Sniffing and Spoofing of UAV C2 Messages

Babak Badnava
Department of Electrical Engineering and Computer Science, University of Kansas

***Abstract*—In this project, we demonstrate and revisit an important attack on the unmanned aerial vehicle's (UAV) communication link with ground control stations (GCS) that is sniffing and spoofing of MAVLink messages. We simulate a UAV using a software-in-the-loop (SITL) simulator, connect it to a GCS over a network, and then show that how an attacker in the same network can listen to the packets being transmitted to the UAV and manipulate them to disarm the UAV and force it to land.**

***Index Terms*—Sniffing, Spoofing, Man in the middle attack, Unmanned Aerial Vehicles**

I. Introduction

Recently, unmanned aerial vehicles (UAVs) have attracted a lot of attention and have been used in various applications and industries including but not limited to crop spraying, mapping and surveying, aerial photography and videography, search and rescue, entertainment, and military purposes. These UAVs are controlled by a ground control station (GCS) over a network. The micro air vehicle link (MAVLink) is a communication protocol that is being used for communications between UAVs and GCSs. This protocol has been used in major autopilot systems such as ArduPilot and PX4. However, it has been shown that the MAVLink protocol has multiple vulnerabilities [1, 2].

In this project, we revisit these vulnerabilities and demonstrate multiple types of attacks on MAVLINK packets including sniffing and spoofing. We show two different techniques to sniff and analyze MAVLink packets. The first way is by using Wireshark, which is a popular network analysis tool, and the second technique is to develop a Python code that is capable of sniffing MAVLink packets being transmitted over a network interface. We also briefly investigate the defense mechanism that could be incorporated to secure the MAVLink protocol [3].

II. Attack Model

We consider a man-in-the-middle (MITM) attack, as depicted in Fig.1, where the attacker can sniff and spoof the packets that are being transferred between the GCS and the UAV. For the purpose of this project, we simulate the drone using the AruPilot software in the loop (SITL) simulator to simulate the behavior of the UAV. The SITL simulator allows us to run different types of vehicles, such as Plane, Copter, or Rover, that use ArduPilot as their autopilot software without any hardware. This SITL will be launched separately in a docker container. There is also another docker container containing the GCS code that sends commands to the UAV. And another docker container for the attacker where it launches its attacks, either sniffing or spoofing attacks.

In this project, we assume that the attacker is on the same network as GCS and UAV, thus we don't need to consider the wireless technology used for the UAV. In other words, the attacker is located somewhere near the GCS on the same network as GCS and UAV.

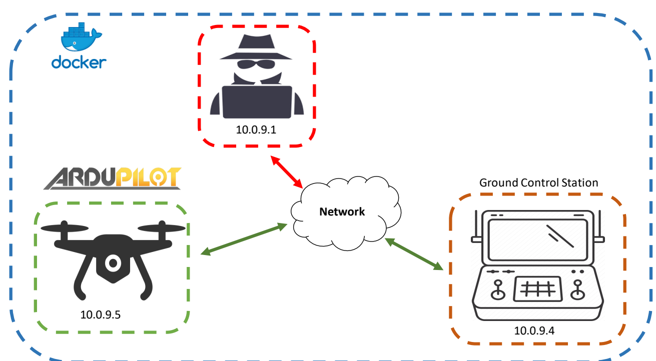

Fig. 1: Attack Model

III. Implementation

In this section, we explain how we implement our attack and simulation. We show how we can use the ArduPilot SITL to simulate a drone and then how to sniff and spoof MAVLink packets that are begin transferred over the network between the GCS and

UAV. We demonstrate two different techniques for sniffing the MAVLink packets; First, we use Wireshark to sniff packets, then we show how we can use Scapy, an interactive Python packet manipulation library, to sniff and spoof the MAVLink packets.

A. *Drone Simulation*

The Ardupilot SITL simulator allows us to run Plane, Copter, or Rover without any hardware. It gives us a native C++ executable code that allows us to test the behavior of the autopilot code without any hardware.

