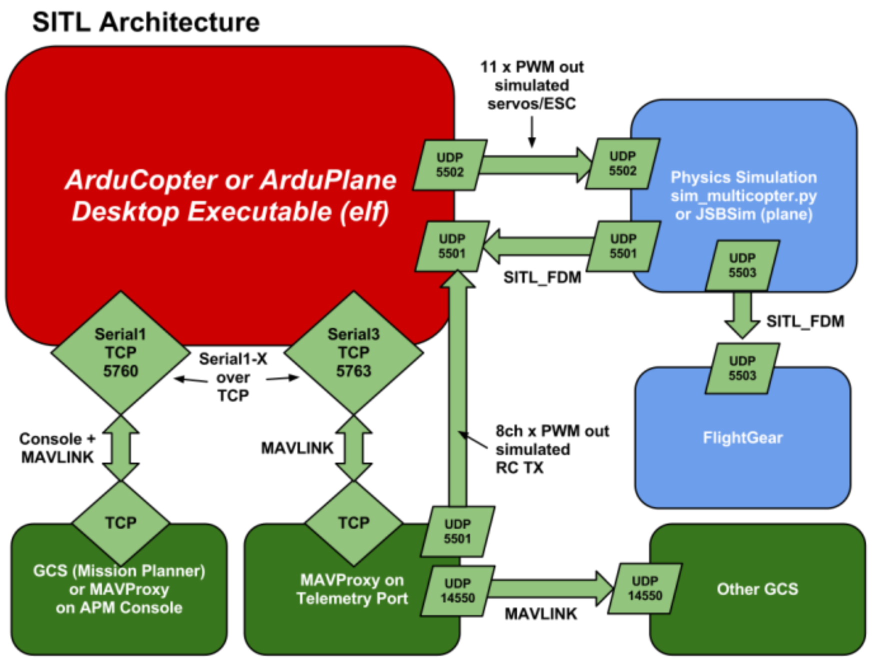

SITL Architecture
ArduCopter or ArduPlane Desktop Executable (elf)
11 x PWM out simulated servos/ESC
UDP 5502
UDP 5502
UDP 5501
UDP 5501
SITL_FDM
Physics Simulation sim_multicopter.py or JSBSim (plane)
UDP 5503
SITL_FDM
UDP 5503
FlightGear
Serial1 TCP 5760
Serial1-X over TCP
Serial3 TCP 5763
Console + MAVLINK
MAVLINK
8ch x PWM out simulated RC TX
TCP
TCP
GCS (Mission Planner) or MAVProxy on APM Console
MAVProxy on Telemetry Port
UDP 5501
UDP 14550
MAVLINK
UDP 14550
Other GCS

Figure 2.1: SITL Architecture from [1]

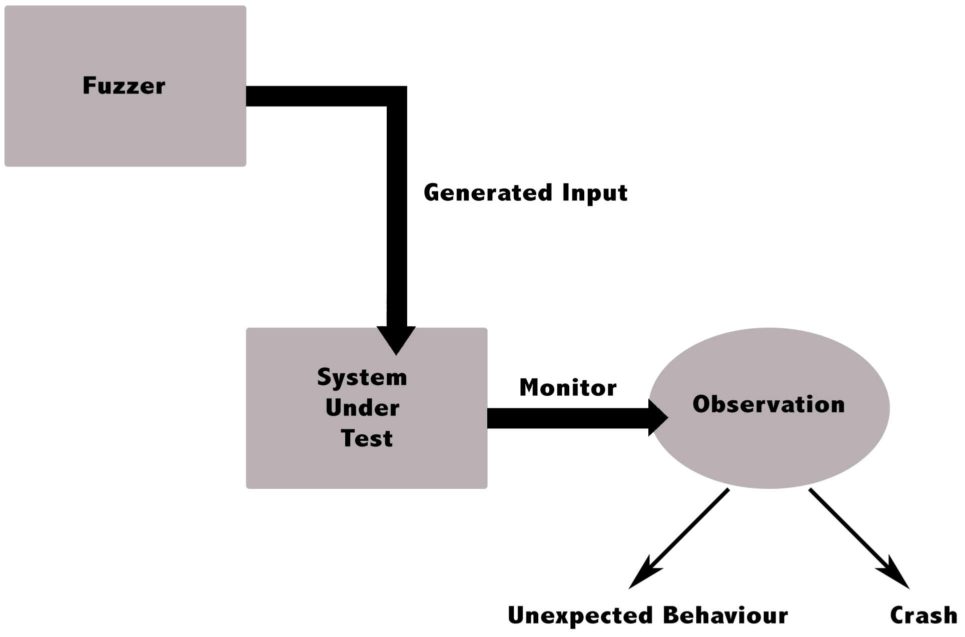


Figure 2.2: Fuzzing Technique

knowledge to create input data. As a result it provides very little assurance on code coverage because it may not execute all the sub functions of a function as the data is malformed [14].

`Protocol Fuzzing` is a more sophisticated way of testing. The input is generated based on the protocol specifications like message format and dependencies between field. This is also called `smart generation` and is able to create semi-valid input. This can be necessary if the input needs to be well formed. The advantage of this technique is that the more intelligence is used, the deeper the fuzzing can penetrate into the software. Protocol fuzzers typically generate input data from scratch, filling the fields with minimum and maximum values [14, 15].

`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. This can be done by breaking the sequence of messages that is typically used in the protocol. The impact of this kind of fuzzing technique depends on which states are skipped [14].

Typically, the fuzzing strategy is to start with a dumb and basic fuzzer and then increase the amount of intelligence when necessary to create a more sophisticated fuzzer [16]. The fuzzing technique is capable of finding faults in error handling, clean-up code and by using state-based fuzzers, fuzzing is able to find faults in state machine logic [17].

A typical fuzzer has several tasks to perform:

1. Generation of the test cases
2. Recording or logging of the test cases for reproduction
3. Transmitting the test cases as input to the tested program
4. Observing the behaviour and detect crashes

Chapter 3

MAVLink Protocol

The Micro Air Vehicle Communication Protocol allows entities to communicate over a wireless channel. When used in drones, it is used for the bidirectional communications between the drone and the GCS. The GCS sends commands and controls to the drone whereas the drone sends telemetry and status information. MAVLink was first released in 2009 under the GNU Lesser General Public License and is now part of the DroneCode project, governed by the Linux Foundation [18]. Currently, the DroneCode project has thousands of developers and over a hundred thousand of users.

MAVLink is used in many different Autopilot systems like ArduPilotMega, pxIMU Autopilot, SLUGS Autopilot, etc. It is also used in software packages like iDoneCtrl (IOS), QGroundControl (Windows/Mac/Linux), APM Planner, MAVProxy, etc. Finally, there are many projects using MAVLink, these include ArduPilotMega, MatrixPilot, PIXHAWK, etc. A full list of Autopilot systems, software packages and projects using MAVLink can be found in Appendix B.

MAVLink messages are defined in an XML file and are then converted to C/C++/C# or Python code via existing code generators. This offers the opportunity te create custom messages. We give a brief example of this process by using the `heartbeat` message, taken from the QGroundControl website [19]. The `heartbeat` message is used by both the communicating entities to indicate they are active and still connected. The XML definition of the heartbeat message is shown in Listing B.1.

After the definition of the messages is written down in the XML, it can be compiled to C/C++/C# or Python code. The corresponding generators with the details of their usage can be found on the website of QGroundControl [19]. If the C-generator is used, the heartbeat message will be converted to a C-struct as can be seen in Listing B.2.

Additionally, functions are generated to serialize (pack) and deserialize (unpack) messages. The example code for the heartbeat message can be seen in Listing B.3.

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. 3.1 shows the structure of a MAVLink message. The minimum length of a MAVLink message is 8 bytes and the maximum length, with full payload, is 263 bytes. We will now give a brief description of the fields included in the message. The *Sync* fields indicates the beginning of a new messages. The *Length* indicates the length of the payload field. A *Sequence number* is included indicates the sequence number of the packet for detecting packet loss. The *System ID* and *Component ID* identify the sending system and component respectively. The *Message ID* is the ID of the message to indicate how to handle the payload. The *Payload* contains the actual data of the message. Add the end, a *CRC* is included as a checksum for validation.

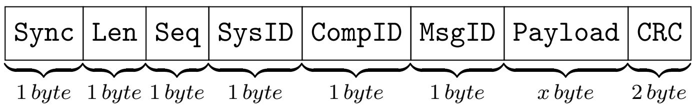


Figure 3.1: MAVLink Message Structure

Once the message is received, the value of the Length field is read (n), and is used to verify the checksum after n bytes. If the checksum is valid, the message is translated in the software, if the checksum is a mismatch, the message will be dropped. Once the message is translated and valid, the software starts interpreting the message by calling the function `handleMessage(msg)`. This function has a switch statement, handling the different message IDs. While handling the message, the payload is extracted and put inside a packet, representing a certain type of information. The packet is then further processed through the software [20].

A special part of the MAVLink protocol is the MAVLink Mission Interface. This is a data format allowing to store missions to be carried out by drones. The fact that it stores data on the drone makes it an interesting subject to research, since it enables to alter the internal memory or state of the drone. The data sent to the drone can include waypoints or advanced features, which can be transmitted to the by sending `command` messages. A `command` message is stored in a 14 byte array and the structure can be seen in Table 3.1 [21].

The Command ID of the message defines three different categories of commands:

1. **NAV Commands**: navigation commands to control the movement of the drone.

2. **DO Commands**: auxiliary functions for advanced settings.

3. **Condition Commands**: delay a DO Commands until a certain condition is met.

Parameters 1 to 4 are used for generic commands and parameters 5 to 7 are used for navigation commands.

Byte #	**Address**	**Data Type**	**Function**
0	0x00	byte	Command ID
1	0x01	byte	Options
2	0x02	byte	Parameter 1
3	0x03	long	Parameter 2
4	0x04	..	
5	0x05	..	
6	0x06	..	
7	0x07	long	Parameter 3
8	0x08	..	
9	0x09	..	
10	0x0A	..	
11	0x0B	long	Parameter 4
12	0x0C	..	
13	0x0D	..	
14	0x0E	..	

Table 3.1: Command Message Structure

Chapter 4

Methodology

This chapter describes the setup that is used for the experiments. In the real world, the setup consists of a drone and a GCS. In our experiments we use a simulator from the SITL environment from section 2.1, providing a virtual drone. First, the lab setup is explained, describing the used equipment, software and how to connect all components together. Afterwards, the fuzzing methodology is described in detail and the test cases are presented.

4.1 Lab Setup

The SITL simulator is run on a Linux virtual machine (Ubuntu 14.04 TLS). The laboratory setup employs the simulator for a multirotor UAV, ArduCopter. The simulator can be run as shown in Listing 4.1.

Listing 4.1: Run SITL

```
$ ./ArduCopter.elf --home -35,149,584,270 --model quad --
    speedup 100 --wipe
Started model quad at -35,149,584,270 at speed 1.0
Starting sketch 'ArduCopter'
Starting SITL input
bind port 5760 for 0
Serial port 0 on TCP port 5760
Waiting for connection ....
```

This starts the ArduCopter simulation for a QuadCopter, at a home location with coordinates *35,149,584,270*, all of the internal memory wiped and faster operation. The output shows that after execution of the command, a connection to the SITL can be established using TCP/IP at the network address of the machine running STIL at port 5760. In the normal setup, SITL is used in combination with a GCS, like MAVProxy (see chapter 3). Since the GCS is replaced by a fuzzer, we do not need to use MAVProxy.

The fuzzer is running on the host machine and written in Python 2.7. The host system is running OS X El Capitan (8gb RAM, 2,4 GHz Intel Core i5). Both

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

Table 4.1: Configuration for Lab Setup

machines are using static IP addresses to facilitate the network analysis. Figure 4.1 shows an overview of the operational and the lab setup. The used configuration is shown in Table 4.1.

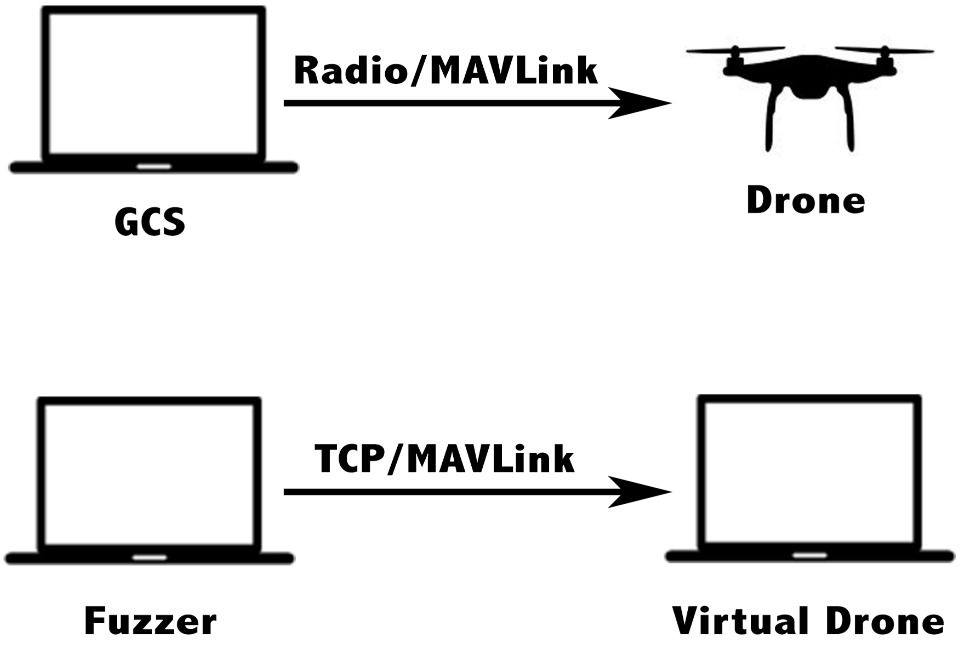


Figure 4.1: Operational Setup Vs. Lab Setup

The fuzzer and the virtual drone communicate over a TCP connection, as this is built into the SITL environment. The fuzzer establishes a TCP connection by performing a three-way handshake. The steps of the handshake are as follows:

1. **SYN**: The client (fuzzer) sends a SYN message to the server (simulator). The sequence number of the message is set to a random value X.
2. **SYN-ACK**: The server replies with a SYN-ACK message. The acknowledgement number is set to the received sequence number plus one (X+1), and the sequence number of the reply is chosen by the server and is a random number Y.
3. **ACK**: The client sends an ACK message back to the server. The sequence number of this message is set to the received acknowledgement number from the SYN-ACK message, and the acknowledgement number is set to the received sequence number plus one (Y+1).

