WHITE PAPER

Not the Drones You're Looking For

Spoofing Drone Locations by Manipulating Remote ID Protocols and Communications

Table of Contents

1. Introduction 4

2. Drone Remote Identification (RID) 5

2.1 RID Background 7

2.1.1 RID Standards and Regulations 7

2.1.2 RID Protocol Security Flaws 7

2.2 DJI and the OcuSync-Based DroneID Protocol 8

3. Open Drone ID (ODID) Protocol 9

3.1 ODID Protocol Background 9

3.1.1 ODID Communication Methods and Technologies 11

3.2 ODID Message Format and Encoding 12

3.2.1 ODID Message Blocks 13

3.2.2 ODID Message Packs 16

3.2.3 ODID Reference Implementation Library 17

3.3 Wi-Fi ODID Traffic Capture Example 21

4. Reverse Engineering the BlueMark DroneScout ds230 24

4.1 DroneScout ds230 Characteristics 24

4.2 DroneScout ds230 Hardware 26

4.3 DroneScout Firmware Reverse Engineering 29

4.3.1 DroneScout Firmware Main Executable: dronescout.arm64 32

4.4 DroneScout ds230 Vulnerabilities 39

4.4.1 Vulnerability Analysis: CVE-2023-29156 40

4.4.2 Vulnerability Analysis: CVE-2023-31191 43

5. DJI, OcuSync and DroneID Protocol 46

5.1 DJI Aeroscope Appliance 46

5.2 OcuSync Communication Methods and Technologies 48

5.2.1 OcuSync Message Format and Encoding 49

5.2.2 Telemetry Signal Detection and Extraction 51

5.2.3 Telemetry Signal Decoding 52

5.2.4 Telemetry Packet Data Structure 53

5.3 Creating Fake DJI Telemetry Packets 55

5.3.1 Replay of OcuSync Telemetry Packets 56

6. DroneScout ds230 Attack Scenarios **57**

6.1 ODID Injection Framework 57

6.2 OcuSync Telemetry Data Injection Framework 58

6.3 Attack Scenarios 59

6.3.1 Attack Scenario 1: Single Drone Forging 61

6.3.2 Attack Scenario 2: Multiple Drone Forging 63

6.3.3 Attack Scenario 3: Drone Cloud 66

6.3.4 Attack Scenario 4: DroneScout Timer (CVE-2023-29156) 68

6.3.5 Attack Scenario 5: DroneScout Adjacent Channels (CVE-2023-31191) 70

7. Conclusion **72**

1. Introduction

Consumer drones freely accessing the airspace are fundamentally changing aviation and posing new safety and security challenges. For this reason, Civil Aviation Authorities (CAAs) worldwide have started pushing for the adoption of Remote Identification (RID) rules and protocols for consumer drones. RID regulations require drones to periodically broadcast their telemetry information, enabling third-party entities such as law enforcement to identify and locate drones and their operators. These regulations began to be finalized around 2022, and in the second half of that year, the first RID-capable consumer drones became available on the market. The support for RID technologies will become mandatory and pervasive in the years to come.

One of the main reasons RID regulations and standards have been developed is that drones are "low observable" objects; they have small radar cross sections, make noise that is difficult to detect beyond a certain distance, can fly at extremely low altitudes and are highly maneuverable and thus can fly under trees and between buildings. These characteristics make existing airspace monitoring technologies, like radar and vision systems, not always suitable for reliably tracking drones.

RID standards and regulations aim to increase safety and security for both drone operators and airspace operations. One or more unauthorized drones entering the airspace of an airport, military base or other critical infrastructure facility could result in huge economic and/or physical damage. For example, Gatwick Airport experienced an incident in 2018 where two drones flying near the airport forced authorities to divert approximately 1,000 flights, affecting around 140,000 passengers with a loss of about £800,000.

However, another scenario presents itself: imagine if a malicious user were able to recreate the impact of drones in a no-fly zones—without requiring a real drone. Such a situation could be achieved by injecting fake RID data into a wireless channel to emulate the presence of drones. This could allow threat actors to execute pervasive and low effort drone-based attacks in order to disrupt critical infrastructure services.

Recognizing the importance of drone RID technologies for the future of aviation, Nozomi Networks Labs conducted research on their vulnerabilities and risks. This white paper details the results of our research, which resulted in attack scenarios illustrating how an attacker could forge the presence of drones, inject fake drone trajectories and disrupt RID functionalities on RID protocols. The purpose of this research and of these attack scenarios is to highlight the type of risks involved in the use of these technologies as currently designed.

In the following chapters, we introduce current RID protocols and share our analysis of two of the most widely used: Open Drone ID and DJI's DroneID. Open Drone ID (ODID) is an open-source protocol compliant with most widespread RID specifications. DJI is the leading consumer drone vendor, and their DroneID RID protocol is based on the proprietary Radio Frequency (RF) protocol called OcuSync.

Next, we look at ground station receivers, which receive the signals broadcast by drones and are potential targets of attacks involving RID data spoofing. We found multiple vulnerabilities in the DroneScout ds230, one of the first commercially available fixed ground station receivers compliant with the ODID protocol. We built attack scenarios that then exploited these vulnerabilities to illustrate the security weaknesses in RID protocol receivers.

Finally, Nozomi Networks Labs developed a Software Defined Radio (SDR) OcuSync signal injector and an asynchronous ODID reception and injection framework that allowed us to experiment with RID traffic. We used these tools to create proof-of-concept attack scenarios that showcase how an attacker could forge the presence of drones by injecting fake trajectories and disrupting RID functionalities.

2. Drone Remote Identification (RID)

RID protocols established by the CAA mandate consumer drones (or more technically Unmanned Aircraft Systems [UAS]) to periodically broadcast their telemetry information. Dedicated fixed or mobile ground station receivers then receive and process this information, enabling third-party entities such as law enforcement, critical infrastructure managers, other airspace participants, UAS Traffic Management (UTM) and UAS Service Suppliers (USS) to identify and locate drones and their operators.

The main goal of RID is to bolster the safety, security and responsibility of drone activities, especially when drones might be flying near other aircraft or sensitive locations. This identification is also a deterrent to illicit or unauthorized drone actions like smuggling, espionage or attacks.

Our research in this white paper focuses on Direct or Broadcast RID,[1] its intrinsic weaknesses and the vulnerabilities in compliant receivers to showcase the risks involved in the use of this technology. Broadcast RID specifies the use of wireless technology to broadcast RID data to nearby surroundings; an example scenario is depicted in Figure 1. Any interested observer can use any compliant receiver to capture and visualize RID information and the location of surrounding drones on a map. Captured data can potentially be forwarded to a USS for further elaboration.