First, we need to set up and build the Ardupilot code [4], and then we can use the SITL simulator to simulate a drone by executing the following command on the terminal:

```
sim_vehicle.py -v ArduCopter --map
    --console
```

By running this command, we are asking the SITL to run an instance of ArduCopter code for us and activate the map and console plugins. The map plugin helps us to visualize the environment in which the UAV is operating, and the console plugin helps us monitor the status of the UAV and the messages it receives during the operation. Fig. 2 is a demonstration of running an instance of the SITL with both map and console plugins enabled.

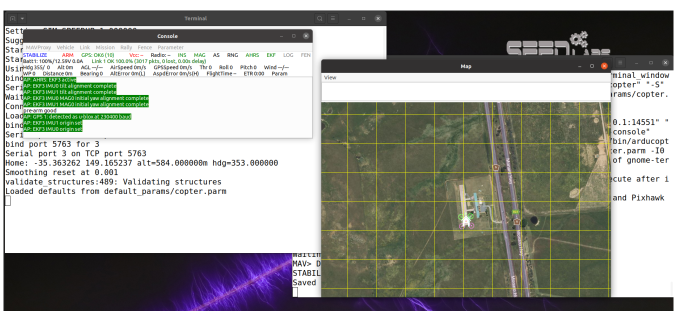

Fig. 2: Ardupilot SITL running an instance of an ArduCopter

B. *Sniffing Using Wireshark*

Wireshark is a powerful network analyzing tool and has been used by many researchers for analyzing and monitoring different network protocols. It provides an interface where users can look at the packets being transmitted over the network and analyze their contents.

In order to be able to analyze the MAVLink packets in the Wireshark, we need to generate a plugin that parses the messages. We generate this plugin using a tool, called MAVGen, which is part of the PyMAVLink library as it has been explained in [5]. We need to run the following command to generate a Lua plugin file.

```
mavgen.py --lang=WLua --wire-protocol=2.0
    --output=mavlink_2_common common.xml
```

The output of this command would be a file, named ”mavlink_2_common.lua,“ which we edit by adding the following lines of code to the end of the file to monitor the ports used by Ardupilot:

```
-- bind protocol dissector to ports
local udp_dissector_table =
    DissectorTable.get("udp.port")
udp_dissector_table:add(14550,
    mavlink_proto)
udp_dissector_table:add(14560,
    mavlink_proto)
local tcp_dissector_table =
    DissectorTable.get("tcp.port")
tcp_dissector_table:add(5760,
    mavlink_proto)
```

Then, we copy the file to the plugin directory of Wireshark: /.local/lib/wireshark/plugins. Now, when we run Wireshark, after choosing the network interface that we want to monitor and applying mavlink_proto filter, we can monitor all of the MAVLink messages being transferred on the chosen interface. Fig. 3 shows the result of running the plugin and sniffing one of the messages; this message is a battery status message sent from the UAV to the GCS.

C. *Sniffing Using Scapy*

In this section, we use Scapy, which is a Python library that enables users to sniff, send, and dissect network packets, to sniff MAVLink messages.

We first need to analyze the message frame structure of the MAVLink messages. Fig. 4 shows the format of MAVLink messages. In MAVLink version 2, the size of each packet can range from 12 to 280 bytes, and every message starts with a packet start marker, which in MAVLink version 2 is 0xFD. There are multiple other fields in the message, but the MSG_ID and PAYLOAD are the two fields that