The message sequence for the three-way handshake is illustrated in Figure 4.2.

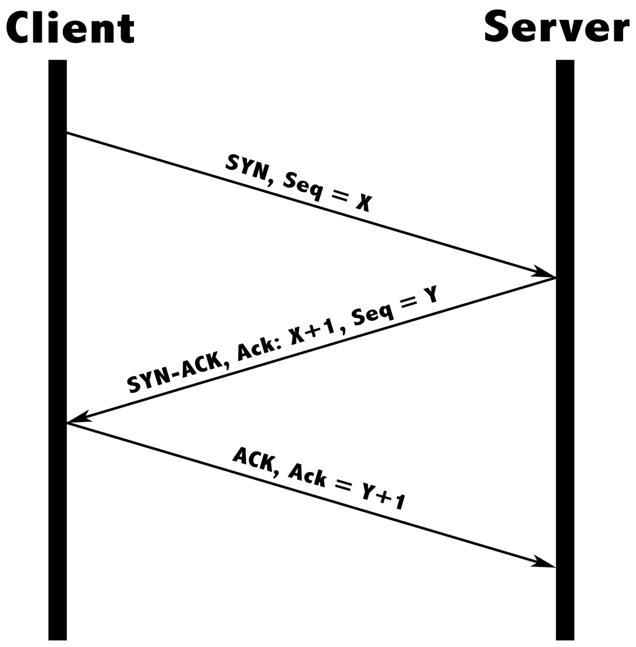


Figure 4.2: Three-Way Handshake

Establishing the connection in Python can easily be done in three lines of code as shown in Listing 4.2. These lines of code simply create a new stream socket and use the `connect()` method to initiate the three-way handshake protocol with the machine at the given network address and port. After the connection is established, data can be sent by invoking the `send()` method on the created socket.

Listing 4.2: Establish TCP Connection

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.56.102", 5760))
```

4.2 Fuzzing Methodology

For our experiments we have build a fuzzer capable of creating custom MAVLink messages. Several strategies are followed to construct the messages that are sent to the drone. Initially, random dumb fuzzing is used to observe how the software handles invalid messages. Afterwards, smarter fuzzing techniques are used while taking into account the message format, constructing semi-valid messages. The value of every field is represented by a hexadecimal value. The structure of the MAVLink message as mentioned in Figure 3.1 is used by the fuzzer as described below.

The Sync is a fixed value and is set to "fe", indicating the start of a new message. The value of the Length field indicates the size of the payload field, which ranges from zero to 255 bytes. This means that the minimum and maximum value of the length field are (zero) "00" and (255) "ff" respectively. In every transmitted message the Seq is increased by one, and it is reset to zero if it reaches 255. The SysID and CompID are kept fixed, i.e., "ff" and "00", respectively. The MsgId is a value within the range "00" to "ff". The payload contains the parameters that are used internally, such as the height of the drone). This field is generated based on the different strategies, which will be discussed more in detail in each of the experiments.

The CRC parameters were obtained by looking at the available documentation and experimenting on a website with a crc generator [22]. The generator was essential to acquire all necessary parameters. We found that the CRC value is generated using a CRC-16 function, which has the following parameters: 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 input to the generation function is as follows: *Length+seq+SysID+CompID+MsgID+Payload+Seed*. The seed is a 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 (see chapter 3 for more details about the XML structure). The Python code for generating the correct CRC can be seen in Listing 4.3

Listing 4.3: Generate CRC

```python
from crcmod import *

crc16_func = crcmod.mkCrcFun(0x11021, initCrc=0xFFFF)  # Configure CRC function

def calc_crc_16(data):
    s = unhexlify(data)
  reversed_crc_value = crc16_func(s)
  reversed_crc_value_hex = format(crc16_func(s),'x')
  x=reversed_crc_value_hex[0:2] # Reverse the values
  y=reversed_crc_value_hex[2:4]
  return [str(y.zfill(2)), str(x.zfill(2))] # Add padding to the result
```

There are some properties that a fuzzer needs to have, as mentioned in section 2.2. 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. The generated messages can be sent to the drone over the socket that is used to establish the connection, as described in section 4.1.

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 the gdb debugger [23].

4.3 Case Studies

In the following section we give a description of every test case that has been conducted. For some of the test cases the python code is included as an example. For the remainder of the test cases, the source code can be found in Appendix A. Before a message is sent to the virtual drone, it is written to a log for reproduction or analysis. We also write the begin and end times of a test case to this log for time statistics.

`Test Case 1a.` In the first test case, we start with what we called `Dumb Fuzzing`. We do not use any intelligence about the protocol and we do not take into account the actual structure of a MAVLink message. We establish a connection to the drone and start to send completely random data including numbers, letters and characters. Important to notice is that the crc also consists of random data. The data is sent character by character to the drone. This means that every character is included in a different TCP packet and received solely by the drone. We start by sending one character and increase the amount of characters until a length of 1000 characters is reached. This test is conducted to inspect how the software handles random data. The code for this test case is shown in Listing 4.4. The corresponding message structure can be seen in Figure 4.3.

Listing 4.4: Test Case 1

```
def case1(iteration):
  for l in range(1,1001):
    start = time.time()
    f = open('tests/complete_random/'+'_length_'+str(l)+'
      _it_'+str(iteration), 'w')
    f.write('Start Time: '+str(start)+'\n')
    a=[]
    a.extend("".join([random.choice(string.printable) for
        i in
    xrange(l)]))
    f.write('string: ' + str(a) +'\n')
    for y in range(0,len(a)):
      s.send(a[y])
      data = s.recv(4096)
    end = time.time()
        f.write('End Time: ' +str(end) +'\n')
        f.write('Completed in: ' + str(end-start))
```

`Test Case 1b.` In this test case, we follow the same strategy as the first test case. The only difference is that the data is not sent character per character. We

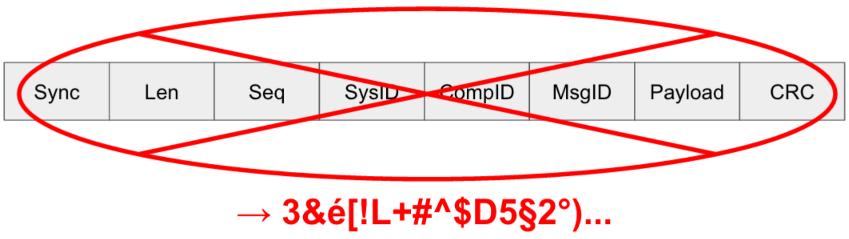


Figure 4.3: Message Structure for Test Case 1

try to include as much data as possible in one TCP packet. We perform this test case to observe how the drone handles incoming data that is not sent bytewise as the protocol expects. This test case requires only two lines of code to be changed compared to the previous test case, as shown in Listing 4.5.

Listing 4.5: Test Case 1b

```
a="".join([random.choice(string.printable) for i in
    xrange(l)]))
f.write('string: ' + str(a) +'\n')
s.send(a)
```

`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 to cover more possible combinations for the payload. This test is used to give an indication of how semi-valid messages are handled. Because we are using the correct sync value, sending system and sending component ID and a valid checksum, the messages will be accepted by the virtual drone. Since the structure is correct, the messages will go through many more functions until the actual random data, the payload, is used.This results in higher code coverage. The corresponding message structure can be seen in Figure 4.4.

The code for generating payload is shown in Listing A.1. The parameters of this method include the number of the test case and the length that the payload should have.

`Test Case 3.` We also want to test the behaviour of the virtual drone when the payload of the message is left empty. Therefore we send for every message ID a message without payload and with the length field equal to zero. This test aims to find vulnerabilities that do not depend on the payload. The corresponding message structure can be seen in Figure 4.5.

The next two test cases are used to test how the virtual drone handles messages with payload constructed with the boundary values such as minimum and maximum values.

`Test Case 4.` We construct payloads consisting entirely out of the minimum value. This test investigates how the implementation handles the minimum value

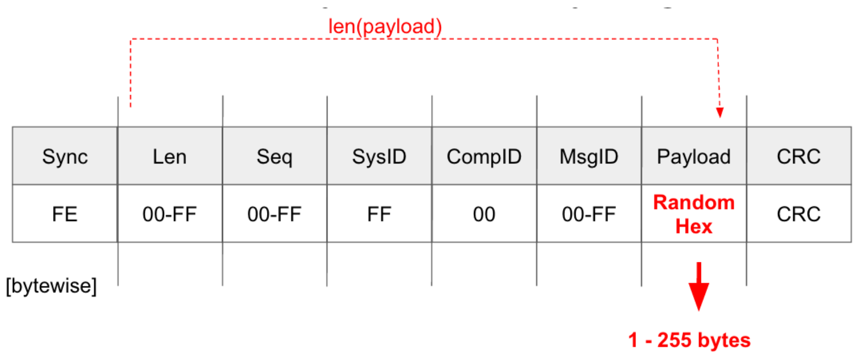

len(payload)
Sync | Len | Seq | SysID | CompID | MsgID | Payload | CRC
FE | 00-FF | 00-FF | FF | 00 | 00-FF | Random Hex | CRC
[bytewise]
1 - 255 bytes

FIGURE 4.4: Message Structure for Test Case 2

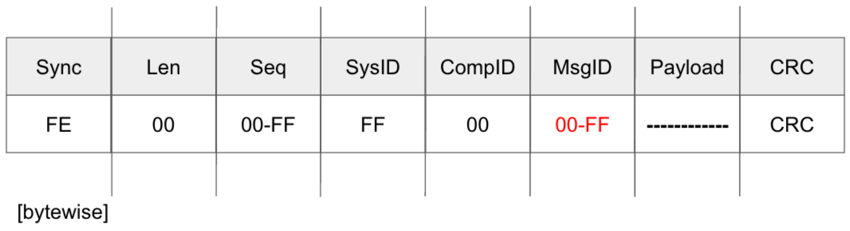

Sync | Len | Seq | SysID | CompID | MsgID | Payload | CRC
FE | 00 | 00-FF | FF | 00 | 00-FF | ------------ | CRC
[bytewise]

FIGURE 4.5: Message Structure for Test Case 3

`"00"`. This is done for payloads with a length ranging from 1 to 255 bytes and for every message ID. The corresponding message structure can be seen in Figure 4.6.

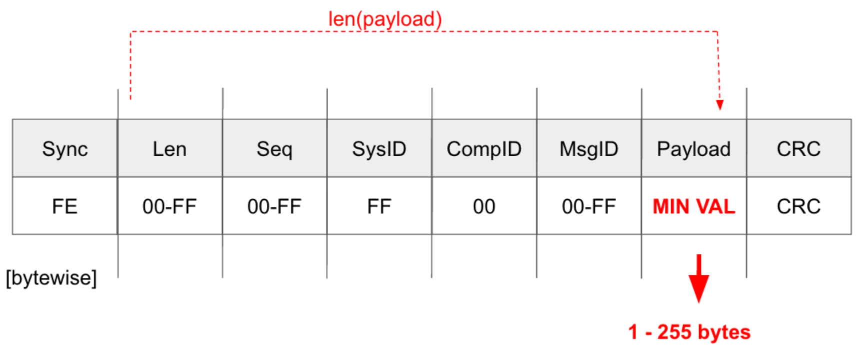

len(payload)
Sync | Len | Seq | SysID | CompID | MsgID | Payload | CRC
FE | 00-FF | 00-FF | FF | 00 | 00-FF | MIN VAL | CRC
[bytewise]
1 - 255 bytes

FIGURE 4.6: Message Structure for Test Case 4

`Test Case 5.` Following the previous test case, we repeat the test for the maximum value `"ff"`. The corresponding message structure can be seen in Figure 4.7.

`Test Case 6.` Within this test case, we replicated the test from test case 2, but we do not send the messages byte per byte. An entire message with all the necessary fields is included in one TCP packet. By sending more data in one packet then the

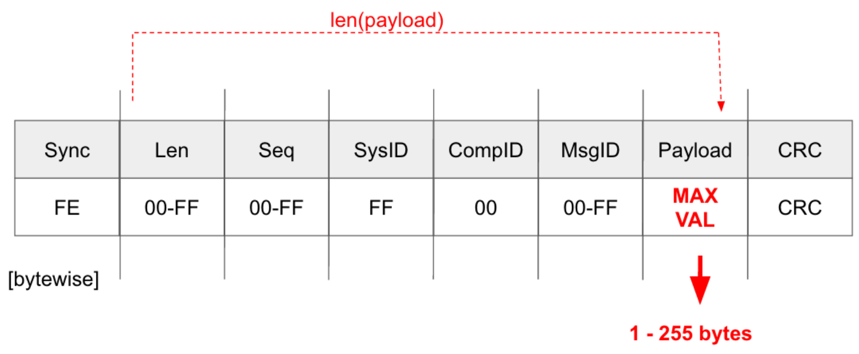


Figure 4.7: Message Structure for Test Case 5

drone expects, we try to find vulnerabilities in how the software decides were a value starts and ends. We increase the length of the random payload from 1 to 255 bytes extending the length of the entire message. The corresponding message structure can be seen in Figure 4.8.

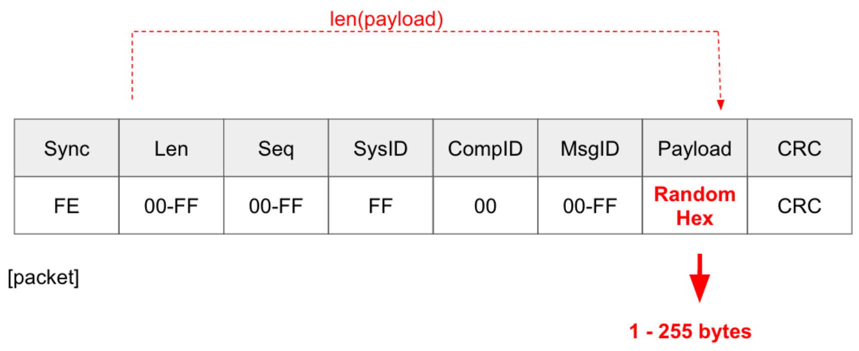


Figure 4.8: Message Structure for Test Case 6

`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. The corresponding message structure can be seen in Figure 4.9.

`Test Case 8` Within this test case, a message is sent a thousand times. The process is repeated for many different messages with different msgIds.

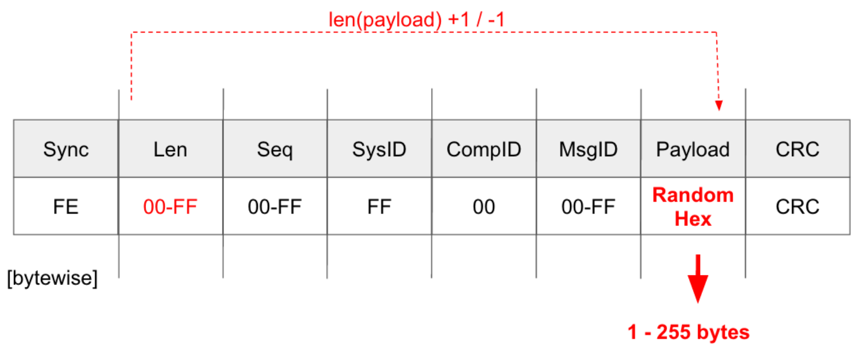


Figure 4.9: Message Structure for Test Case 7

Chapter 5

Results

After running the test cases from section 4.3, the results of these tests will be analysed in this chapter. We start by describing the obtained results followed by a further analysis of the results under the gdb debugger. Based on these results, succeeding case studies are performed and their results will be discussed as well.

5.1 Results

Resulting from the test case 6, in the previous chapter, we were able to identify a possible security flaw. This test case increased the random payload and sent its messages not bytewise, trying to find vulnerabilities in how the software decides were a value starts and ends. The test was able to crash the virtual drone. The floating point exception error caused by the fuzzing script, can be seen in Listing 5.1.

Listing 5.1: Flaoting Point Exception

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

5.2 Analysis Under GDB

By running the virtual drone under the gdb debugger, we were able to further investigate the cause of the floating point exception by using the `backtrace` command. The output of this command can be seen in Appendix C.

By using the `backtrace full` command, which also prints the values of the variables, we obtained more details about the messages that have been sent to the virtual drone. The extra details can be found in Listing 5.2, which contains a sniplet of the output. From these details we learn that the sent message has msgID 27, this is in hexadecimal representation. Converted to decimal representation, the message ID is 39, which is the ID for the MAVLINK_MSG_ID_MISSION_ITEM message. This is an important message since it is used for taking real-time action (e.g. Setting waypoints, Loiter turns, etc.) [20]. This is used in the MAVLink Mission Interface, which is discussed in chapter 3. The message is received by the drone, handled via the `handleMessage()` method, which calls the `handleMission()` method, reading the ID in the payload and performing a switch (case) on the ID, handling the different commands. The method handling this message can be found in Listing 5.3 and gives the extra information that the command will be stored in EEPROM, which means this alters the internal memory of the drone.

Listing 5.2: GDB backtrace full