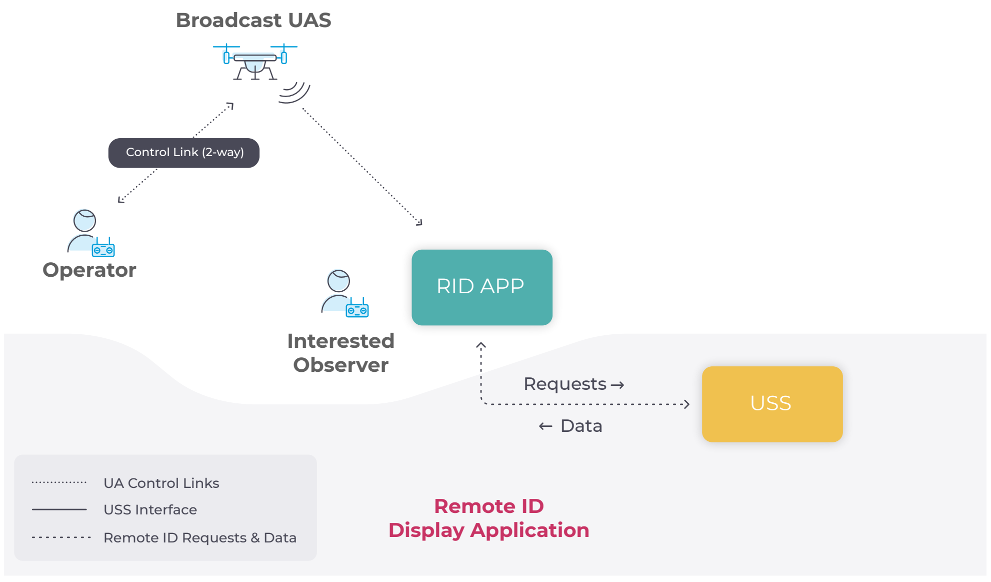


Figure 1 - With Direct/Broadcast RID, any interested observer can use any compliant receiver to capture RID information.

The advantage of Broadcast RID systems is that they do not require any internet connection or cloud backend infrastructure to work. They are designed to make the data available to any close receiver. A second RID mechanism, Network RID, requires the use of cellular networks (4G, 5G, etc.) to transmit telemetry data to an authenticated server which can then be accessed by authorized parties anywhere in the world. Given that Network RID

[1] Specifically, it is called Direct RID in Europe and Broadcast RID in the United States. For simplicity, we use "Broadcast RID" throughout this white paper.

specifications are currently a work in progress and that there was no public implementation available for testing at the time of writing, the research in this white paper focuses on Broadcast RID and will not discuss Network RID.[2]

The type of information required to be broadcast by each drone depends on local regulations, which vary by region. However, as a rule, all drones must periodically transmit the following basic information from takeoff to shutdown:

- Drone ID or UAS ID (where the ID format and the procedure for registering a drone ID is defined by local regulators).
- Drone longitude, latitude, altitude, direction and velocity.
- Control station (or operator) real-time location and elevation.
- Time mark.
- Drone class and operation category.

While there are many types of drones, like fixed-wing, rotary-wing, hybrid, balloon, rocket or others, as a rule, all consumer and commercial drones weighing more than 249 grams (0.55 pounds)[3] must be compliant with RID regulations. Exceptions apply to certain drone models operated according to specific guidelines or for government, military and other authorized entities. Figure 2 provides examples of drones that must be compliant with RID rules (In Scope) and drones that are exempt (Out of Scope).

A drone can either support RID capabilities through a built-in module installed or activated by the manufacturer[4] or with an RID add-on device consisting of transmitters that attach to a drone's body. The latter is especially useful for pilots with older or custom-built drones that cannot be upgraded to support RID policies and protocols.

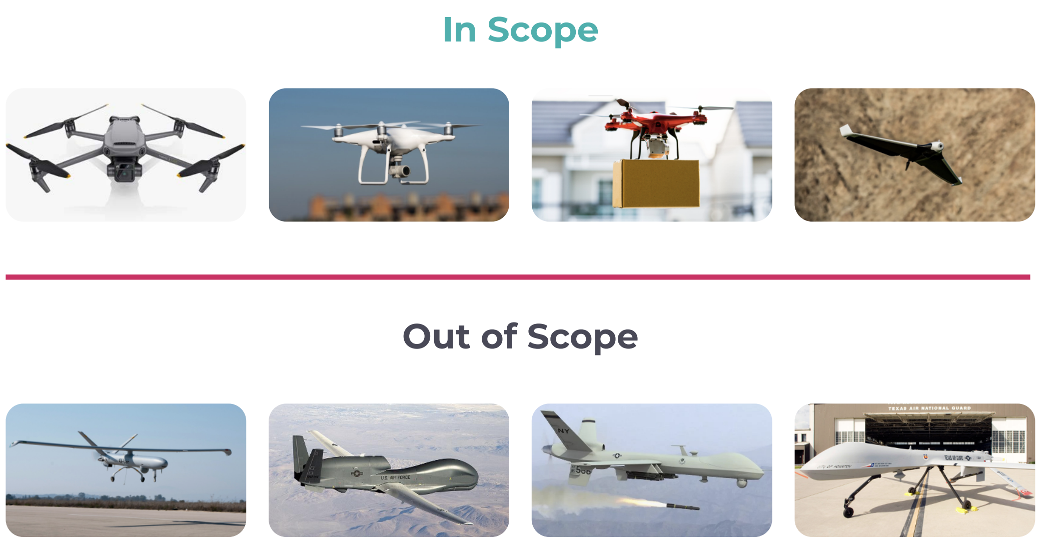


Figure 2 - Drones that are in-scope and out-of-scope for RID compliance.

[2] Network RID will eventually become mandatory in Europe under the U-Space program, while, at the time of writing, there is no Network RID mandate in the US. It is considered a technically more complicated approach than the Direct/Broadcast RID, as Network RID requires a reliable and secure channel.
[3] Some vendors are also enabling RID functionality on drones weighing less then 249 grams, like DJI's Mini 3 Pro. Many regulations also state that once RID is turned on, it cannot be turned off again (e.g., through a firmware downgrade).
[4] Called Standard RID drone in U.S. and C-class certified drone in the EU. Currently, the IETF DRIP (Drone Remote ID Protocol) task force is working to define mechanisms to support security in the context of the ASTM RID.

2.1 RID Background

2.1.1 RID Standards and Regulations

Europe, the U.S. and Japan are currently the most active regions in terms of developing Direct/Broadcast RID policies and rules. These three regions share similar high-level architecture and RID system rules as all three allow the usage of the same RID protocol reference implementation, Open Drone ID.