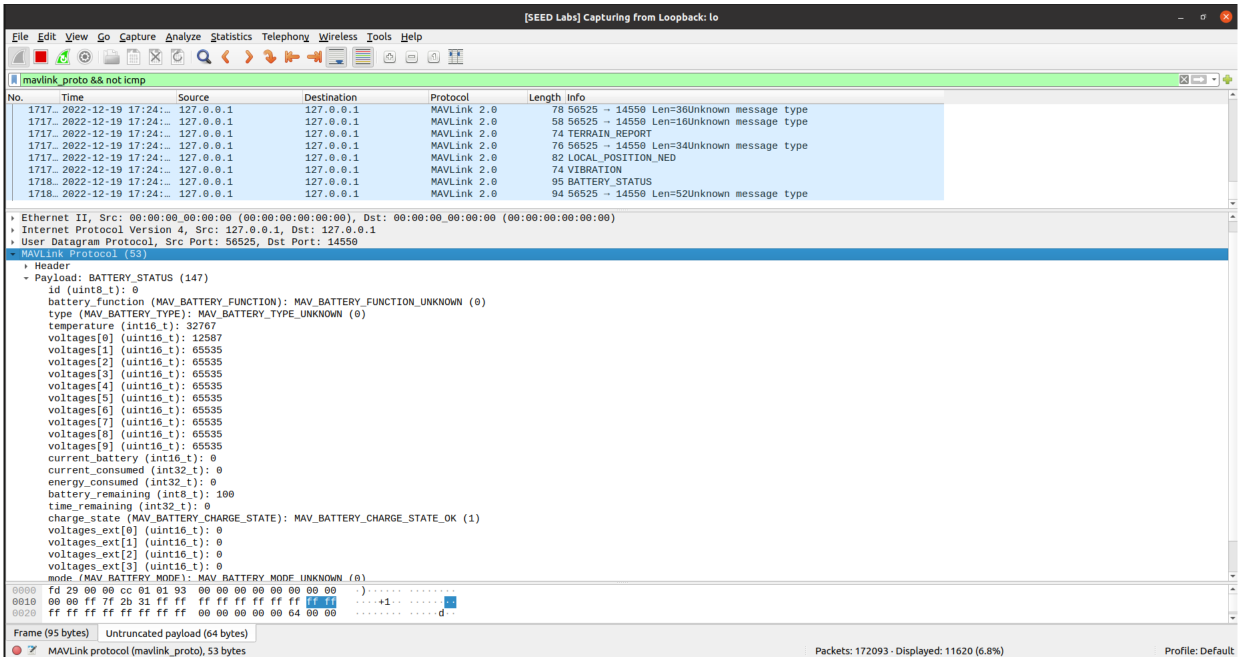

Fig. 3: Result of Executing the Wireshark for Sniffing

Fig. 4: MAVLink 2 Packet Format [6]

distinguish the MAVLink functionalities. A detailed description of other fields in the MAVLink messages is presented in [6] that we use to develop our Scapy packet reader.

The main code that we use to sniff MAVLink messages is presented in the following code block:

```
def print_mavlink_message(pkt):
    if MAVLink in pkt:
       pkt.show()
# Sniff TCP query packets and invoke
    print_mavlink_message().
f = 'tcp and dst port 5760'
pkt = sniff(iface='lo', filter=f,
    prn=print_mavlink_message)
```

This code block filters all of the TCP packets destined to port 5760 on the loopback interface, and pass them to a function called print_mavlink_message that is responsible for printing MAVLink messages. However, Scapy needs a more detailed description of the MAVLink message

Fig. 5: Scapy MAVLink layer

fields in order to deserialize them and print their values. Scapy provides this possibility to add a new protocol easily by defining a new layer and then registering it to Scapy. A layer is composed of a list of fields that comprise the message. We develop