```
#3  0x0000000000427400 in GCS_MAVLINK::update (this=this@entry=0x6ff090 <copter+5904>, run_cli=...)
  at /home/karel/ardupilot/libraries/GCS_MAVLink/GCS_Common.cpp:869
        c = 119 'w'
        i = <optimized out>
        msg = {checksum = 30546, magic = 254 '\376', len = 182 '\266', seq = 182 '\266',
          sysid = 255 '\377', compid = 0 '\000', msgid = 39 '\'', payload64 = {16318079958426237188,
            8614598618634184389, 2071452411579456358, 14199959831438642112, 6215921153525093120,
            5763815289992474356, 17095648438040612499, 9858078924406871889, 4196118435133347697,
            7432560902066985275, 3852945335020562047, 4657062751962446000, 5369536186669462290,
            2048373184283352483, 11066557593512957704, 3374153009834036582, 3643933501664134640,
            14389727886515427936, 14774576087069922633, 16887653879363676716, 4465987971392911105,
            12219972551944917287, 8598080079541627538, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}
        status = {msg_received = 67 'C', buffer_overrun = 198 '\306', parse_error = 35 '#',
          parse_state = MAVLINK_PARSE_STATE_IDLE, packet_idx = 182 '\266',
          current_rx_seq = 183 '\267', current_tx_seq = 76 'L', packet_rx_success_count = 171,
          packet_rx_drop_count = 0}
        tnow = <optimized out>
        wp_recv_time = <optimized out>
        nbytes = 44
```

Listing 5.3: MAVLink Handle msgID 39

```
// GCS has sent us a command from GCS, store to EEPROM
    case MAVLINK_MSG_ID_MISSION_ITEM:    // MAV ID: 39
    {
        handle_mission_item(msg, mission);
        break;
    }
```

5.3 Succeeding Case Studies

After the previous indication of a vulnerability in the MAVLink Mission Interface, we can further refine our test cases based on the acquired knowledge. We focus the research on the MAVLINK_MSG_ID_MISSION_ITEM message with msgID 39. By doing this, we want to gain more insight in the cause of the exception. For this purpose, a new test case is written which can be found in Listing 5.4. In this test the message ID is kept fixed at 39, the payload is a random combination of hexadecimal values, for every possible length and the message is not sent bytewise.

Listing 5.4: Test Case for msgID 39

```
def create_payload_of_size(size):
    pl=''.join([random.choice('0123456789abcdef') for x
       in range(2*size)])
    return pl

def calculate_length(payload):
    return format(len(payload)/2,'02x')

def construct_string(newSeq,msgId,lengthPayload):
    stx="fe"
    seq=str(format(newSeq,'02x'))
    sysID="ff"
    compID="00"
    msgID=str(format(msgId,'02x'))
    payload=create_payload_of_size(lengthPayload)
    length=str(calculate_length(payload))
    magic=str(format(MAVLINK_MESSAGE_CRCS[int(msgID,16)
       ],'02x'))
    crc=calc_crc_16(length+seq+sysID+compID+msgID+payload
       +magic)
    return(stx+length+seq+sysID+compID+msgID+payload+crc)

def send_mavlink_message39(testNb,initialSeq):
    for i in range(0,1000):  # repeat 1000 times
        seq = initialSeq
        for j in range(0,256): # for each possible length
            of payload: max 255
```

Filename:line	**Times encountered**
AP_Mission.cpp:514	1
AP_Mission.cpp:521	1
AP_Mission.cpp:578	1
AP_Mission.cpp:588	1
AP_Mission.cpp:597	1
AP_Mission.cpp:598	2
AP_Mission.cpp:613	1
AP_Mission.cpp:624	2
AP_Mission.cpp:629	2
AP_Mission.cpp:646	5
AP_Mission.cpp:652	1
AP_Mission.cpp:661	1
AP_Mission.cpp:692	1
AP_Mission.cpp:703	4
AP_Mission.cpp:709	1

Table 5.1: Floating Point Exceptions Encountered

```
            string = construct_msg.construct_string(seq
               ,39,j)
            s.send(string.decode('hex'))
            data = s.recv(4096)
            if seq < 255:
                seq += 1
            else:
                seq = 0
```

We have run this test 25 times, which resulted in a floating point exception every time. In some of the runs the exception occurred at exactly the same place, but we managed to generate 15 unique occurrences of the exception in different places in the source code. The results can be seen in Table 5.1.

In this chapter we discussed the results of the performed case studies. One of the test cases resulted in a floating point exception. Further analysis of the error indicated that the exception resulted from a MAVLINK_MSG_ID_MISSION_ITEM message with msgID 39. Further research focused on this message and was able to generate 15 unique floating point exceptions.

Chapter 6

Discussion of the Results

This chapter discusses the results of the analysis and findings in chapter 5.

In this work we were able to obtain 15 different floating point exceptions, all resulting from the MAVLINK_MSG_ID_MISSION_ITEM message, with msgID 39. These floating point exceptions resulted in crashing the virtual drone.

We encountered 15 unique floating points exceptions, this means they are thrown in different places in the source code. The floating point exception that was encountered most frequent is the one in the AP_Mission.cpp class at line 646, as can be seen in Table 5.1. This is exactly the same exception as we discovered initially. By investigating the source code of AP_Mission.cpp we hope to find the common behaviour between all the cases to find what triggered the exception to be thrown. An overview of all the lines throwing the exception can be seen in Listing 6.1. All the vulnerable lines are part of the mavlink_to_mission_cmd method, converting MAVLink message to an AP_Mission::Mission_Command object which can be stored to EEPROM.

LISTING 6.1: Lines Resulting in Floating Point Exception

```
514: radius_m = fabsf(packet.param3);
521: cmd.p1 = packet.param1;
578: cmd.content.location.lat = packet.param1 * 100;
588: cmd.content.yaw.direction = packet.param3;
597: cmd.content.jump.target = packet.param1;
598: cmd.content.jump.num_times = packet.param2;
613: cmd.p1 = packet.param1;
624: cmd.content.repeat_relay.repeat_count = packet.
   param2;
629: cmd.content.servo.channel = packet.param1;
646: cmd.p1 = packet.param1;
652: cmd.content.digicam_configure.aperture = packet.
   param3;
661: cmd.content.digicam_control.zoom_pos = packet.param2
   ;
692: cmd.content.gripper.action = packet.param2;
```

```
703: cmd.p1 = packet.param1;
709: cmd.content.altitude_wait.wiggle_time = packet.
    param3;
```

By looking at the lines of code resulting in the exception, which can be found in Listing 6.1, we think the exceptions are thrown as a result of a conversion from a floating point number to a data type internally used. All the lines of code, except one, resulting in the exception contain an allocation to a field of a command struct. The allocated value comes from a parameter of the received packet. The observed pattern can be found in Listing 6.2.

Listing 6.2: Pattern Resulting in Floating Point Exception

```
cmd.field = packet.param; // with param equal to param1,
    param2 or param3
```

Chapter 7

Lessons Learned: Problems and Limitations

In this chapter we discuss the problems and limitations that were encountered during the research. We initially did not start by writing a custom fuzzer, but many different alternatives were considered, which will be discussed in this chapter. We start with discussing the problems encountered in the research, followed by the limitations of the research.

7.1 Problems

Our first idea was to use an existing fuzzing platform, and configure this platform to be able to fuzz the MAVLink protocol. Many different platforms and fuzzers including Autofuzz and Sulley were considered. AutoFuzz can automatically understand network protocols and fuzz them to find implementation flaws [24]. The problem with AutoFuzz was that it only works for plain-text protocols, which MAVLink is not. Next we tried a more advanced fuzzing platform called Sulley [25]. Sulley is an automated fuzzing tool that maintains logs and monitors the network. Sulley also monitors the target software and revers the target to a previous state when detecting faults [25]. Although it looked promising, it turned out to have some drawbacks as the Sulley framework is not very user friendly. Another idea was to start from the GCS MAVProxy, and adapt the program to become a fuzzer. Because we wanted a fuzzer that was able to construct valid MAVLink messages independent of using the methods available under the dronecode project, we started creating our own, custom and basic fuzzer.Creating a fuzzer capable of constructing MAVLink messages did not result in any serious problems. Much effort was put into establishing the TCP connection. We first started by using Scapy, a powerful interactive packet manipulation program [26]. Scapy allows to construct packets at the level of TCP/IP. The first step was to perform the three-way handshake. The Scapy code we constructed to establish the connection can be seen in Listing 7.1.

Listing 7.1: Scapy Three-way Handshake

```
from scapy.all import *

sport = random.randint(1024,65535)

ip=IP(src='192.168.56.101', dst='192.168.56.102')

#----- Three-way Handshake Protocol -----#

# 1) SYN
SYN=TCP(sport=sport, dport=5760, flags='S', seq=1000)

# 2) Send SYN and save reply SYNACK
SYNACK=sr1(ip/SYN)

# 3) send ACK reply to SYNACK
ACK=TCP(sport=sport, dport=5760, flags='A', seq=SYNACK.
    ack, ack=SYNACK.seq+1)
send(ip/ACK)
```

After sending the SYN message, the virtual drones replied with a SYN/ACK and after a final ACK from the fuzzer, and the connection is established. Unfortunately, after running the script, the connection was not established. Normally the kernel takes care of sending and receiving network packets. As Scapy does not use the kernel services, the kernel does not know anything about the sent SYN, and sends a RST message to the drone after receiving the SYN/ACK, closing the connection. We solved this problem by using the command found in Listing 7.2.

Listing 7.2: Disable RST Sent By Kernel