To make sense of all available RID documentation and information, one must focus on three aspects:

- **Technical standards:** used to define RID transmission methods and message formats. The two main standards today are the ASTM[5] F3411 in the U.S. and the ASD-STAN prEN4709-02 in the EU. Both rely on wireless protocols in the unlicensed spectrum to broadcast the identification and telemetry data from UAs to ground observers. In particular, they define transport methods over Wi-Fi and Bluetooth.
- **Regulations:** used by local regulation authorities, like the Federal Aviation Administration (FAA) in the U.S. and the European Aviation Safety Agency (EASA) in the EU, to specify RID rules and policies that tailor the technical standards according to specific local regulatory requirements. For instance, while the standards label certain fields as "optional", local regulations might deem some of them necessary. In such cases, the regulation documents can indicate specific deviations from the standards.
- **Reference implementations:** example implementations of a technical standard designed to be flexible enough to be adapted to different regulations. They facilitate interoperability between receivers and transmitters and, by having a single, carefully tested reference implementation shared among vendors, they help reduce security bugs in the code. For Broadcast RID specifications, the most widely used reference implementation is maintained by the Open Drone ID (ODID) project, which we discuss in Chapter 3. One of the first commercially available fixed ground station RID receivers, produced by BlueMark and called DroneScout ds230, is internally based on the ODID reference implementation library and is discussed in Chapter 4.

Being so recent, these regulations, standards and reference implementations are continuously evolving. We advise interested readers to refer to official documentation from local authorities or **drone-remote-id.com**, which summarizes the most recent news on this subject.

2.1.2 RID Protocol Security Flaws

The goal of UAS-RID regulations is to improve the physical safety and security of airspace by providing immediately actionable drone telemetry data to regulators and law enforcement organizations. However, this industry has not yet succeeded in creating reliable workable schemes for the global distribution of cryptographic keys. Leaving the protocols open was the only choice to guarantee that anyone with a qualified ground station could receive, decode and interpret RID signals, although the need to introduce cyber security measures to protect telemetry data was clear to the security professionals involved in RID protocol design. While Trusted Platform Modules (TPM) can be used to safely store keys on drones, this not only increases a drone's cost but still does not solve the problem of distributing the keys.

This conflict between physical and cyber security requirements, together with the fact that drone vendors

[5] ASTM International: **astm.org**.

were forced to comply with UAS-RID rules in a short period of time, led to the design and deployment of RID protocols that do not protect telemetry data confidentiality and integrity, and do not provide telemetry data authentication (Figure 3). This leaves current RID protocols open to traffic injection attacks like those we discuss in the attack scenarios in Chapter 6. Since the RID data is not authenticated, the receivers have no way to differentiate between real RID data transmitted by a real drone and forged data transmitted by a malicious user.

While we are aware that insecure RID is not a trivial problem to solve and that it will require the time and collective effort of security experts around the world, we believe it is important to bring awareness to issues that may arise from a global deployment of RID protocols as currently designed.

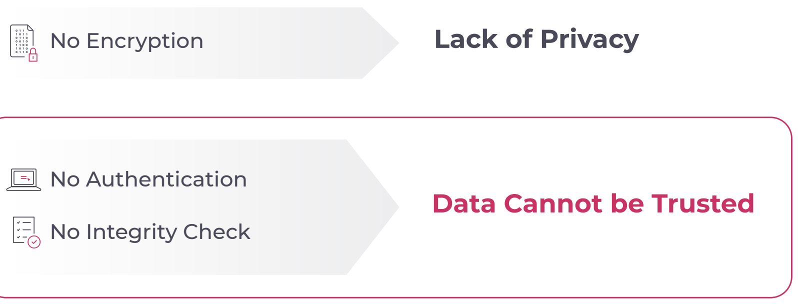


Figure 3 - Current RID protocols lack encryption, authentication and integrity checks, meaning this data cannot be trusted.

2.2 DJI and the OcuSync-Based DroneID Protocol

In the second half of 2022, DJI, the leading drone vendor with over 70% of the global market share, introduced support for RID regulation and policies into its drones though the adoption of the ODID protocol. However, prior to the introduction of RID rules, DJI designed and deployed DroneID, its own proprietary RID protocol. All DJI drones broadcast DroneID telemetry data using DJI's proprietary radio protocol, OcuSync.[6] Given that custom hardware is required to receive OcuSync signals, DJI also started producing and selling proprietary ground station receivers, Aeroscopes, specifically designed to detect DJI drones in surrounding areas and visualize them on a map.

The research presented here relied on the DJI Mini 3 Pro drone model with firmware version 01.00.0150. We used this firmware to perform the experiments, capture RID traffic and test the proof-of-concept attack scenarios. Nozomi Networks Labs also had the opportunity to test one of DJI's Aeroscope appliances and analyze its behavior from a security perspective. Chapter 5 provides more details about OcuSync, DroneID RID protocol and the Aeroscope.

DJI's proprietary RID protocol suffers from the same type of weaknesses as ODID, as it does not protect the confidentiality and integrity of drone telemetry data and does not provide any form of authentication. This makes DroneID subject to the same type of attacks scenarios that target ODID, which we describe in Chapter 6.

[6] Early generations of DJI drones used 5MHz channels with Wi-Fi modulation to broadcast telemetry data. This channel is no longer supported in recent models.

3. Open Drone ID (ODID) Protocol

The Federal Aviation Administration (FAA) in the U.S., Aerospace and Defense Industries Association of Europe (ASD) and other regulatory agencies around the world are moving towards requiring RID for most drones operating in their airspace. The ODID protocol aims to provide a standardized, open-source solution for this requirement[7] by means of a reference implementation library for the protocol and various example tools for testing reception and transmission of ODID messages.

Understanding how the ODID protocol works is necessary to implement the ODID injection framework and develop the attack scenarios that we introduce in Chapter 6. For this reason, this chapter describes the main features of the ODID protocol, its supported communication methods, the format of the specified messages and an example of real ODID (RID) data transmitted by a consumer drone.

3.1 ODID Protocol Background

ODID is an initiative aiming to create an affordable and reliable RID system for UAS that allows receivers and ground stations within range to identify and locate drones and their operators. The transmitted data may include information such as a drone's ID, current location, direction, speed, operator information and other relevant telemetry data.

As mandated by recent RID regulations and policies, telemetry data broadcast by drones using the format specified by the ODID project can be used by the general public, law enforcement, critical infrastructure managers, Air Traffic Control (ATC) systems or even other drones to improve situational awareness of the surrounding airspace. The accessibility of this protocol thereby increases both safety and security and creates accountability for drone operators (Figure 4). This system is in some ways analogous to the transponder technology used in manned aviation.