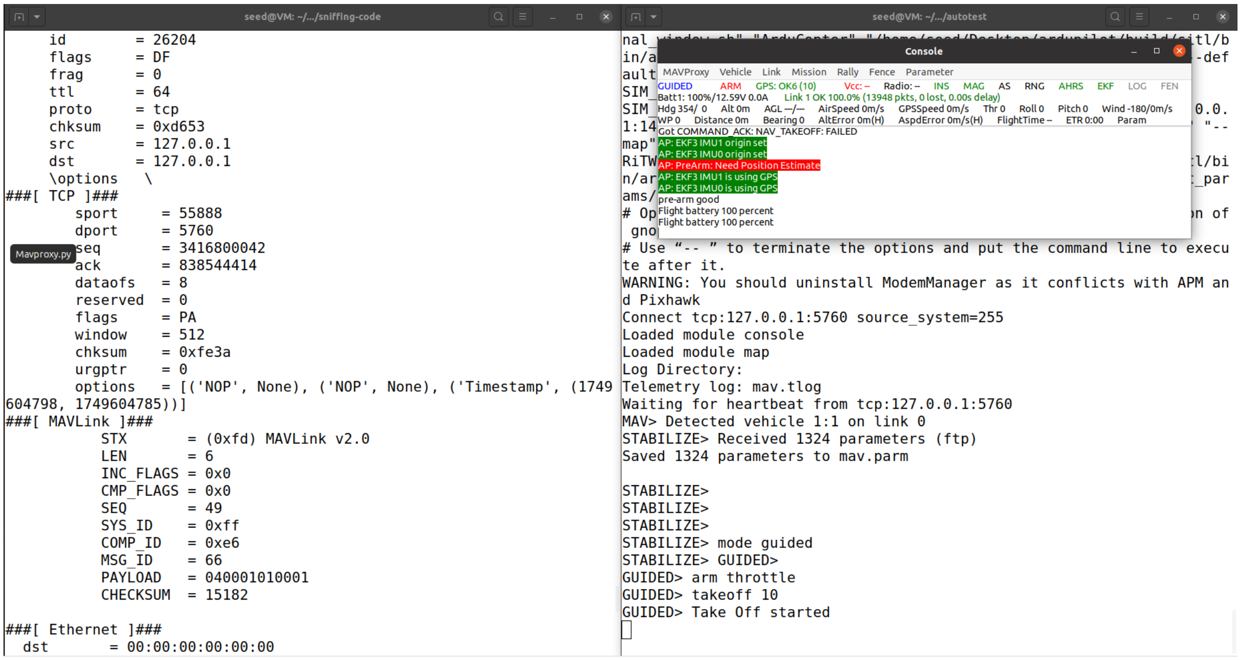

Fig. 6: Result of Executing the Sniffer Code

a layer based on the message format described in [6]. In our implementation depicted in Fig. 5, some of the fields are conditional such as INC_FLAGS, COMP_FLAGS, and SIGNATURE since they may not exist based on the enabled features in the MAVLink protocol.

Fig. 6 demonstrates the result of executing our sniffing code, and it shows a packet captured by the Scapy. The left window, in Fig. 6, is the sniffing code and there are two windows on the right side; the smaller window is a console that shows the state of the simulated UAV, and the window underneath it is our GCS, where you can see we have executed multiple commands to arm the UAV and then ask the UAV to take off to an altitude of 10m above the ground. The captured message, on the left window, with message ID 66 corresponds to the take-off command.

D. Spoofing

In this section, we show how we can spoof a message and force the UAV to land on the ground. First, we need to know what message makes the UAV land on the ground. Then, we can sniff the messages and send the landing command to the UAV at the right moment.

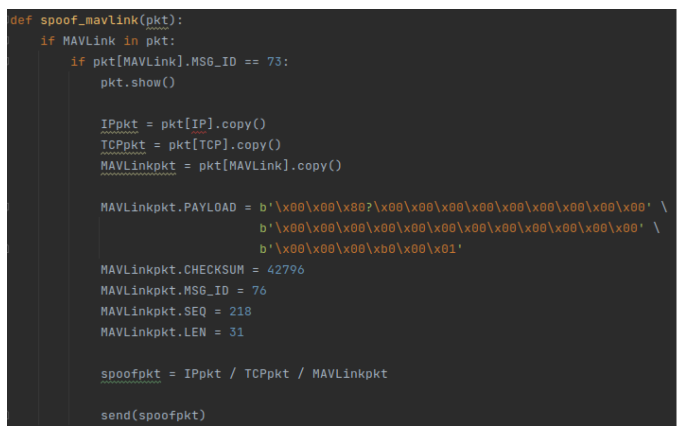

Fig. 7: Scapy Code for Spoofing MAVLink messages

One of the messages that cause the UAV to land on the ground safely is a mode change message, in which we change the operation mode of the UAV to STABILIZE and this will lead the UAV to land on the ground safely. In order to capture the mode change message, we sniffed the messages that have been transferred between the GCS and the UAV, and then developed our spoofing code as depicted in Fig.7. This code would listen to the messages transferred between the GCS and the UAV and then when a message with MSG_ID of 73 was sent to the UAV, it will send a mode change message to the