```
iptables -A OUTPUT -p tcp -tcp-flags RST RST -s {our IP}
    -j DROP
```

Further problems with Scapy appeared when we wanted to sent a constructed message to the drone. We could not just send the data over TCP, a TCP packet had to be constructed from scratch. This resulted in problems with sequence numbers. Yet another problem with scapy was to keep the connection alive. We did many tests, trying to extend the liveness of the connection, but we were not able to keep the connection alive as long as needed. These problems resulted in using simple sockets to establish the TCP connection.

Another problem is related to the TCP connection, using send and receive buffers on both sides. This problem was discovered while analysing the network traffic in Wireshark. The fuzzer sends messages to the simulator, and the simulator responds by sending an acknowledgement back to the fuzzer. These acknowledgement messages are stored in a receive buffer on the fuzzer side. Since we do not use the received replies, this buffer is never flushed and the receive buffer is getting entirely filled with the received acknowledgements. This resulted in the fuzzer sending `TCP Zero Window` messages to the virtual drone. The window size of a machine is the amount of data the machine can receive during the TCP session, while being able to further process the received data. When a connection is established and the server starts sending data (or replies to sent packets with acknowledgements), the client machine will decrement it's window size as the receive buffer fills. When the data in the receive buffer is processed, the buffer is emptied. When the TCP window size becomes zero because the buffer is full, the client machine will not be able to receive any more data. This results in the client machine sending TCP Zero Windows messages to the server, asking to stop sending data until the previous received data is processed [27]. To tackle this problem, we included one line of code, which can be seen in Listening 7.3. This line is executed right after a message has been sent and it reads from the receive buffer. We do not use the data form the receive buffer so the data is discarded and the buffer is not getting entirely filled.

Listing 7.3: TCP Zero Window Solution

```
    data = s.recv(4096)
```

7.2 Limitations

One of the limitations of our research was already mentioned in section 7.1, the time limit. Because of the limited time we had, some of the ideas had to be dropped. Another limitation applies to the obtained results. We were able to identify some implementation flaws by using messages with random payload. It would be interesting to further try all possible permutations for possible payloads, to have a range of messages or values which all result in a floating point exception. By doing so, we can be able to construct a single message with the ability of crashing the virtual drone. At the moment, our script are not optimized to try all possible combinations (read: brute force), because this consumes too much memory. If encryption or signing is added to the protocol in the (near) future, our script for constructing valid MAVLink messages has to be adapted to cope with the new specifications. We think that after adapting the script, we will still be able to fuzz the protocol. Specifically, we can try to fuzz the implementation of the encryption or signing part, to see whether or not the security measures are supported by a secure implementation.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Because drones are getting more common in today's life, the security of drones is becoming a hot topic for research. Since the communication channel and its lack of (strong) security measures is a painful area, we focused our research on this channel. More specifically, we focused the research on the MAVLink protocol for bi-directional communication between the drone and the GCS. This goal of the developing team of MAVLink is to make MAVLink become a worldwide standard. Previous research on the MAVLink protocol is focused on adding encryption to the protocol specifications. We wanted to contribute to the development community by looking at the same security problem from a different perspective. We believe strong security measures alone is not enough and wanted to find a way to analyse and test the security of the implementation. We used the fuzzing technique to analyse the MAVLink protocol and to find implementation flaws in the protocol.

To conduct our research, we have studies the specifications of the MAVLink protocol in detail, with a focus on the message structure. Based on these specifications we have constructed a fuzzer from scratch, capable of creating valid MAVLink messages. Using our own methods for constructing MAVLink messages made the research independent of the message generating code from the MAVLink project, allowing us to focus only on the handling of the messages by the protocol implementation as used by the drone.

To carry out our experiments, we set up test cases based on the most common attack vectors such as random data and minimum and maximum values. By constructing messages with a correct message format, checksum and random payload, we were able to observe the behaviour of the drone in more detail.

From analyzing the results of our research, we have found that there are software vulnerabilities in the implementation of the protocol. Our test cases resulted in a significant amount of floating point exceptions. The first range of tests resulted in a floating point exception for a message with ID 39. We further focussed our analysis on this message ID and were able to obtain 15 unique floating point exceptions. This gives us a strong indication that more implementation flaws can be found by testing the rest of the msg IDs as thoroughly as this one. From the analysis of the test cases with message ID 39, we found a vulnerable pattern that is used throughout the code. We assume the floating point exception is thrown by trying to convert a floating point number to an internal data type. Our fuzzing script was able to generate this exception within less then a minute, resulting in crashing the virtual drone. However, we have to stress that floating point exceptions are not enabled in real operational drones, only in the simulation. If for example a division by zero takes place, this will result in a value of infinite instead of a floating point exception. Therefore, we do not know whether we can crash a real drone or not. Nevertheless, floating point exceptions are undesired and are usually an indication for an underlying bug or vulnerability. The floating point exceptions indicate that some cases are not tested or handled completely. If it is possible to achieve the same results on a real operational drone, this poses serious security threats.

At the same time we have studied the suitability of the fuzzing technique as a complementing technique for finding vulnerabilities in the protocol. The MAVLink protocol was introduced in 2009 and since many contributors and software testers joined the developing team. As the related work suggest, the MAVLink protocol was extensively tested for vulnerabilities and security shortcomings. The fact that we could find 15 unique floating point exceptions with a basic fuzzer demonstrates that the fuzzing technique is suitable and necessary for a thorough security analysis. Once the script was written to generate valid MAVLink messages, the test case generation was evident and the script was able to generate thousands of test messages within no time. We can conclude that it is worth the effort of using the fuzzing technique as an additional testing method.

8.2 Future Work

Future work can start with completing the entire range of the test cases aiming to gain more results identifying error flaws of the MAVLink software implementation. Since only a fraction of all the possible permutations of random payload is tested, the range can be further extended. However, we have to stress that fuzzing all possible combinations 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. Conducting this research with improved fuzzing scripts which are less memory demanding, could result in a range of values the payload can take, to crash the virtual drone. If more successful cases are known, this could be reduced to a single attack vector able to crash the virtual drone.

Improving the fuzzing script can be done by using more sophisticated fuzzing platforms. We briefly discussed some of them in section 7.1. Future research can focus on configuring one or more of these platform to analyse the MAVLink protocol. The techniques used by these platforms are more advanced and will certainly result in more vulnerabilities to be found.

The research can be repeated on a real drone. Since floating point exceptions are not enabled on a real operational drone, it would be interesting to observe the results of the test cases generating these exceptions on the real drone. As the research indicated, it is possible that not all cases are covered and some vulnerabilities still exists, allowing a floating point exception to be thrown while being operational, crashing the drone.

Appendices

Appendix A

Case Studies Python Code

Listing A.1: Generate Payload

```python

# To calculate the seed
MAVLINK_MESSAGE_CRCS = [50, 124, 137, 0, 237, 217, 104,
   119, 0, 0, 0, 89, 0, 0, 0, 0, 0, 0, 0, 0, 214, 159,
   220, 168, 24, 23, 170, 144, 67, 115, 39, 246, 185,
   104, 237, 244, 222, 212, 9, 254, 230, 28, 28, 132,
   221, 232, 11, 153, 41, 39, 214, 223, 141, 33, 15, 3,
   100, 24, 239, 238, 0, 0, 183, 0, 130, 0, 148, 21, 0,
   52, 124, 0, 0, 0, 20, 0, 152, 143, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 183, 63, 54, 0, 0, 0, 0, 0, 0, 0,
   19, 102, 158, 208, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   134, 219, 208, 188, 84, 22, 19, 21, 134, 0, 78, 68,
   189, 127, 42, 21, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   204, 49, 170, 44, 83, 46, 247]


def generate_payload(x,l):
  if x == 1:
    result = generate_random_payload(l)
    return result
  elif x == 3:
    return ["00"]*l
  elif x == 4:
    return ["ff"]*l
  else:
    print('else case')
    return []

def generate_random_payload(l):
  a=[]
  for x in range(0,l):
    a.append(''.join([random.choice('0123456789abcdef')
        for x in range(2)]))
  return a
```

Listing A.2: Test Case 2

```
def case2(iteration): # Random payload
  seq = 0
  for m in range(0,256):
    stx = "fe"
    sysID = "ff"
    compID = "00"
    msgID = str(format(m,'02x'))
    magic=str(format(MAVLINK_MESSAGE_CRCS[m],'02x'))
    for l in range(1,256): #lengths of payload
      start = time.time()
      f = open('tests/random_payload/msgID_'+str(m)+'
          _length_'+str(l)+'_it_'+str(iteration), 'w')
      f.write('Start Time: '+str(start)+'\n')
      a=[]
      newSeq =str(format(seq,'02x'))
      a.append(stx)
      payload=[]
      payload = generate_payload(1,l)
      length = str(format(len(payload),'02x'))
      a.append(length)
      a.append(newSeq)
      a.append(sysID)
      a.append(compID)
      a.extend(payload)
      pl=''.join(payload)
      crc = calc_crc_16(length+newSeq+sysID+compID+msgID+
          pl+magic)
      a.extend(crc)
      f.write('string: ' + str(a) +'\n')
      f.close()
      for y in range(0,len(a)):
        s.send(a[y].decode('hex'))
        data = s.recv(4096)
    if seq < 255:
      seq += 1
    else:
      seq = 0
```

Listing A.3: Test Case 3

```
def case3(): # No payload
  seq = 0
  for m in range(0,256):
    stx = "fe"
    sysID = "ff"
    compID = "00"
    msgID = str(format(m,'02x'))
    magic=str(format(MAVLINK_MESSAGE_CRCS[m],'02x'))
    start = time.time()
    f = open('tests/zero_payload/msgID_'+str(m)+'_length_
       '+'0', 'w')
    f.write('Start Time: '+str(start)+'\n')
    a=[]
    newSeq =str(format(seq,'02x'))
    a.append(stx)
    payload = ""
    length = "00"
    a.append(length)
    a.append(newSeq)
    a.append(sysID)
    a.append(compID)
    crc = calc_crc_16(length+newSeq+sysID+compID+msgID+
       magic)
    a.extend(crc)
    f.write('string: ' + str(a) +'\n')
    for y in range(0,len(a)):
      s.send(a[y].decode('hex'))
      data = s.recv(4096)
    end = time.time()
      f.write('End Time: ' +str(end) +'\n')
      f.write('Completed in: ' + str(end-start))
  if seq < 255:
    seq += 1
  else:
    seq = 0
```

Listing A.4: Test Case 4

```
def case4(): # all zero payload
 seq = 0
 for m in range(0,256):
   stx = "fe"
   sysID = "ff"
   compID = "00"
   msgID = str(format(m,'02x'))
   magic=str(format(MAVLINK_MESSAGE_CRCS[m],'02x'))
   for l in range(1,256): #lengths of payload
     start = time.time()
     f = open('tests/all_zero/msgID_'+str(m)+'_length_'+
        str(l), 'w')
     f.write('Start Time: '+str(start)+'\n')
     a=[]
     newSeq =str(format(seq,'02x'))
     a.append(stx)
     payload = generate_payload(3,l)
     length = str(format(len(payload),'02x'))
     a.append(length)
     a.append(newSeq)
     a.append(sysID)
     a.append(compID)
     a.extend(payload)
     pl=''.join(payload)
     crc = calc_crc_16(length+newSeq+sysID+compID+msgID+
        pl+magic)
     a.extend(crc)
     f.write('string: ' + str(a) +'\n')
     for y in range(0,len(a)):
       s.send(a[y].decode('hex'))
       data = s.recv(4096)
     end = time.time()
         f.write('End Time: ' +str(end) +'\n')
         f.write('Completed in: ' + str(end-start))
   if seq < 255:
     seq += 1
   else:
     seq = 0
```

Listing A.5: Test Case 5

```
def case5(): # all max payload
 seq = 0
 for m in range(0,256):
   stx = "fe"
```

```
    sysID = "ff"
    compID = "00"
    msgID = str(format(m,'02x'))
    magic=str(format(MAVLINK_MESSAGE_CRCS[m],'02x'))
    for l in range(1,256): #lengths of payload
      start = time.time()
      f = open('tests/all_max/msgID_'+str(m)+'_length_'+
         str(l), 'w')
      f.write('Start Time: '+str(start)+'\n')
      a=[]
      newSeq =str(format(seq,'02x'))
      a.append(stx)
      payload = generate_payload(4,l)
      length = str(format(len(payload),'02x'))
      a.append(length)
      a.append(newSeq)
      a.append(sysID)
      a.append(compID)
      a.extend(payload)
      pl=''.join(payload)
      crc = calc_crc_16(length+newSeq+sysID+compID+msgID+
         pl+magic)
      a.extend(crc)
      f.write('string: ' + str(a) +'\n')
      for y in range(0,len(a)):
        s.send(a[y].decode('hex'))
        data = s.recv(4096)
      end = time.time()
          f.write('End Time: ' +str(end) +'\n')
          f.write('Completed in: ' + str(end-start))
    if seq < 255:
      seq += 1
    else:
      seq = 0
```

Listing A.6: Test Case 6

```
def case6(iteration): # Sent as a packet
    for i in range(0,256):
        seq = 0
        for j in range(0,256):
          stx="fe"
          newSeq=str(format(seq,'02x'))
          sysID="ff"
          compID="00"
          msgID=str(format(msgId,'02x'))
          payload=create_payload_of_size(lengthPayload)
          length=str(calculate_length(payload))
          magic=str(format(MAVLINK_MESSAGE_CRCS[int(msgID,16)],'02x'))
          crc=calc_crc_16(length+seq+sysID+compID+msgID+payload+magic)
          string = (stx+length+seq+sysID+compID+msgID+payload+crc)
          s.send(string.decode('hex'))
          data = s.recv(4096)
       if seq < 255:
        seq += 1
       else:
        seq = 0
```

Listing A.7: Test Case 7