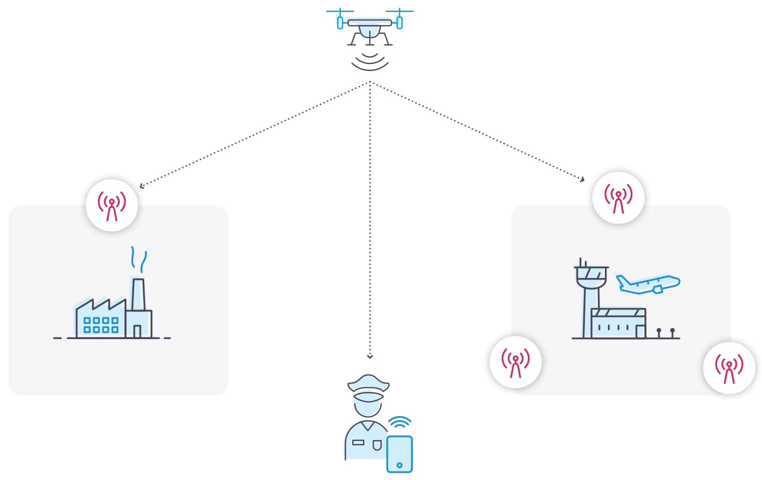


Figure 4 - Open Drone ID broadcasts can be received by law enforcement officers, critical infrastructure managers and Air Traffic Control systems.[8]

[7] The project is hosted in a dedicated GitHub organization: **github.com/opendroneid**.

Most RID regulations around the world divide transmitted information into static and dynamic data, where the static data can be broadcast less frequently than the dynamic data. The ODID project provides the flexibility to create different types of messages transporting different types of information, making it possible to create messages that only contain static data. For example, Figure 5 shows a scenario where a drone broadcasts one message transporting static data for every three messages transporting dynamic data. The static data includes the drone registration number and flight information, which does not change during a single flight. Meanwhile, dynamic data can include the drone's location (latitude and longitude), which changes with high frequency and thus must be broadcast more often to allow an ODID receiver (e.g., law enforcement officer) to have accurate location information.

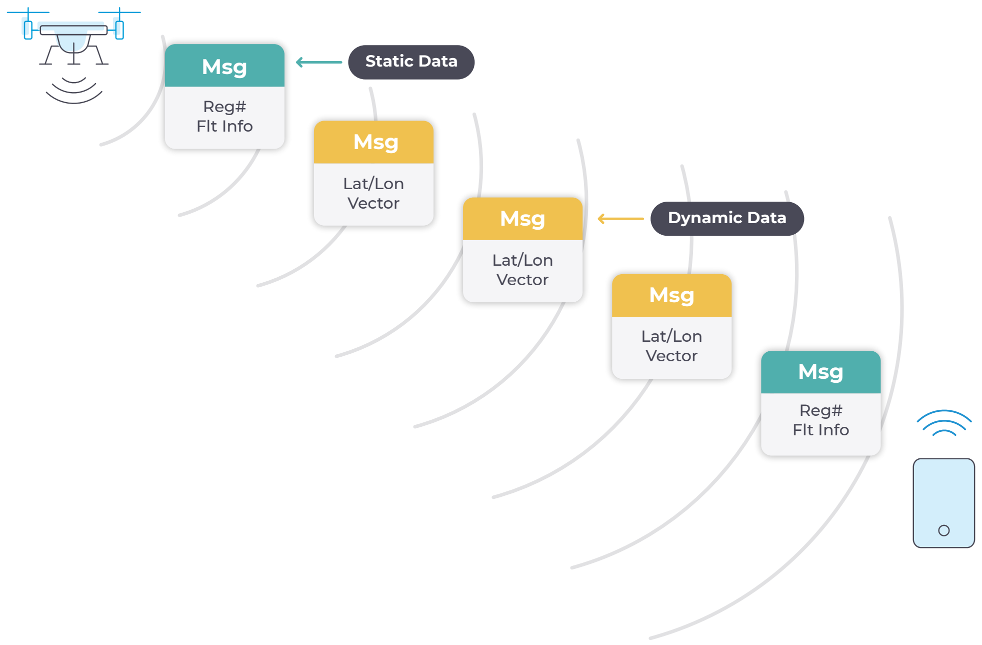


Figure 5 - Dynamic RID data, such as a drone's location, is often shared more frequently than static RID data like operator information.[9]

The decision to split static and dynamic RID information between two different message types transmitted with different frequencies is up to the developer of the RID module running on the drone. For example, a drone vendor could decide to develop RID functionalities that include both static and dynamic data in all the broadcast RID messages and then configure a transmission frequency that is enough to satisfy the timing requirements of the dynamic data.[8]

[8] The disadvantage of this approach is that all the ODID messages transmitted by the drone will be a few bytes longer than necessary.

Messages transmitted by drones are always "connectionless advertisements", meaning that they do not require acknowledgment from the receiver. Considering the unreliable nature of wireless transmission as a medium (i.e., a transmitted message is not guaranteed to be received at destination), frequently transmitting dynamic data increases the probability that enough messages will reach their destination (i.e., RID receivers).

ODID's reference implementation core library allows developers to implement both the transmission and reception functionalities of an RID system. The transmission logic, which is in charge of broadcasting RID messages, is typically installed on drones or on add-on modules[9] that must be mounted on drones. The reception logic, in charge of monitoring wireless channels and decoding the received RID messages, is typically installed on receiving equipment like mobile phones, tablets or dedicated fixed ground stations.

The Open Drone ID project focuses exclusively on the Broadcast RID method[10] with the purpose of specifying exactly how a drone must transmit each RID message (i.e., how the data contained in the messages is encoded and packed) so that a compatible receiver system is able to receive, decode and interpret those messages. The standard specifies that ODID messages containing RID data must be transmitted using either Wi-Fi or Bluetooth, making it possible to receive RID data using inexpensive hardware already available on the market. No other wireless technology is currently supported or specified by the standard.

3.1.1 ODID Communication Methods and Technologies

ODID messages can be transmitted using Wi-Fi (IEEE 802.11), Bluetooth or both. When Wi-Fi is used, the ODID transmitter (i.e., the drone) can use either the 2.4GHz or 5GHz band, although the former is more common because it provides a bigger expected transmission range and is compatible with more types of receiving hardware. Different broadcast modes are supported for both Wi-Fi and Bluetooth: Wi-Fi (IEEE 802.11) Neighbor Awareness Networking (NaN), Wi-Fi beacon, Bluetooth Low Energy (BLE) (Bluetooth 4.x compatible) Advertisements and Bluetooth 5.0 Extended Advertisements.