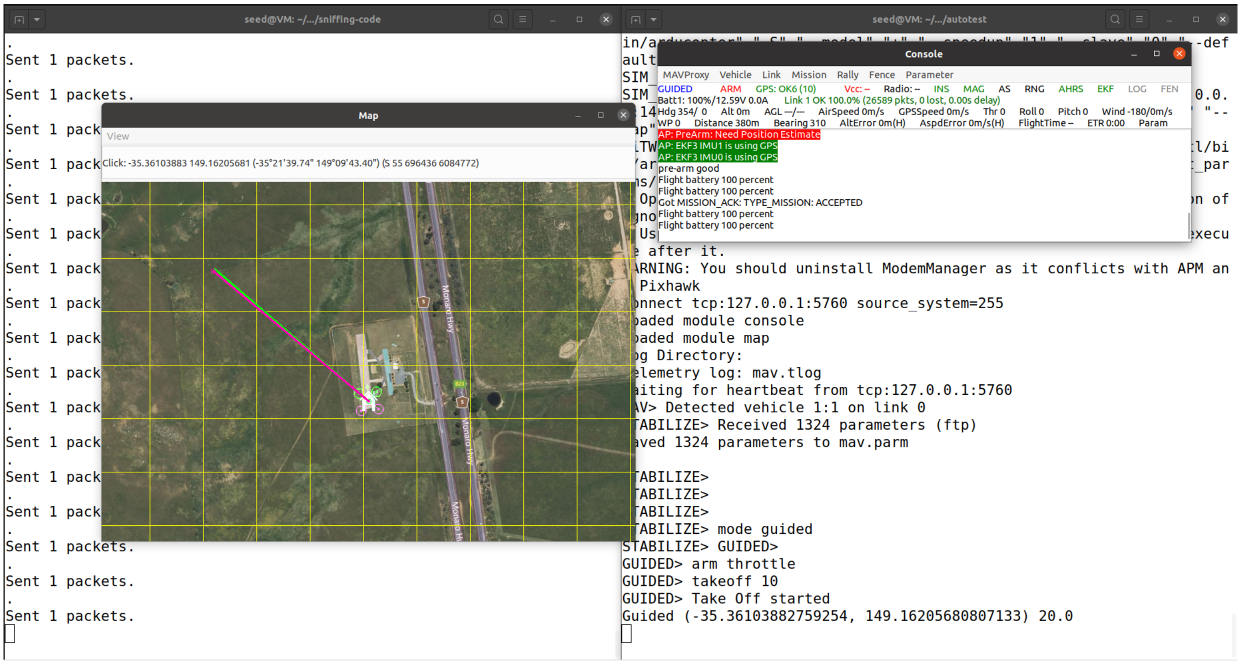

Fig. 8: Result of Executing the Spoofing Code

UAV. MSG_ID of 73 corresponds to the GO_TO message, where the UAV is asked to fly to a certain geographical location.

Fig. 8 shows the result of spoofing a mode-change packet. On the window with the map, the point that the UAV has been asked to fly to is demonstrated, and on the window underneath it, we see the messages sent by the spoofing code. After executing the spoofing code, the UAV stops following the path to the desired point and land on the same location.

IV. SECURING MAVLINK PROTOCOL

As stated in [3], the confidentiality of the exchanged messages between UAVs and GCSs can be guaranteed by encryption. The authors of [3] show that by using different encryption techniques such as Advanced Encryption Standard in Counter Mode (AES-CTR), Advanced Encryption Standard in Cipher Block Chaining Mode (AES-CBC), RC4, and CHaCha20, they can satisfy the confidentiality condition in the UAV's communication.

They show that by encrypting the message payload, they can guarantee confidentiality while using almost the same amount of memory and CPU resources, as depicted in Fig. 10 and Fig.9. All of the used encryption methods except for RC4 show negligible performance degradation compared with insecure MAVLink.