```
def case2(iteration): # Length vs actual length payload
  seq = 0
  for m in range(0,256):
    stx = "fe"
    sysID = "ff"
    compID = "00"
    msgID = str(format(m,'02x'))
    magic=str(format(MAVLINK_MESSAGE_CRCS[m],'02x'))
    for l in range(1,256): #lengths of payload
      start = time.time()
      f = open('tests/random_payload/msgID_'+str(m)+'_length_'+str(l)+'_it_'+str(iteration), 'w')
      f.write('Start Time: '+str(start)+'\n')
      a=[]
      newSeq =str(format(seq,'02x'))
      a.append(stx)
      payload=[]
      payload = generate_payload(1,l)
      length = str(format((len(payload)+1),'02x')) # or -1
      a.append(length)
      a.append(newSeq)
      a.append(sysID)
      a.append(compID)
      a.extend(payload)
      pl=''.join(payload)
      crc = calc_crc_16(length+newSeq+sysID+compID+msgID+
         pl+magic)
      a.extend(crc)
      f.write('string: ' + str(a) +'\n')
      f.close()
      for y in range(0,len(a)):
        s.send(a[y].decode('hex'))
        data = s.recv(4096)
    if seq < 255:
      seq += 1
    else:
      seq = 0
```

Appendix B

Autopilots systems, Software and Projects using MAVLink

B.1 Autopilot Systems Using MAVLink

- ArduPilotMega (main protocol)
- pxIMU Autopilot (main protocol)
- SLUGS Autopilot (main protocol)
- FLEXIPILOT (optional protocol)
- UAVDevBoard/Gentlenav/MatrixPilot (optional protocol)
- SenseSoar Autopilot (main protocol)
- SmartAP Autopilot (main protocol)
- AutoQuad 6 AutoPilot (main protocol)

B.2 Software Packages Using MAVLink

- iDroneCtrl (iOS)
- QGroundControl (Windows/Mac/Linux)
- HK Ground Control Station (Windows)
- APM Planner (Windows/Mac)
- QGroundControl w/ AutoQuad MainWidget (Windows/Mac/Linux)
- Copter GCS (Android)
- AutoQuad GCS (Android)

- ROS to MAVLink bridge: https://github.com/mavlink/mavlink-ros
- MAVCONN Lightweight Aerial Middleware http://pixhawk.ethz.ch/software/middleware/start
- oooArk / MAVSim http://www.youtube.com/watch?v=-wQVrM5SL2o&fe
- MAVLink python bindings
- MAVProxy (allows to connect multiple UDP/serial links, including flightgear): git://git.samba.org/tridge/UAV/MAVProxy.git

B.3 Projects Using MAVLink

- ArduPilotMega http://code.google.com/p/ardupilot-mega/
- MatrixPilot UAV DevBoard http://code.google.com/p/gentlenav/
- PIXHAWK http://pixhawk.ethz.ch/
- ETH Flying Machine Arena http://www.idsc.ethz.ch/Research_DAndrea/FMA
- ETH SenseSoar Solar Airplane Project http://www.sensesoar.ethz.ch/doku.php?id=news
- ETH Skye Blimp Project http://www.projectskye.ch/
- UC Santa Cruz SLUGS http://slugsuav.soe.ucsc.edu/index.html
- ArduCAM OSD http://code.google.com/p/arducam-osd/
- Sky-Drones - UAV Flight Control Systems http://www.sky-drones.com/
- AutoQuad - Autonomous Multirotor Vehicle controller http://autoquad.org/

B.4 MAVLink Examples

Listing B.1: Heartbeat Message XML Definition

```xml
<message id="0" name="HEARTBEAT">
  <description>The heartbeat message shows that a system
     is present and responding. The type of the MAV and
     Autopilot hardware allow the receiving system to
     treat further messages from this system appropriate
     (e.g. by laying out the user interface based on the
     autopilot).</description>
  <field type="uint8_t" name="type">Type of the MAV (quadrotor, helicopter, etc., up to 15 types, defined in MAV_TYPE ENUM)</field>
  <field type="uint8_t" name="autopilot">Autopilot type / class. defined in MAV_CLASS ENUM</field>
  <field type="uint8_t" name="base_mode">System mode bitfield, see MAV_MODE_FLAGS ENUM in mavlink/include/mavlink_types.h</field>
  <field type="uint32_t" name="custom_mode">Navigation mode bitfield, see MAV_AUTOPILOT_CUSTOM_MODE ENUM for some examples. This field is autopilot-specific.</field>
  <field type="uint8_t" name="system_status">System status flag, see MAV_STATUS ENUM</field>
  <field type="uint8_t_mavlink_version" name="mavlink_version">MAVLink version</field>
</message>
```

Listing B.2: Heartbeat Message C-Struct Definition

```
#define MAVLINK_MSG_ID_HEARTBEAT 0

typedef struct __mavlink_heartbeat_t
{
 uint32_t custom_mode; ///< Navigation mode bitfield, see MAV_AUTOPILOT_CUSTOM_MODE ENUM for some examples. This field is autopilot-specific.
 uint8_t type; ///< Type of the MAV (quadrotor, helicopter, etc., up to 15 types, defined in MAV_TYPE ENUM)
 uint8_t autopilot; ///< Autopilot type / class. defined in MAV_CLASS ENUM
 uint8_t base_mode; ///< System mode bitfield, see MAV_MODE_FLAGS ENUM in mavlink/include/mavlink_types.h
 uint8_t system_status; ///< System status flag, see MAV_STATUS ENUM
 uint8_t mavlink_version; ///< MAVLink version
} mavlink_heartbeat_t;
```

Listing B.3: Heartbeat Pack and Unpack Functions Definition

```
/**
 * @brief Pack a heartbeat message
 * @param system_id ID of this system
 * @param component_id ID of this component (e.g. 200 for
     IMU)
 * @param msg The MAVLink message to compress the data
    into
 *
 * @param type Type of the MAV (quadrotor, helicopter,
    etc., up to 15 types, defined in MAV_TYPE ENUM)
 * @param autopilot Autopilot type / class. defined in
    MAV_CLASS ENUM
 * @param base_mode System mode bitfield, see
    MAV_MODE_FLAGS ENUM in mavlink/include/mavlink_types.
    h
 * @param custom_mode Navigation mode bitfield, see
    MAV_AUTOPILOT_CUSTOM_MODE ENUM for some examples.
    This field is autopilot-specific.
 * @param system_status System status flag, see
    MAV_STATUS ENUM
 * @return length of the message in bytes (excluding
    serial stream start sign)
 */
static inline uint16_t mavlink_msg_heartbeat_pack(uint8_t
    system_id, uint8_t component_id, mavlink_message_t*
   msg,
                         uint8_t type, uint8_t autopilot,
                            uint8_t base_mode, uint32_t
                            custom_mode, uint8_t system_status)


/**
 * @brief Encode a heartbeat struct into a message
 *
 * @param system_id ID of this system
 * @param component_id ID of this component (e.g. 200 for
     IMU)
 * @param msg The MAVLink message to compress the data
    into
 * @param heartbeat C-struct to read the message contents
     from
 */
static inline uint16_t mavlink_msg_heartbeat_encode(
   uint8_t system_id, uint8_t component_id,
   mavlink_message_t* msg, const mavlink_heartbeat_t*
   heartbeat)

/**
```

```
 * @brief Decode a heartbeat message into a struct
 *
 * @param msg The message to decode
 * @param heartbeat C-struct to decode the message
     contents into
 */
static inline void mavlink_msg_heartbeat_decode(const
    mavlink_message_t* msg, mavlink_heartbeat_t* heartbeat
    )
```

Appendix C

Analysis of the Results

Listing C.1: GDB Output