This white paper focuses exclusively on the Wi-Fi beacon[11] broadcasting method as, at the time this research was carried out, it was the only ODID broadcasting method supported by the first RID compliant drone models available on the market. In short, with Wi-Fi beacon ODID broadcasting, the transmitter encapsulates ODID telemetry messages in a vendor specific (0xdd) Information Element (IE), with the Organizationally Unique Identifier (OUI) set to ASD-STAN (0xfa 0x0b 0xbc 0x0d) and inserts them within standard Wi-Fi beacon frames.

[9] If a drone's hardware does not support the transmission methods required to be compliant with RID standards, regulations and policies, then an external add-on module must be used. See Chapter 2.
[10] For the Network RID mechanism there are currently several alternative projects (although they do not appear to be well maintained), namely: **github.com/interuss** and **github.com/uastech/standards**.
[11] Wi-Fi management (type 0) beacon (sub-type 8) frames are the frames periodically transmitted by a Wi-Fi Access Point (AP) to advertise the presence and capabilities of a Wi-Fi network to any nearby Wi-Fi device.

3.2 ODID Message Format and Encoding

ODID protocol message format defines two types of "macro" messages: message blocks and message packs. Message blocks represent the minimal unit of ODID broadcast information and are the messages that contain the actual RID information (and other drone-related data). Message packs are used to aggregate multiple ODID message blocks into a single transmission.

While the specific wireless technology used for broadcasting does not affect message format, it does affect the mechanism used for encapsulating and transporting the ODID message. In the case of Wi-Fi beacon broadcasting method, as we already briefly mentioned, the ODID data is transported in a vendor-specific Information Element (IE) field. This IE field is typically found towards the end of the beacon frames and its maximum length is 255 bytes. ODID message blocks are encoded in an ODID message pack and the resulting Vendor Specific IE is formatted as described in Table 1.[12]

Byte Offset	Length (bytes)	Value	Description
0	1	0xdd	0xdd represents a vendor-specific Information Element
1	1	0x2 – 0xFF	Length of the Information Element (maximum 255 bytes)
2	4	0xFA 0x0B 0xBC 0x0D	ASD-STAN identifier
6	1	0x00 – 0xFF	ODID message counter that increments with each ODID message pack sent and resets back to 0 after 0xFF is reached
7	3 + N * 25	Variable	Message pack header + message blocks. A maximum of 9 message blocks (N) are allowed.

Table 1 - Wi-Fi beacon: Format of the Information Element containing ODID-specific data.

[12] We refer the reader to the official Open DroneID specifications for more details on how ODID message encapsulation works with other broadcasting methods: **github.com/opendroneid/specs**.

3.2.1 ODID Message Blocks

An ODID message block is always 25 bytes in length—the message can be padded with null bytes if needed. The block starts with a 1-byte header followed by 24 bytes of data whose format depends on the block message type which is specified in a dedicated 4-bit field in the header as shown in Table 2.

Block Header		Block Data
Message Type (4 bits)	Protocol Version (4 bits)	Message fields base on Message Type
0x1 – 0xF	0x0	<Message Data>

Table 2 - ODID message block format.

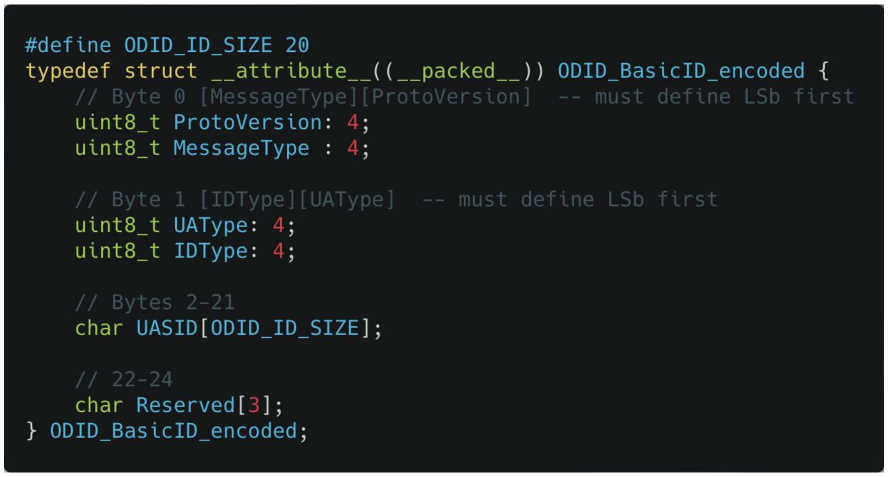
The 16- or 32-bit numerical fields that can be present in the message block data are always transmitted in little endian order, while all other types of data (e.g., non-magnitude values, strings, IDs) are transmitted in big endian order. ODID specifications provide different types of message block types based on the kind of information they contain.

The list of currently existing ODID message block types is provided in Table 3. A message type of value 0xFF is considered invalid and, as we will see later, the message type 0xF is used for ODID message packs. In the following sections we will investigate the most important block message types and their fields. We refer the reader to the official ODID specifications for message blocks not discussed here.

Message Type	Message Name	Description
0x0	Basic ID	Provides ID for the UAS, characterizes the type of ID and identifies the type of UAS
0x1	Location/Vector	Provides location, altitude, direction and speed of the UAS
0x2	Authentication	Optional message that provides authentication data for the UAS
0x3	Self-ID	Optional message that can be used by operators to identify themselves and the purpose of an operation
0x4	System	Identifies the location of the operator
0x5	Operator	Provides the operator ID

Table 3 - ODID message block types.

The Basic ID message block is used to identify the drone and includes the ID Type, UAS Type and the Unique ID. Figure 6 shows its packed form in the reference implementation.[13] The Unmanned Aerial System Identifier (UASID) is maximum 20 bytes and the actual format and procedure for obtaining one are country-specific and defined by local regulators. UASID values can be used to identify a specific drone in a particular geographic region.