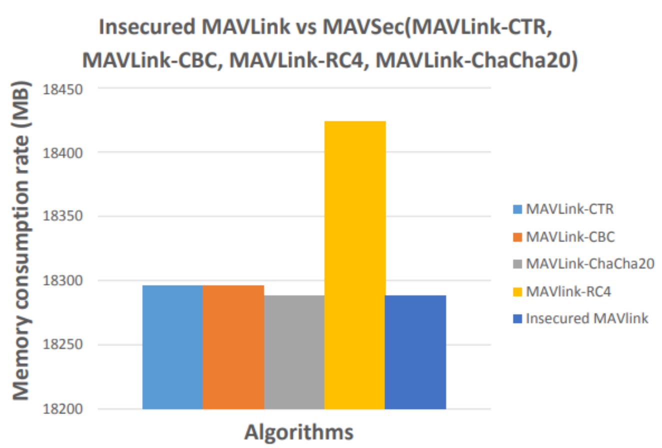

Fig. 9: Memory consumption of the different encryption algorithms used in [3]

V. CONCLUSION

In this project, we investigated a MITM attack model and demonstrated how an attacker can sniff and spoof the MAVLink messages transferred between the UAV and GCS. We developed a Wireshark plugin that enables us to monitor and analyze the MAVLink messages and then developed a Python tool using the Scapy library to sniff the

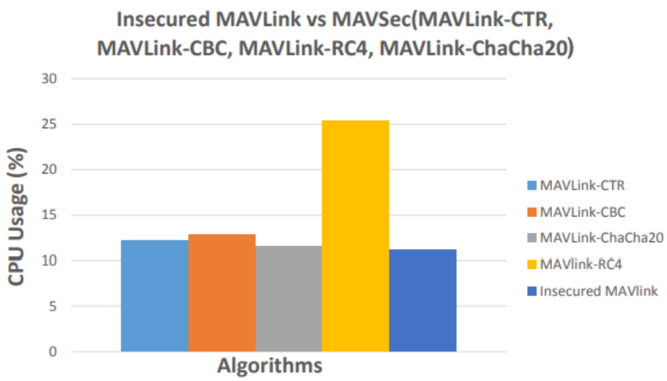

Fig. 10: CPU resource utilization of the different encryption algorithms used in [3]

MAVLink messages. We also developed a script that forces the UAV to land safely on the ground by sending a mode change command to the UAV. The code developed for this project can be accessed from [7]. We also discussed a possible solution to guarantee the confidentiality of the MAVLink messages, where the payload section in the MAVLink message would be encrypted to prevent an attacker from understanding the messages being transferred over the link.

References

[1] Y.-M. Kwon, J. Yu, B.-M. Cho, Y. Eun, and K.-J. Park, "Empirical analysis of mavlink protocol vulnerability for attacking unmanned aerial vehicles," *IEEE Access*, vol. 6, pp. 43 203–43 212, 2018.

[2] H. Xu, H. Zhang, J. Sun, W. Xu, W. Wang, H. Li, and J. Zhang, "Experimental analysis of mavlink protocol vulnerability on uavs security experiment platform," in *2021 3rd International Conference on Industrial Artificial Intelligence (IAI)*, 2021, pp. 1–6.

[3] A. Allouch, O. Cheikhrouhou, A. Koubâa, M. Khalgui, and T. Abbes, "Mavsec: Securing the mavlink protocol for ardupilot/px4 unmanned aerial systems," in *2019 15th International Wireless Communications Mobile Computing Conference (IWCMC)*, 2019, pp. 621–628.

[4] "https://github.com/ArduPilot/ardupilot," https://mavlink.io/en/guide/serialization.html, accessed: 2022-12-19.

[5] "Parsing MAVLink Messages in Wireshark," https://mavlink.io/en/guide/wireshark.html, accessed: 2022-12-19.

[6] "MAVLink Packet Serialization," https://mavlink.io/en/guide/serialization.html, accessed: 2022-12-19.

[7] "MAVLink Man-in-the-middle attack demonstration," https://github.com/badnava-babak/eecs866-final-project, accessed: 2022-12-19.