```
Program received signal SIGFPE, Arithmetic exception.
0x00000000004a3b84 in AP_Mission::mavlink_to_mission_cmd
   (packet=..., cmd=...)
  at /home/karel/ardupilot/libraries/AP_Mission/
     AP_Mission.cpp:646
646         cmd.p1 = packet.param1;
                               // 0 = no roi, 1 = next waypoint
   , 2 = waypoint number, 3 = fixed location, 4 = given
   target (not supported)
(gdb) backtrace
#0  0x00000000004a3b84 in AP_Mission::
   mavlink_to_mission_cmd (packet=..., cmd=...)
  at /home/karel/ardupilot/libraries/AP_Mission/
     AP_Mission.cpp:646
#1  0x0000000000426ffa in GCS_MAVLINK::
   handle_mission_item (this=0x6ff090 <copter+5904>,
  msg=0x7fffffffdb00, mission=...) at /home/karel/
     ardupilot/libraries/GCS_MAVLink/GCS_Common.cpp:667
#2  0x0000000000417a72 in GCS_MAVLINK::handleMessage (
   this=this@entry=0x6ff090 <copter+5904>,
  msg=msg@entry=0x7fffffffdb00) at GCS_Mavlink.cpp:1050
#3  0x0000000000427400 in GCS_MAVLINK::update (this=
   this@entry=0x6ff090 <copter+5904>, run_cli=...)
  at /home/karel/ardupilot/libraries/GCS_MAVLink/
     GCS_Common.cpp:869
#4  0x00000000004166cf in Copter::gcs_check_input (this=0
   x6fd980 <copter>) at GCS_Mavlink.cpp:2022
#5  0x00000000004b9a8d in operator() (this=0x7fffffffdca0
   )
  at /home/karel/ardupilot/libraries/AP_HAL/utility/
     functor.h:55
#6  AP_Scheduler::run (this=0x6fdcb8 <copter+824>,
   time_available=2500)
  at /home/karel/ardupilot/libraries/AP_Scheduler/
     AP_Scheduler.cpp:92
#7  0x00000000004bfab9 in HAL_SITL::run (this=0x701300 <
   AP_HAL::get_HAL()::hal>, argc=<optimized out>,
  argv=<optimized out>, callbacks=0x6fd980 <copter>)
  at /home/karel/ardupilot/libraries/AP_HAL_SITL/
     HAL_SITL_Class.cpp:89
#8  0x0000000000402c8e in main (argc=<optimized out>,
   argv=<optimized out>) at ArduCopter.cpp:650
```

The output in Listing C.1 gives the execution trace of the methods during execution and need to be read bottom up.

Appendix D

Scientific Paper - Dutch

Veiligheidsanalyse van het Drone Communicatieprotocol: Fuzzing Toegepast op het MAVLink Protocol

Karel Domin
karel.domin@student.kuleuven.be
KU Leuven

***Samenvatting*—In de huidige samenleving zijn drones alomtegenwoordig geworden. Ze worden zowel door hobbyisten gebruikt als door professionelen voor het ondersteunen van essentiële diensten. Desondanks het feit dat drones zeer veel voordelen te bieden hebben, kunnen ze ook enorme veiligheidsbedreigingen inhouden. Verschillende aanvallen tegen drones zijn naar voren gekomen de afgelopen jaren. Tegelijkertijd is er ook zeer veel onderzoek verricht om drone gebruik veiliger te maken. een belangrijk onderdeel van dit onderzoek is gericht op het draadloze communicatiekanaal tussen de drone en het controlestation, dat kwetsbaar is door het ontbreken van (sterke) beveiligingsmechanismen. Het onderzoek richt zich voornamelijk op het aanbieden van de nodige veiligheidsmechanismen voor dit kanaal, zoals het gebruik van encryptie. Daarentegen zijn wij van mening dat de beveiliging van het kanaal op zich niet voldoende is. We hebben ook nood aan een veilige implementatie van deze mechanismen. Ons onderzoek richt zich op de veiligheid van het MAVLink protocol omdat dit een wereldwijde standaard tracht te bekomen. Het doel van ons onderzoek is om een veiligheidsanalyse uit te voeren voor dit protocol en mogelijke ontwerp of implementatie gebreken aan te tonen. Hiervoor maken we gebruik van fuzzing technieken. We hebben een fuzzer ontwikkeld die in staat is om willekeurige en semi geldige berichten te genereren die geïnjecteerd kunnen worden in het protocol. De testgevallen die uitgevoerd werden resulteerden in het bekomen van een floating point exception op een virtuele drone waardoor we deze konden doen crashen. Na analyse van de fout werden verdere, meer verfijnde, testgevallen opgesteld. Dit resulteerde in de identificatie van 15 kwetsbaarheden in de implementatie waar de floating point exceptions bekomen kunnen worden. Verder onderzoek wees uit dat het versturen van een specifiek bericht type met willekeurige data een kwetsbaarheid vormde voor de virtuele drone. Vervolgens waren we in staat om een bepaald patroon te identificeren in de implementatie dat kwetsbaar was voor deze aanval.**

I. Introductie

A. Motivatie

Onbemande Luchtvaartuigen, beter bekend als drones, zijn luchtvoertuigen die vanop afstand bestuurd worden vanuit een controlestation of die volledig autonoom kunnen vliegen aan de hand van een voorgeprogrammeerde missie. Op basis van hun toepassing zijn er twee verschillende categorieën te onderscheiden, militaire drones en commerciële drones. Doorheen deze paper richten we ons op commerciële drones.

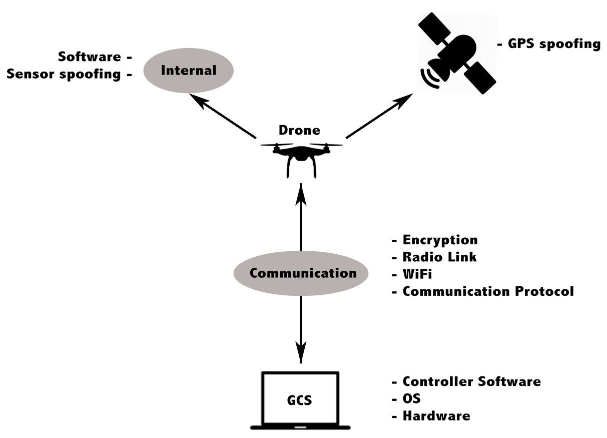
Tegenwoordig worden drones niet enkel gebruikt door hobbyisten, maar ondersteunen zij overigens essentiële diensten. Drones worden ingezet voor de detectie van bosbranden en illegale jacht, reddingsoperaties en zelfs voor het aanleveren van medicijnen. De toepassingsgebieden van drones zijn aan het verruimen omdat zij eenvoudig en snel gebieden kunnen bereiken die onbereikbaar zijn voor mensen. Om al deze taken te kunnen uitvoeren zijn drones uitgerust met een GPS systeem, camera en audio module. Verder hebben drones een radio die draadloze communicatie toelaat met een afstandsbediening of controlestation op de grond. Ondanks de vele voordelen die het gebruik van drones biedt, kunnen ze ook belangrijke veiligheid en privacy bedreigingen met zich meebrengen. Dit komt voornamelijk doordat het draadloze communicatiekanaal een doel vormt voor verschillende types aanvallen vanop afstand. Zo kunnen kwaadwilligen proberen om gevoelige of geheime informatie te verkrijgen door het communicatiekanaal af te luisteren, kwaadaardige commando's door te sturen of de software van de drone aan te passen over ftp. Als een drone overgenomen kan worden, kan deze omgevormd worden tot een projectiel, wat een enorme bedreiging kan betekenen voor omstaanders.

B. Gerelateerd Werk

Figuur 1 toont een overzicht van een drone opstelling met de mogelijke bijhorende kwetsbaarheden. Deze opstelling omvat de drone, een GPS module en een controlestation of afstandsbediening. Deze hoofdcomponenten vormen de basis voor kwetsbaarheden. Zo kan men bijvoorbeeld de software van de drone aanpassen, kwetsbaarheden vinden in de software of de toestandsmachine van de drone uitbuiten. Het communicatiekanaal tussen de GPS module van de drone en de GPS satelliet maakt de drone kwetsbaar voor GPS jamming en spoofing aanvallen. Het communicatiekanaal tussen de drone en het controlestation daarentegen, is kwetsbaar op verschillende manieren en laat o.a. afluisteren en replay aanvallen toe.

Een GPS spoofing attack probeert de GPS ontvanger van de drone te misleiden door valse GPS signalen uit te zenden, die zich voordoen als legitieme signalen uitgezonden door een GPS satelliet. Deze aanval kan ertoe leiden dat elk systeem gebruik makend van GPS signalen van koers verandert of denkt zich op een andere locatie te bevinden dan de werkelijke [11]. Tippenhauer et. al (2011) hebben grondig onderzocht wat de vereisten zijn voor een succesvolle GPS spoofing aanval [24].

Ander onderzoek legt de focus op het communicatiekanaal tussen de drone en het controlestation. Het voornaamste probleem met dit kanaal is dat het gebruikt wordt zonder enige



Figuur 1. Drone kwetsbaarheden

vorm van encryptie of met een te zwakke vorm van encryptie die gekraakt kan worden. In 2009 slaagden militanten in Irak er in om de beelden te onderscheppen van de militaire U.S. Predator drones. Dit was mogelijk doordat de communicatie tussen de drone en het controlestation niet geëncrypteerd was. Encryptie bood een eenvoudige oplossing voor dit probleem. Echter werd er besloten geen encryptie te gebruiken vanwege het te hoge kostenplaatje [25].

Een interessant communicatie protocol is het MAVLink protocol dat gebruikt kan worden voor bidirectionele communicatie tussen de drone en het controlestation en dat een wereldwijde standaard tracht te bekomen. Het MAVLink protocol biedt echter geen veiligheidsmaatregelen, buiten een controlesom om corrupte paketten te ontdekken. J. Marty introduceerde in zijn werk over de kwetsbaarheden van het MAVLink protocol een methode om de kost om het MAVLink protocol te beveiligen te kwantificeren door het meten van de netwerk vertraging, het energieverbruik, en de kans op slagen van een aanval [4]. Het onderzoek van Thomas M. Duboisson introduceerde een inkapseling formaat dat gebruikt kan worden in combinatie met MAVLink om te beveiligen tegen bericht manipulatie, afluisteren en replay aanvallen. Dit kan bekomen worden door gebruik te maken van cryptografische methoden, die 16 bijkomende bytes toevoegen aan elk bericht [12]. Ook N. Butcher et. al bestudeerden mogelijke draadloze aanvallen op MAVLink en stelden het RC5 encryptie algoritme voor als een tegenmaatregel voor deze aanvallen. Tegelijkertijd onderzochten ze de impact van encryptie op de performance van de drone [5].

Toevoegen van encryptie aan het MAVLink protocol zou het protocol zeker bestend maken tegen bepaalde aanvallen. Tot op de dag van vandaag is encryptie nog niet toegevoegd aan de officiële distributie van het MAVLink protocol. Een van de redenen is dat encrypteren van MAVLink berichten niet rechtstreeks mogelijk is omdat berichten geïdentificeerd worden door een waarde in de berichthoofding. Indien men deze hoofding zou encrypteren, dan kan de drone het bericht niet meer onderscheiden van ruis [5].

Evenwel is het beveiligen van het draadloos communicatiekanaal op zich niet voldoende. In ons onderzoek willen we kijken naar dezelfde kwetsbaarheden vanuit een ander standpunt. We hebben nood aan een combinatie van sterke veiligheidsmaatregelen en tegelijkertijd een veilige implementatie.

C. Onderzoeksdoelen, Onderzoeksvragen en Onze Aanpak

Het onderzoeksdoel van deze paper is om een veiligheidsanalyse uit te voeren voor het MAVLink protocol en mogelijke gebreken in het ontwerp of de implementatie te vinden. Hiervoor beroepen we ons op fuzzing. Het doel van fuzzing is om ongeldige of semi-ongeldige data te injecteren om vervolgens het geteste systeem te observeren op abnormaal gedrag of crashes. Voor zover wij weten is de fuzzing techniek nog niet toegepast op een analyse van het MAVLink protocol. Tegelijkertijd trachten we de geschiktheid van deze techniek te bestuderen als een aanvulling op de reeds bestaande technieken om mogelijke kwetsbaarheden te ontdekken in het protocol. Voor ons onderzoek hebben we de specificaties van het protocol en de structuur van de berichten grondig bestudeerd en op basis hiervan een fuzzer geconstrueerd. De fuzzer is in staat om automatisch geldige MAVLink berichten aan te maken. We formuleren drie onderzoeksvragen die we meer in detail willen verkennen doorheen het onderzoek. Deze omvatten: (i) Hoe kunnen we softwarematige veiligheidsgebreken identificeren in het MAVLink framework?, (ii) Is de fuzzing techniek geschikt om gebreken in de implementatie van het protocol te ontdekken?, (iii) Wat zijn de gevolgen van de uitbuiting van de gevonden kwetsbaarheden?

II. Achtergrond Informatie

A. Software In The Loop

Software In The Loop (SITL), biedt simulators aan voor de ArduCopter, ArduPlane en ArduRover van het DroneCode project. Het gebruik van simulators laat toe om het gedrag van deze voertuigen te bestuderen zonder de eigenlijke hardware te gebruiken. De simulators kunnen op verschillende platformen gebruikt worden, o.a. Linux en Windows [7].

B. MAVLink

Het Micro Air Vehicle Communication Protocol laat instanties toe om te communiceren over een draadloos communicatiekanaal. In combinatie met drones, ondersteunt het de communicatie tussen de drone en het controlestation. Het controlestation stuurt commando's en controlesignalen naar de drone en de drone stuurt telemetrische informatie terug. MAVLink werd geïntroduceerd in 2009 door Lorenz Meier onder de GNU Lesser General Public License en is momenteel onderdeel van het DroneCode project beheerst door de Linux Foundation [20]. Het DroneCode project heeft duizenden ontwikkelaars en meer dan honderdduizend gebruikers. Een bericht in het MAVLink protocol wordt byte per byte verstuurd over het communicatiekanaal, gevolgd door een controlesom. Indien de som niet correct is, is het bericht corrupt en zal het verworpen worden. Figuur 2 toont de structuur van een

MAVLink bericht. De minimum lengte van een bericht is 8 bytes en de maximum lengte, met volledige payload, bedraagt 263 bytes.

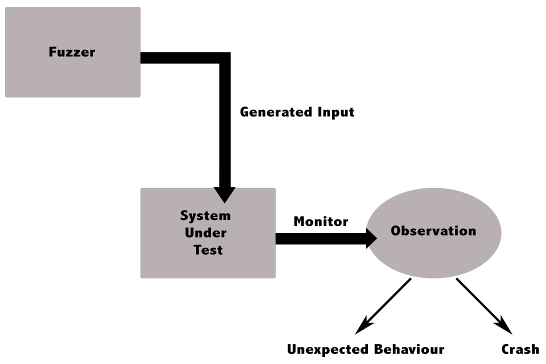
$$\underbrace{\boxed{\texttt{Sync}}}_{1\,byte}\underbrace{\boxed{\texttt{Len}}}_{1\,byte}\underbrace{\boxed{\texttt{Seq}}}_{1\,byte}\underbrace{\boxed{\texttt{Sys}}}_{1\,byte}\underbrace{\boxed{\texttt{Comp}}}_{1\,byte}\underbrace{\boxed{\texttt{MsgID}}}_{1\,byte}\underbrace{\boxed{\texttt{Payload}}}_{x\,byte}\underbrace{\boxed{\texttt{CRC}}}_{2\,byte}$$

Figuur 2. MAVLink Message Structure

We geven een bondige beschrijving van de verschillende velden in het bericht. Het *Sync* veld duidt het begin van een nieuw bericht aan. Het *Length* veld geeft de lengte van de payload aan. Een *Seq* of sequence number is inbegrepen om pakketverlies te detecteren. Het *SystemID* en *ComponentID* worden gebruikt om respectievelijk het verzendende systeem of component aan te duiden. Het *MessageID* bepaalt het type bericht en geeft aan hoe de payload afgehandeld moet worden. De *Payload* bevat de werkelijke data van het bericht. Op het einde van het bericht volgt een *CRC* als controlesom voor validatie.

C. Fuzzen

Fuzzen is een techniek om kwetsbaarheden en bugs in software te vinden door ongeldige of semi ongeldige data te injecteren. De te injecteren data kunnen minimum en maximum waarden bevatten, maar ook ongeldige en compleet willekeurige waarden of karakters. Volgend op het injecteren van de data kan het systeem geobserveerd worden om eventueel onverwacht gedrag waar te nemen zoals een software crash. Figuur 3 illustreert de werking van de techniek. In fuzzen kan men drie hoofdcategorieën onderscheiden: Plain Fuzzen, Protocol Fuzzen en Toestandsmachine Fuzzen [15], [16].



Figuur 3. Fuzzing Techniek

Plain fuzzen is de meest simpele manier van fuzzen. De input data kan bestaan uit volledig willekeurige data of wordt bekomen door geldige data die men heeft kunnen monitoren aan te passen [14].

Protocol fuzzen is een gespecialiseerdere manier van testen. De input wordt gegenereerd op basis van de protocol specificaties zoals het berichtenformaat of pakketformaat en de afhankelijkheid tussen de verschillende velden in het bericht. Dit noemt men slimme generatie en is de mogelijkheid om geldige of semi-geldige berichten te construeren. Dit kan noodzakelijk zijn indien de input voor een programma of protocol een bepaalde structuur moet hebben die betekenis heeft voor het programma of protocol. Het grote voordeel van deze techniek is dat hoe meer intelligentie gebruikt wordt, des te dieper men kan gaan in de software [3], [17].

State-based fuzzen is een fuzzing techniek waarbij men tracht om de toestandsmachine van de software te fuzzen. Dit kan bekomen worden door de volgorde van verzonden berichten, zoals ze doorgaans gebruikt worden in de software, te doorbreken. De impact van deze methode hangt af van de toestanden die men kan overslaan [3].

De meest gangbare manier van fuzzen is om te starten met een basis fuzzer die geen intelligentie over de te fuzzen software of protocol gebruikt en de hoeveelheid gebruikte intelligentie te verhogen indien nodig [18]. De fuzzing techniek heeft de mogelijkheid om gebreken te vinden in de afhandeling van fouten, clean-up code of in de toestandsmachine. Een goede fuzzer moet volgende taken kunnen uitvoeren:

1) Testgevallen opstellen
2) Opslaan of loggen van de testgevallen voor reproductie
3) De testgevallen doorsturen naar het systeem onder test
4) Het gedrag van het systeem onder test observeren en crashes detecteren.

III. Methode

In de realiteit bestaat een opstelling uit een drone en een controlestation. Ons onderzoek voeren op een echte drone is niet haalbaar. Dit zou betekenen dat we rekening moeten houden met de levensduur van de batterij, zorgen dat we geen risico's vormen voor omstaanders en uiteraard zien dat we de drone niet onherstelbaar beschadigen. We hebben nood aan een simulator voor de drone en SITL zoals voorgaand besproken, biedt deze mogelijkheid.

A. Onderzoeks Opstelling

De SITL simulator wordt uitgevoerd op een Linux virtuele machine met Ubuntu 14.04 TLS. We gebruiken de simulator voor de ArduCopter. De simulator kan opgestart worden aan de hand van het commando in Listing 1.

Listing 1. Run SITL

```
$ ./ArduCopter.elf --home -35,149,584,270
    --model quad --speedup 100 --wipe
Started model quad at -35,149,584,270 at
   speed 1.0
Starting sketch 'ArduCopter'
Starting SITL input
bind port 5760 for 0
Serial port 0 on TCP port 5760
Waiting for connection ....
```

De output van het commando toont aan hoe we kunnen verbinden met SITL door een TCP verbinding te bewerkstelligen naar het ip adres van de virtuele drone op poort 5760. In een normale setup gebruiken we hiervoor een controlestation, dewelke in ons onderzoek vervangen wordt door een fuzzer.

De fuzzer, geschreven in Python, zal uitgevoerd worden op de host machine, met als besturingssysteem OS X El Capitan (8gb RAM, 2,4 GHz Intel Core i5). Beide machines gebruiken statische ip adressen om netwerk analyse te vereenvoudigen.

Om de fuzzer te verbinden met de virtuele drone moet een three-way handshake geïnitialiseerd worden. De handshake kan in Python zeer eenvoudig uitgevoerd worden door middel van drie lijnen code, te zien in Listing 2. Deze code creëert een nieuwe stream socket en gebruikt vervolgens de *connect()* methode op de socket om de three-way handshake te initialiseren. Wanneer beide machines een verbinding hebben, kan data gestuurd worden door de *send()* methode op te roepen op de gecreëerde socket.

Listing 2. TCP Connection