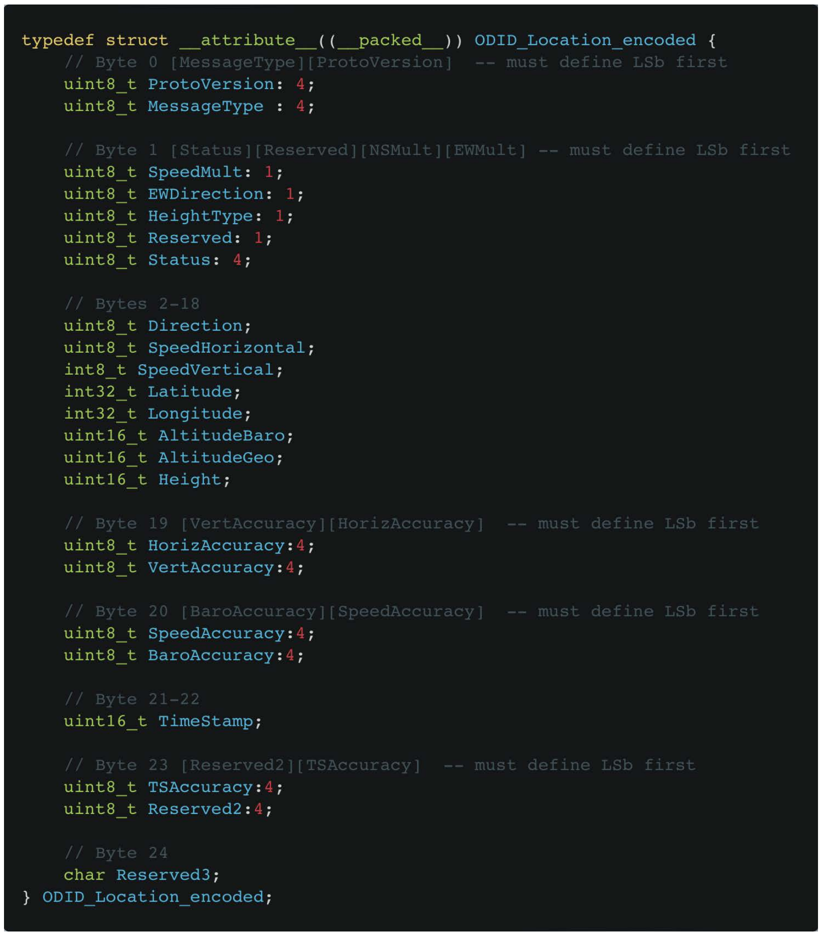
```
#define ODID_ID_SIZE 20
typedef struct __attribute__((__packed__)) ODID_BasicID_encoded {
    // Byte 0 [MessageType][ProtoVersion]  -- must define LSb first
    uint8_t ProtoVersion: 4;
    uint8_t MessageType : 4;

    // Byte 1 [IDType][UAType]  -- must define LSb first
    uint8_t UAType: 4;
    uint8_t IDType: 4;

    // Bytes 2-21
    char UASID[ODID_ID_SIZE];

    // 22-24
    char Reserved[3];
} ODID_BasicID_encoded;
```

Figure 6 - ODID Basic ID message block.

The Location/Vector block message, among other things, provides the location, altitude, direction and speed of the drone. Figure 7 shows its packed form in the reference implementation.[14]

[13] See the reference implementation: **github.com/opendroneid/opendroneid-core-c/blob/6f0bc76fddb11730ed280582a4b878979b499b66/libopendroneid/opendroneid.h#L423**

[14] See the reference implementation: **github.com/opendroneid/opendroneid-core-c/blob/6f0bc76fddb11730ed280582a4b878979b499b66/libopendroneid/opendroneid.h#L439**



```
typedef struct __attribute__((__packed__)) ODID_Location_encoded {
    // Byte 0 [MessageType][ProtoVersion]  -- must define LSb first
    uint8_t ProtoVersion: 4;
    uint8_t MessageType : 4;

    // Byte 1 [Status][Reserved][NSMult][EWMult] -- must define LSb first
    uint8_t SpeedMult: 1;
    uint8_t EWDirection: 1;
    uint8_t HeightType: 1;
    uint8_t Reserved: 1;
    uint8_t Status: 4;

    // Bytes 2-18
    uint8_t Direction;
    uint8_t SpeedHorizontal;
    int8_t SpeedVertical;
    int32_t Latitude;
    int32_t Longitude;
    uint16_t AltitudeBaro;
    uint16_t AltitudeGeo;
    uint16_t Height;

    // Byte 19 [VertAccuracy][HorizAccuracy]  -- must define LSb first
    uint8_t HorizAccuracy:4;
    uint8_t VertAccuracy:4;

    // Byte 20 [BaroAccuracy][SpeedAccuracy]  -- must define LSb first
    uint8_t SpeedAccuracy:4;
    uint8_t BaroAccuracy:4;

    // Byte 21-22
    uint16_t TimeStamp;

    // Byte 23 [Reserved2][TSAccuracy]  -- must define LSb first
    uint8_t TSAccuracy:4;
    uint8_t Reserved2:4;

    // Byte 24
    char Reserved3;
} ODID_Location_encoded;
```

Figure 7 - ODID Location/Vector message block.

In order to save space and allow all the location data to be packed into the 24 bytes available in the message block payload, some of the fields in the Location/Vector message block (and also System message block which we discuss further on) have some encoding techniques to either compress the data or allow for more optimal and precise resolutions.

For example, the latitude and longitude values are 32-bit signed integer values where the actual latitude/longitude value is encoded by multiplying it by 10^7.

This means that to decode the value at the receiver, the Latitude and Longitude field values must be divided by 10^7. Let's consider Nozomi Networks' offices in Mendrisio, Switzerland, which are located more or less at latitude 45.878780 and longitude 8.979026. Those two values will be encoded in the Latitude and Longitude fields of the Location/Vector message block respectively as 458787800 and 89790260. We refer the reader to official ODID specifications[15] for details on how the various fields are encoded in the message blocks.

[15] See **github.com/opendroneid/specs** for official ODID specifications.

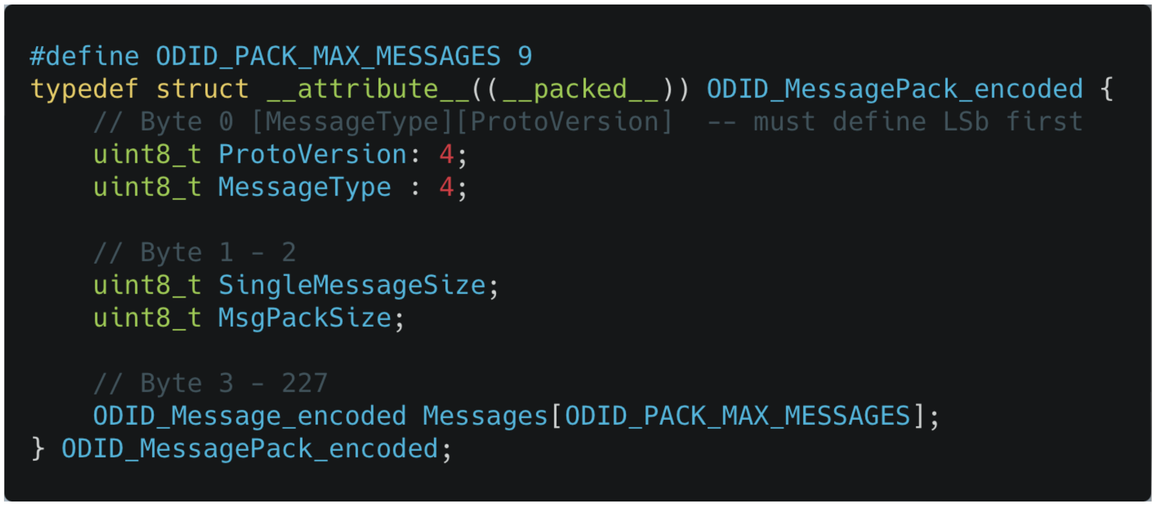
3.2.2 ODID Message Packs

Multiple message blocks can be grouped into a single message and encoded into an ODID message pack, whose format is shown in Table 4.

The message pack always starts with a 3-byte header which, among other things, includes the size of a single message block (which is always 25 bytes in the current version of ODID protocol) and the number *N* of message blocks that follow the header. The packed form of the ODID message pack in Figure 8 shows that the current ODID protocol specifies that 9 is maximum number *N* of message blocks transported by a message pack.[16]

ODID Message Pack						
Message Pack Header				Message Block	Message Block	...
Message Type (4 bits)	Version (4 bits)	Single Message Size (always 0x16) (1 byte)	N. Messages in Pack (N) (1 byte)	25 bytes	25 bytes	...

Table 4 - ODID message pack.



```
#define ODID_PACK_MAX_MESSAGES 9
typedef struct __attribute__((__packed__)) ODID_MessagePack_encoded {
    // Byte 0 [MessageType][ProtoVersion]  -- must define LSb first
    uint8_t ProtoVersion: 4;
    uint8_t MessageType : 4;

    // Byte 1 - 2
    uint8_t SingleMessageSize;
    uint8_t MsgPackSize;

    // Byte 3 - 227
    ODID_Message_encoded Messages[ODID_PACK_MAX_MESSAGES];
} ODID_MessagePack_encoded;
```

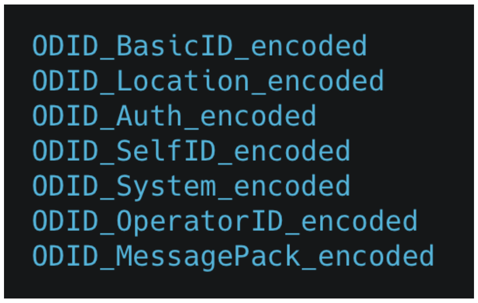
Figure 8 - ODID message pack Definition.

[16] This is due to the size limit of Wi-Fi Beacon IE length.

3.2.3 Open Drone ID Reference Implementation Library

The ODID project's reference implementation library implements the encoding and decoding functionalities for all the messages specified by the ODID protocol. For each ODID message (block and pack), as shown in Figure 9, the library provides a data structure representing the encoded version of the messages (i.e., the messages that are actually transmitted and received over the air).[17]

This library is particularly important because, as we will see in the following chapters, it is used by both the BlueMark DroneScout ds230 RID ground station receiver and the Open Drone ID reception/injection framework developed by Nozomi Networks Labs for implementing the proof-of-concept attack scenarios presented later in this white paper. The reference implementation library is open source and available on GitHub and exposes data structure types representing all the possible messages provided by the ODID protocol.[18]



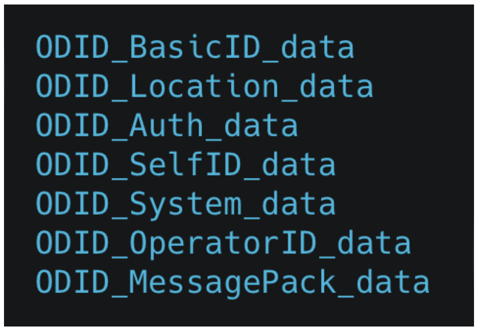
```
ODID_BasicID_encoded
ODID_Location_encoded
ODID_Auth_encoded
ODID_SelfID_encoded
ODID_System_encoded
ODID_OperatorID_encoded
ODID_MessagePack_encoded
```

Figure 9 - Reference implementation library encoded ODID message data structures.

For each of these data structures, the library provides the corresponding non-encoded version (also called non-packed or normative form) (Figure 10). These are the data structures actually used in the code that implements the ODID transmitter or receiver logic because they are easier to use for parsing and elaborating drone data. For example, strings like UASID which are not null terminated in the encoded data structures are instead null terminated in the normative form data structures.

[17] The encoded versions of the ODID messages are what is passed to the lower network layer to be transmitted over any of the broadcast methods in Open Drone ID and what is received from the lower network layer during reception.

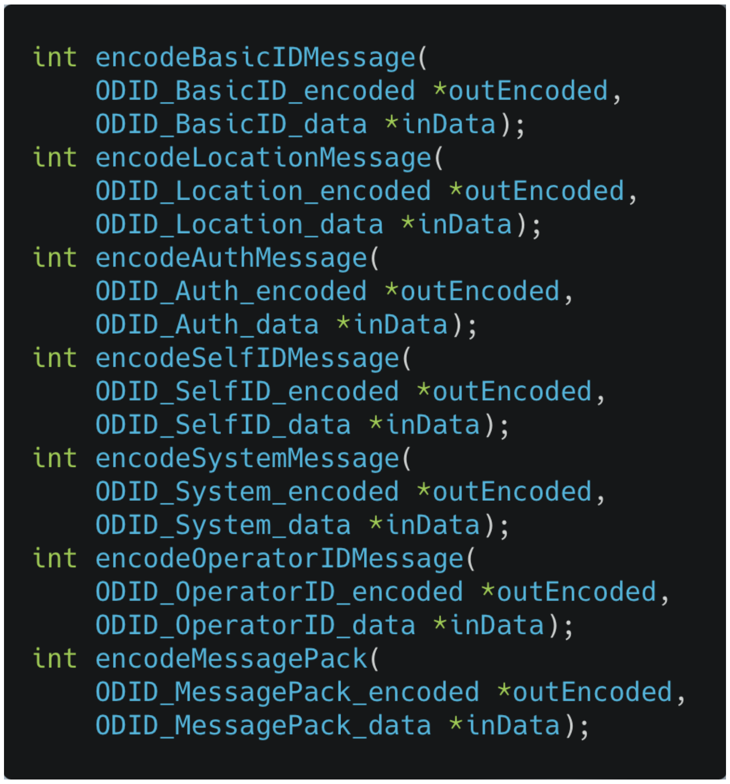
[18] See **github.com/opendroneid/opendroneid-core-c**. This repository provides a C-code function library for encoding and decoding (packing/unpacking) Open Drone ID messages, as the format is defined in the ASTM F3411 RID and the ASD-STAN prEN 4709-002 Direct/Broadcast RID specifications.



```
ODID_BasicID_data
ODID_Location_data
ODID_Auth_data
ODID_SelfID_data
ODID_System_data
ODID_OperatorID_data
ODID_MessagePack_data
```

Figure 10 - Reference implementation library normative ODID message data structures.