```
import socket

s = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
s.connect(("192.168.56.102", 5760))
```

B. Fuzzing Methode

Doorheen het onderzoek starten we met een eenvoudige fuzzer die initieel geen gebruik maakt van de protocol specificaties en later voegen we meer intelligentie toe voor gedetailleerdere testgevallen. De waarde van elk veld in een MAVLink bericht (in hexadecimale waarde) zoals gebruikt door de fuzzer wordt hieronder beschreven. Het *Sync* veld krijgt de vaste waarde *"fe"*. De waarde van het *Length* veld wordt gelijk gesteld aan de lengte van de payload, dewelke kan variëren van nul tot 255 bytes. Met elk verzonden bericht wordt de waarde van *Seq* met één verhoogd en terug gezet op nul indien het maximum van 255 bereikt wordt. Het *SysID* en *CompID* worden gelijkgesteld aan *"ff"* en *"00"* respectievelijk. De *MsgID* is een waarde is het gebied tussen *"00"* en *"ff"*. De *Payload* omvat de parameters die intern gebruikt worden (vb. de hoogte van de drone) en wordt geconstrueerd op verschillende manieren, afhankelijk van de testgevallen. De *CRC* waarde is gegenereerd gebruikmakend van een CRC-16 functie die een polynoom generator 1021 gebruikt. De initiële waarde is *"FFFF"* en de input data bytes worden omgedraaid alsook het CRC resultaat alvorens de finale XOR operatie. De input voor de generatie functie is als volgt: *Length+seq+SysID+CompID+MsgID+Payload+Seed.* De *Seed* wordt gebruikt om na te gaan of beide communicerende entiteiten dezelfde berichtenset ondersteunen.

C. Testgevallen

In deze sectie geven we een beschrijving van de verschillende testgevallen die uitgevoerd werden. In de eerste tests wordt er volledig willekeurige data doorgestuurd naar de drone. Deze data omvat zowel getallen, letters als karakters. De data zal karakter per karakter naar de drone verstuurd worden en dit voor een stijgende lengte tot 1000 karakters. Deze test wordt uitgevoerd om na te gaan hoe de software omgaat met ongeldige data. Vervolgens wordt deze test herhaald maar in plaats van karakter per karakter te verzenden sturen we de rij van karakters in zijn geheel door.

In de tweede test wordt voor elk message ID een bericht gecreëerd met een payload gaande van de minimum lengte (1 byte) tot de maximum lengte (255 bytes). De payload bestaat hier uit volledig willekeurige combinaties van hexadecimale waarden. Deze test wordt gebruikt om na te gaan hoe semi geldige berichten afgehandeld worden. Aangezien deze berichten gebruik maken van de correcte berichtenstructuur, correcte sync waarde, correcte sendID en compID en vervolgens een correcte controlesom, kunnen ze tot diep in de software doordringen.

In de derde test testen we het gedrag van de virtuele drone indien een bericht zonder payload wordt ontvangen. Het doel van deze test is om kwetsbaarheden te vinden die onafhankelijk zijn van de payload.

De volgende twee testen worden gebruikt om na te gaan hoe de virtuele drone met berichten omgaat die opgebouwd zijn met de minimum en maximum waarden.

De vierde test maakt berichten waarvan de payload volledig uit de minimum waarde , *"00"*, bestaat, voor alle mogelijke lengtes van de payload. De vijfde test doet hetzelfde maar voor de maximum waarde, *"ff"*.

In de zesde test repliceren we de tweede test. Het enige verschil deze keer is dat het bericht niet byte per byte verstuurd wordt. Een volledig bericht inclusief alle noodzakelijke velden zal toegevoegd worden aan een TCP pakket. De lengte van het bericht zal stijgen door toenemende payload.

In de zevende test wordt getracht kwetsbaarheden te vinden die afhangen van de waarde van het Length veld en de werkelijke lengte van de payload. De lengte van de payload zal correct berekend worden, maar de waarde van het Length veld is steeds vermeerderd of verminderd met één.

In de achtste test wordt eenzelfde bericht duizend maal verzonden. Dit zal gebeuren voor verschillende berichten, met verschillende message IDs.

IV. Resultaten

Na het uitvoeren van de vorige testgevallen bespreken we de bekomen resultaten. Op basis van deze tests konden we een mogelijk veiligheidsgebrek aantonen. De test met de willekeurige hexadecimale payload die niet byte per byte verzonden werd, test zes, slaagde erin de virtuele drone te crashen. De fout die veroorzaakt werd door het fuzzing script, een floating point exception, is terug te vinden in Listing 3. Een floating point getal is een zwevendekommagetal. Een floating point exception is een fout die voorkomt wanneer een niet toegestane handeling wordt uitgevoerd met een zwevendekommagetal zoals het delen door nul.

Listing 3. Flaoting Point Exception

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

Voor verdere analyse maken we gebruik van de gdb debugger. Na het uitvoeren van het `backtrace full` commando is het mogelijk om gedetailleerde info te verkrijgen over de bekomen fout. Deze details zijn terug te vinden in Listing 4, dat een fragment van de output bevat. Uit deze details valt op te maken dat het verzonden bericht message ID 27 heeft. Dit is echter de hexadecimale waarde, dewelke overeen komt met 39 decimaal. Dit is het ID van een MAVLINK_MSG_ID_MISSION_ITEM bericht. Documentatie over dit bericht, terug te vinden in The MAVLink Tutorial For Dummies [21], zegt dat dit bericht zeer belangrijk is omdat het gebruikt wordt om realtime actie te ondernemen (tussenstops ingeven, thuis locatie instellen, servo zetten, etc.). De methode die instaat voor het afhandelen van dit bericht is terug te vinden in Listing 5 en voorziet ons van de extra informatie dat de gegevens van het commando opgeslagen zullen worden in het intern geheugen van de drone.

Listing 4. GDB backtrace full

```
#3  0x0000000000427400 in GCS_MAVLINK::
    update (this=this@entry=0x6ff090 <
    copter+5904>, run_cli=...)
  at /home/karel/ardupilot/libraries/
     GCS_MAVLink/GCS_Common.cpp:869
      c = 119 'w'
      i = <optimized out>
      msg = {checksum = 30546, magic =
          254 '\376', len = 182 '\266',
          seq = 182 '\266',
        sysid = 255 '\377', compid = 0
           '\000', msgid = 39 '\'',
           payload64 = {..., 0, 0, 0, 0,
           0, 0, 0, 0, 0, 0}}
      status = {msg_received = 67 'C',
          buffer_overrun = 198 '\306',
          parse_error = 35 '#',
        parse_state =
           MAVLINK_PARSE_STATE_IDLE,
           packet_idx = 182 '\266',
        current_rx_seq = 183 '\267',
           current_tx_seq = 76 'L',
           packet_rx_success_count = 171,
        packet_rx_drop_count = 0}
      tnow = <optimized out>
      wp_recv_time = <optimized out>
      nbytes = 44
```

Listing 5. MAVLink Handle msgID 39

```
// GCS has sent us a command from GCS,
    store to EEPROM
     case MAVLINK_MSG_ID_MISSION_ITEM: {
        // MAV ID: 39
        handle_mission_item(msg, mission)
            ;
        break; }
```

Vervolgens verfijnen we het onderzoek verder door de focus te verleggen naar het bericht dat de fout veroorzaakte. Op deze manier hopen we meer inzicht te krijgen in de oorzaak van de fout. Hiervoor schrijven we een extra test waarbij het ID gelijk blijft aan 39. De testcode is terug te vinden in Listing 6.

Broncode:lijn	Aantal
AP_Mission.cpp:514	1
AP_Mission.cpp:521	1
AP_Mission.cpp:578	1
AP_Mission.cpp:588	1
AP_Mission.cpp:597	1
AP_Mission.cpp:598	2
AP_Mission.cpp:613	1
AP_Mission.cpp:624	2
AP_Mission.cpp:629	2
AP_Mission.cpp:646	5
AP_Mission.cpp:652	1
AP_Mission.cpp:661	1
AP_Mission.cpp:692	1
AP_Mission.cpp:703	4
AP_Mission.cpp:709	1

Tabel I
FLOATING POINT EXCEPTIONS

De payload bevat willekeurige hexadecimale waarde, voor een lengte gaande van één tot 255 bytes.

Listing 6. Test Case voor msgID 39

```
def send_mavlink_message39(testNb,
    initialSeq):
     for i in range(0,1001):  # repeat
         1000 times
          seq = initialSeq
          for j in range(0,256): # for
              every msgID
               string = construct_msg.
                   construct_string(seq,39,j)
                    # generate message with
                   msgID 39 and random
                   payload of length j
               s.send(string.decode('hex'))
               data = s.recv(4096)
```

Deze test is in totaal 25 keer uitgevoerd. In elk van de gevallen resulteerde dit in een floating point exception. Soms gebeurde dit op dezelfde locatie, maar de tests slaagden er in om 15 unieke gevallen te genereren. De resultaten van deze test zijn terug te vinden in Table I.

V. Bespreking van de Resulaten

In deze sectie bespreken we de resultaten van voorgaande analyse. Door de testgevallen konden 15 verschillende floating point exceptions bekomen worden. Deze kwamen allemaal voor door berichten met ID 39 te sturen, MAVLINK_MSG_ID_MISSION_ITEM berichten. Deze berichten slaagden er in om de virtuele drone te crashen.

Volgend uit de gegevens van Table I kwamen de fouten voor in AP_Mission.cpp. Door de broncode van AP_Mission.cpp te onderzoeken hopen we een kwetsbaar patroon te ontdekken dat deze fout kan uitlokken. Een overzicht van alle lijnen code waar de fout bekomen werd is terug te vinden in Listing 7. Al deze lijnen code maken deel uit van de mavlink_to_mission_cmd methode. Deze methode vormt een bericht naar een AP_Mission::Mission_Command object dat opgeslagen wordt in het interne geheugen van de drone.

Listing 7. Lijnen code Gevoelig voor Floating Point Exception

```
514: radius_m = fabsf(packet.param3);
521: cmd.p1 = packet.param1;
578: cmd.content.location.lat = packet.
   param1 * 100;
588: cmd.content.yaw.direction = packet.
   param3;
597: cmd.content.jump.target = packet.
   param1;
598: cmd.content.jump.num_times = packet.
   param2;
613: cmd.p1 = packet.param1;
624: cmd.content.repeat_relay.
   repeat_count = packet.param2;
629: cmd.content.servo.channel = packet.
   param1;
646: cmd.p1 = packet.param1;
652: cmd.content.digicam_configure.
   aperture = packet.param3;
661: cmd.content.digicam_control.zoom_pos
    = packet.param2;
692: cmd.content.gripper.action = packet.
   param2;
703: cmd.p1 = packet.param1;
709: cmd.content.altitude_wait.
   wiggle_time = packet.param3;
```

Door deze lijnen broncode te observeren hebben we het vermoeden dat de fouten bekomen worden door een convertering van een zwevendekommagetal naar een data type dat intern gebruikt wordt. Alle fouten, buiten één enkele, bevatten een toekenning van een waarde uit de payload aan een cmd struct. Het geobserveerde patroon is terug te vinden in Listing 8.

Listing 8. Pattern Resulting in Floating Point Exception

```
cmd.field = packet.param; // with param
   equal to param1, param2 or param3
```

VI. Besluit

Omdat drones de dag van vandaag alomtegenwoordig zijn, is de veiligheid van drones een zeer belangrijk onderzoeksdomein geworden. Omdat het communicatiekanaal tussen de drone en het controlestation een pijnpunt is door het gebrek aan (voldoende) veiligheidsmechanismen, werd dit het aandachtspunt van ons onderzoek. In het bijzonder focussen we op het MAVLink protocol voor bidirectionele communicatie tussen drone en controlestation. Een van de redenen is dat het ontwikkelingsteam van MAVLink tracht om het te laten uitgroeien tot een wereldwijde standaard.

Voorgaand onderzoek naar MAVLink legde de focus op het toevoegen van encryptie aan het protocol. Wij wilden daarentegen een bijdrage leveren aan de veiligheidsanalyse door het probleem vanuit een ander standpunt te bekijken. We zijn er van overtuigd dat sterke beveiliginsmechanismen op zich niet voldoende zijn, maar dat er ook nood is aan een veilige implementatie. Om dit te onderzoeken maakten we gebruik van fuzzing technieken om gebreken en kwetsbaarheden in de protocol implementatie te analyseren.

Om ons onderzoek te kunnen voeren hebben we de specificaties van het MAVLink protocol in detail bestudeerd, waarbij de nadruk werd gelegd op de berichtenstructuur. Op basis van deze specificaties hebben we een fuzzer gecreëerd die in de mogelijkheid is om geldige MAVLink berichten op te stellen. Door gebruik te maken van een zelfgemaakte fuzzer was het onderzoek onafhankelijk van de ontwikkelaars code om berichten op te stellen. Op deze manier konden we de focus leggen op het afhandelen van berichten door het protocol zoals het geïmplementeerd is op de drone.

De testgevallen werden opgesteld op basis van de meest voorkomende aanvalsvectoren zoals willekeurige data en minimum of maximum waarden. Omdat de berichtstructuur gerespecteerd werd was de fuzzer in staat berichten te creëren die tot diep in het protocol konden doordringen. Vervolgens kon het gedrag van de drone aandachtig bestudeerd worden.

Door te tests uit te voeren en te analyseren kan vastgesteld worden dat er kwetsbaarheden in de implementatie van het protocol terug te vinden zijn. De testen maakten het mogelijk om een beduidend aantal floating point exceptions te bekomen. De analyse wees uit dat deze fout het eerst voor kwam bij messageID 39. Door te tests verder te verfijnen konden we tot 15 fouten genereren, op verschillende plaatsen in de code. Verdere analyse toonde aan dat de meerderheid van deze fouten resulteerde uit een specifiek patroon in de code. Dit geeft ons een sterke indicatie dat er tal van andere fouten gevonden kunnen worden door ook de andere messageIds even grondig te testen. We veronderstellen dat de floating point exception bekomen worden door het omvormen van een zwevendekommagetal naar een intern data type. Het door ons opgestelde fuzzing script was in de mogelijkheid om in minder dan één minuut de floating point exception te bekomen, wat resulteerde in het crashen van de virtuele drone.

Daarentegen moeten we wel stellen dat floating point exceptions niet geactiveerd zijn wanneer een echte operationele drone gebruikt wordt. Indien men bijvoorbeeld bij een operationele drone een deling door nul doet, dan zal dit niet resulteren in een floating point exception maar zal het resultaat gelijk zijn aan oneindig. Hierdoor kunnen we niet met zekerheid zeggen of we ook in de mogelijkheid zijn om een echte drone te laten crashen door het fuzzing script. Desalniettemin moeten we stellen dat floating point exceptions ongewenst zijn en een indicatie kunnen zijn voor andere onderliggende gebreken. Indien het mogelijk is om dezelfde resultaten te bekomen met een operationele drone, dan kan dit serieuse veiligheidsrisico's met zich meebrengen.

Tegelijkertijd hebben we de toepasbaarheid van de fuzzing methode bestudeerd als een toevoeging aan de huidige technieken om kwetsbaarheden te vinden in het protocol. Sinds MAVLink in 2009 geïntroduceerd werd hebben er duizenden ontwikkelaars en software testers zich bij het project gevoegd. Zoals de sectie over gerelateerd werk laat uitschijnen, is het MAVLink protocol al zeer hard onderworpen aan tal van veiligheidsanalyses. Het feit dat we door gebruik te maken van een basis fuzzer 15 unieke floating point exceptions konden bekomen, duidt er op dat de fuzzing techniek zeker een meerwaarde te bieden heeft voor een grondige en complete veiligheidsanalyse. Het ontwikkelen van het fuzzing script verliep zeer vlot eens de berichtenstructuur en protocolspecificaties grondig doorgenomen werden. De fuzzer is in staat om in een mum van tijd duizenden testberichten op te stellen, op een geautomatiseerde wijze. We kunnen besluiten dat het zeker de moeite loont om fuzzing technieken toe te voegen aan de toolbox van onderzoeksmethoden.

VII. Toekomstwerk

Toekomst werk kan eruit bestaan om het volledige gebied van testgevallen te doorlopen om zo meerdere kwetsbaarheden in de implementatie van MAVLink te ontdekken. Omdat we met onze fuzzer slechts een fractie van alle mogelijke payload hebben kunnen testen (willekeurig), kan het zijn dat we verschillende mogelijke berichten die de drone doen crashen nog niet ontdekt hebben. We moeten wel benadrukken dat alle mogelijke permutaties van mogelijke payload, voor elke mogelijke lengte, heel wat geheugencapaciteit vraagt. Met de combinatie van het huidige script en de huidige setup is het niet mogelijk om alle mogelijkheden te proberen (lees: bruteforcen). Verbeterde fuzzing scripts, die beter met het geheugen omgaan, kunnen hier een oplossing bieden. Indien meerdere berichten gevonden worden, kan dit eventueel leiden tot het construeren van één enkel bericht dat in staat is de (virtuele) drone te doen crashen. Men kan ook gebruik maken van meer geavanceerde fuzzing platformen zoals Sulley [6]. Deze platformen gebruiken gespecialiseerde technieken om de te injecteren data te construeren. We vermoeden dat het gebruik van dergelijke platformen zal leiden tot het ontdekken van vele andere gebreken in de implementatie. Vervolgens zou dit ondezoek herhaald moeten worden op een operationele drone. Hierdoor kan nagegaan worden of de gevonden kwetsbaarheden al dan niet de operationele drone kunnen doen crashen.

Referenties

[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.Ñeystadt, *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.
[24] N. O. Tippenhauer, C. Popper, K. B. Rasmussen and S. Capkun, *On the Requirements for Successful GPS Spoofing Attacks*, ETH Zurich, Switzerland and UCI, Irvine, CA, 2011.
[25] Capture Video Feed, http://www.cbsnews.com/news/insurgents-intercepted-drone-spy-videos/, 03 05 2016.

Appendix E

Poster

KATHOLIEKE UNIVERSITEIT
LEUVEN

FACULTEIT
INGENIEURSWETENSCHAPPEN

Master of Science in Engineering: Computer Science

Master's thesis
Karel Domin

Promotor
Prof. Dr. Ir. B. Preneel
Prof .Dr. Ir. F. Piessens

Academic year
2015-2016

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

Context

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 a GCS or a remote control. Drones are ubiquitous because they perform tasks that humans aren't capable of or reach places that humans can't reach. Besides the clear benefits of using drones, they can also pose important security and privacy threats. in particular, 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 even alter its software. If a drone is hijacked, it can be turned into a projectile, posing serious threats.

An interesting communication protocol is the MAVLink protocol for bi-directional communication between the drone and a GCS, and is attempting to become a worldwide standard. The MAVLink protocol does *not* provide any security measures apart from a checksum for packet corruption detection. Adding encryption to the MAVLink protocol would make the protocol more secure for certain attacks, but is until today not yet implemented in the official distribution of the MAVLink protocol. However, securing the wireless channel alone is not enough, we also need a secure implementation. We want to further search the space of software vulnerabilities of the MAVLink protocol and look at the same security problems from a different perspective. We need a combination of both strong security measures and a secure implementation.

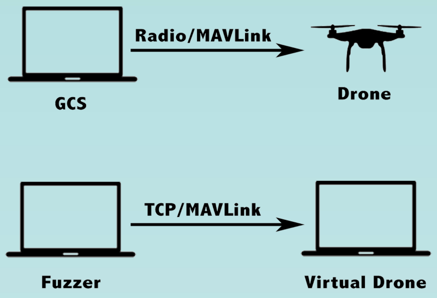
Goals

Our goal is to carry out a software security analysis of the MAVLink protocol and investigate potential design or implementation protocol flaws. For this, we use fuzzing techniques. The goal of fuzzing is to inject invalid or semi-invalid data to produce an unexpected software behaviour or crash the software. In addition, we want to study the suitability of the fuzzing technique as a complementing method for identifying possible security flaws in the implementation of the MAVLink protocol. We constructed a fuzzer capable of creating MAVLink messages in an automated manner. We briefly formulate three different research questions that we would like to explore more in detail in the rest of our work. This includes:

1. how can we identify software security flaws in the MAVLink framework?
2. Is the fuzzing technique suitable for discovering implementation flaws in the MAVLink protocol?
3. What are the consequences of exploiting these security flaws?

Methodology

In the real world, the setup consists of a drone and a GCS. Performing the lab tests on a real drone is not achievable. There is need for a simulator for the drone that behaves like the real drone. The SITL environment from DroneCode offers exactly what we need. Our fuzzer establishes a connection with the virtual drone and starts sending custom messages. The difference between a real operational setup and the lab setup can be seen in the figure below.



Results

Resulting from the conducted test cases, we were able to identify a possible security flaw. The fuzzer generated an exception, crashing the virtual drone. The encountered exception was a floating point exception, can be seen below:

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

A floating point number is a number with a decimal point and a *floating point exception* is an error that occurs when an operation is performed on a floating point number that is not allowed. The message ID of the message resulting in the exception was 39, which turns out to correspond to a MAVLINK_MSG_ID_MISSION_ITEM message. The documentation about this states that this is an important message since it is used for taking real-time action (e.g. Setting waypoints, Loiter turns, Set home, Loiter time, Repeat servo, Set servo, etc.), and stores data in the internal memory of the drone. After finding this possible vulnerability, we further refined our test cases, with focus on message ID 39. We were able to obtain 15 unique floating point exceptions. This gives us a strong indication that more implementation flaws can be found by testing the rest of the message IDs as thoroughly as this one. From the analysis of the test cases we found a vulnerable pattern that is used throughout the code:

```
cmd.field = packet.param;
```

We assume the floating point exception is thrown by trying to convert a floating point number to an internal data type. Our fuzzing script was able to generate this exception within less then a minute, resulting in crashing the virtual drone.

However, we have to stress that floating point exceptions are not enabled in real operational drones, only in the simulation. Therefore, we do not know whether we can crash a real drone or not. Nevertheless, floating point exceptions are undesired and are usually an indication for an underlying bug or vulnerability. The floating point exceptions indicate that some cases are not tested or handled completely. The research should be repeated on a real operational drone and if it is possible to achieve the same results, this poses serious security threats.

Bibliography

[1] ardupilot. Software in the loop. URL:http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html. last checked on 2016-03-24.

[2] ieee.org. Ar drone that infects other drones with virus wins dronegames. URL:http://spectrum.ieee.org/automaton/robotics/diy/ar-drone-that-infects-other-drones-with-virus-wins-dronegames. last checked on 2016-05-03.

[3] SamyK. Skyjack. URL: https://github.com/samyk/skyjack. last checked on 2016-05-03.

[4] garage4hackers. Maldrone. URL:http://garage4hackers.com/entry.php?b=3105. last checked on 2016-05-03.

[5] mitre.org. Gps spoofing. URL:https://capec.mitre.org/data/definitions/628.html. last checked on 2016-05-03.

[6] K. B. Rasmussen N. O. Tippenhauer, C. Popper and S. Capkun. On the requirements for successful gps spoofing attacks. Technical report, ETH Zurich, Switzerland and UCI, Irvine, CA, 2011.

[7] R. Creutzburg J. S. Pleban, R. Band. Hacking and securing the ar.drone 2.0 quadcopter - investigations for improving the security of a toy. Technical report, Brandenburg University of Applied Sciences, 2014.

[8] cbsnews. Insurgents intercepted drone spy videos. URL:http://www.cbsnews.com/news/insurgents-intercepted-drone-spy-videos/. last checked on 2016-05-20.

[9] J. A. Marty. Vulnerability analysis of the mavlink protocol for command and control of unmanned aircraft. Master's thesis, Air Force Institute of Technology, 2014.

[10] T. M. DuBuisson. Smaccmpilot secure mavlink communications. Technical report, Galois, Inc.1, 2013.

[11] A. Stewart N. Butcher and Dr. S. Biaz. Securing the mavlink communication protocol for unmanned aircraft systems. Technical report, Appalachian State University, Auburn University, 2013.

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

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

[14] B. Hond. Fuzzing the gsm protocol. Master's thesis, Radboud University Nijmegen, 2011.

[15] owasp. Owasp fuzzing. URL:`https://www.owasp.org/index.php/Fuzzing`. last checked on 2016-05-03.

[16] M. Hillman. 15 minute guide to fuzzing. URL:`https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/`. last checked on 2016-05-03.

[17] J. Neystadt. Automated penetration testing with white-box fuzzing. URL:`https://msdn.microsoft.com/en-us/library/cc162782.aspx`. last checked on 2016-03-24.

[18] qgroundcontrol. Mavlink protocol. URL:`http://qgroundcontrol.org/mavlink/start`. last checked on 2016-05-03.

[19] qgroundcontrol. Generator for compiling xml definitions to c/c++/c# or python code. URL:`http://qgroundcontrol.org/mavlink/generator`. last checked on 2016-05-25.

[20] S. Balasubramanian. Mavlink tutorial for absolute dummies (part-i). URL:`http://dev.ardupilot.com/wp-content/uploads/sites/6/2015/05/MAVLINK_FOR_DUMMIESPart1_v.1.1.pdf`. last checked on 2016-05-03.

[21] ardupilot. Command message structure. URL:`http://ardupilot.org/dev/docs/ardupilot-mavlink-command-package-format.html`. last checked on 2016-05-25.

[22] S. Reifegerste. Crc generator. URL:`http://www.zorc.breitbandkatze.de/crc.html`. last checked on 2016-05-25.

[23] gnu. Gdb: The gnu project debugger. URL:`https://www.gnu.org/software/gdb/`. last checked on 2016-05-03.

[24] autofuzz. Autofuzz. URL:`http://autofuzz.sourceforge.net/`. last checked on 2016-05-20.

[25] fuzzing.org. Sulley manual. URL:`http://www.fuzzing.org/wp-content/SulleyManual.pdf`. last checked on 2016-03-24.

[26] secdev. Scapy. URL:`http://www.secdev.org/projects/scapy/`. last checked on 2016-05-20.

[27] wireshark. Tcp zerowindow - the wireshark wiki. URL:https://wiki.wireshark.org/TCP%20ZeroWindow. last checked on 2016-05-05.

Master thesis filing card

Student: Karel Domin

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

Dutch title: Veiligheidsanalyse van het Drone Communicatieprotocol: Fuzzing Toegepast op het MAVLink Protocol

UDC: 681.3

Abstract:

In our current society, commercial drones have become common. They are used by hobbyists as well as by professionals to support critical services. Even though drones can provide many benefits, they can also pose serious security threats. Several attacks against drones have emerged the last few years. Together with the attacks, a considerable amount of research is performed trying to establish secure drone operations. An important part of the drone security research is focused on the wireless communication channel. This channel is vulnerable because the lack of security measures. Research did focus on providing a secure wireless communication channel. However, securing the wireless channel alone is not enough, there is also a need for a secure implementation. In our research we focussed on the MAVLink protocol because this is trying to become a worldwide standard. Our goal is to carry out a security analysis of the MAVLink protocol and investigate potential design or implementation flaws in the protocol. For this, we use fuzzing techniques. In our work we created a fuzzer capable of generating random and semi-valid MAVLink messages which can be injected in the protocol. Our experiments resulted in the generation of a floating point exception, which made the virtual drone crash. After analysing the exception, other test cases were constructed, further refining our test method. This led to the identification of 15 vulnerable implementation parts were the floating point exceptions can be thrown. Further analysis of the results indicated that a certain combination of sent messages with random data posed a vulnerability for the drone. We were able to identify a certain pattern in the implementation that is vulnerable to this attack.

Thesis submitted for the degree of Master of Science in Engineering: Computer Science, specialisation Secure Software

Thesis supervisors: Prof. dr. ir. B. Preneel
Prof. dr. ir. F. Piessens

Assessors: Prof. dr. T. Holvoet
dr. Ir. M. Mustafa

Mentors: Ir. E. Marin
Ir. I. Symeonidis