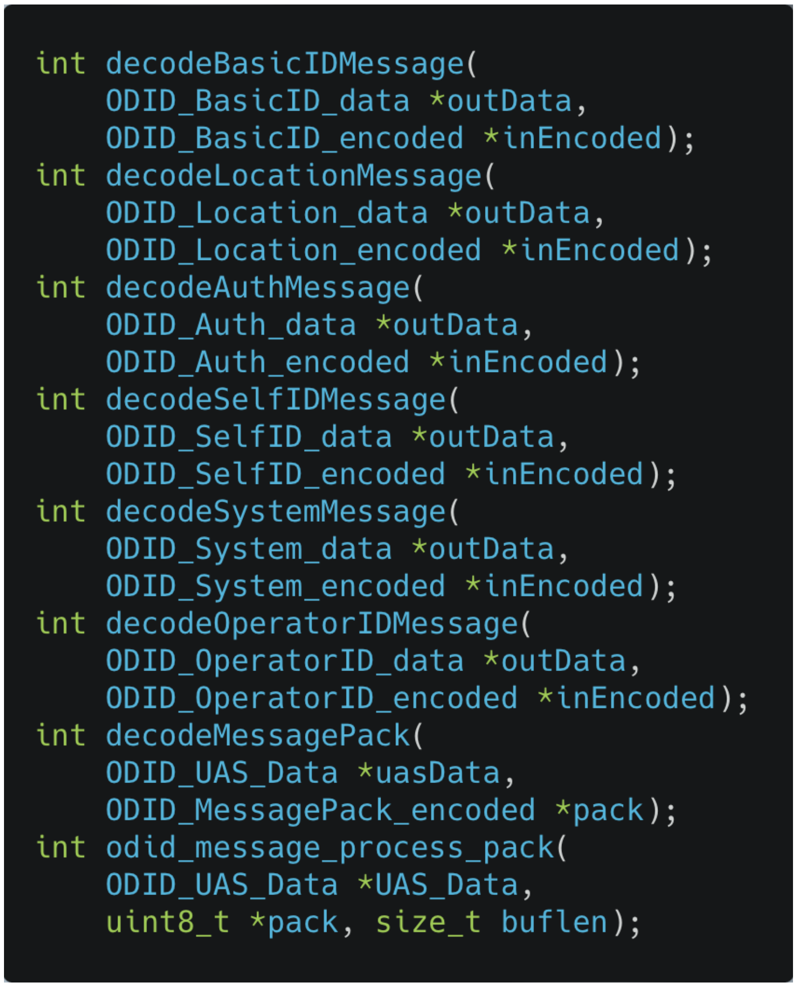
The library also provides the encoding functions for mapping the normative form data structures representing the ODID messages into their corresponding encoded forms. For example, the function *encodeBasicIDMessage(…)* maps the normative form of the Basic ID ODID message block *(ODID_BasicID_data)* into its corresponding encoded version *(ODID_BasicID_encoded)* which can be used to transmit the data. The list of all available encoding functions is shown in Figure 11.



```
int encodeBasicIDMessage(
    ODID_BasicID_encoded *outEncoded,
    ODID_BasicID_data *inData);
int encodeLocationMessage(
    ODID_Location_encoded *outEncoded,
    ODID_Location_data *inData);
int encodeAuthMessage(
    ODID_Auth_encoded *outEncoded,
    ODID_Auth_data *inData);
int encodeSelfIDMessage(
    ODID_SelfID_encoded *outEncoded,
    ODID_SelfID_data *inData);
int encodeSystemMessage(
    ODID_System_encoded *outEncoded,
    ODID_System_data *inData);
int encodeOperatorIDMessage(
    ODID_OperatorID_encoded *outEncoded,
    ODID_OperatorID_data *inData);
int encodeMessagePack(
    ODID_MessagePack_encoded *outEncoded,
    ODID_MessagePack_data *inData);
```

Figure 11 - Open Drone ID reference implementation library encoding functions.

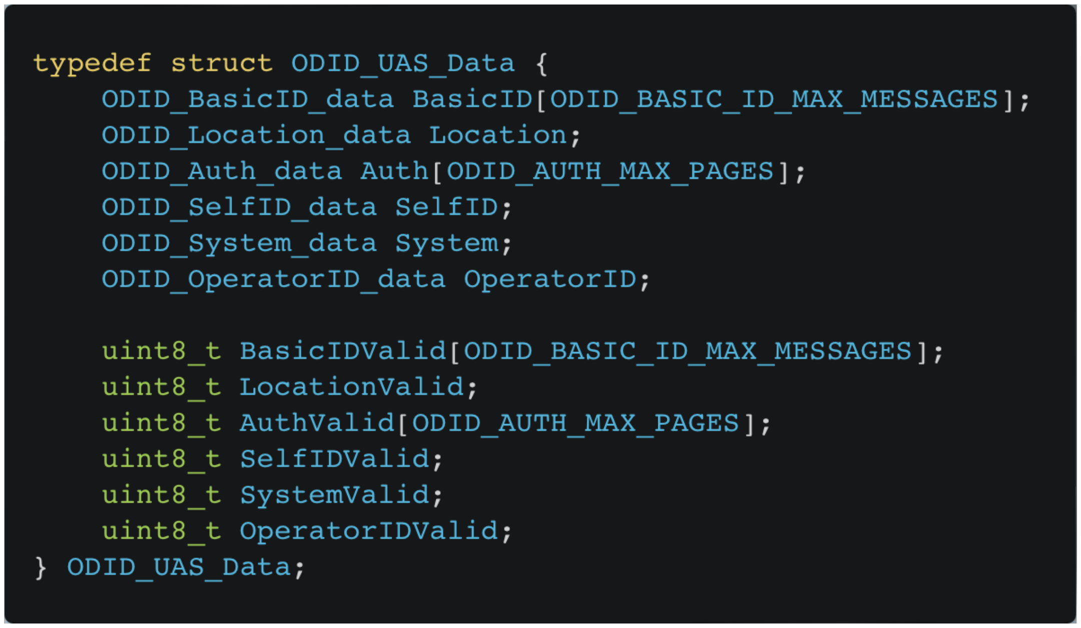
Figure 12 shows the corresponding decoding functions which are used to map data structures representing the encoded ODID messages into their corresponding non-packed form. For example, the function *decodeLocationMessage(...)* maps the encoded form of the Location/Vector ODID message block *(ODID_Location_encoded)* into its corresponding normative form *(ODID_Location_data)*.



```
int decodeBasicIDMessage(
    ODID_BasicID_data *outData,
    ODID_BasicID_encoded *inEncoded);
int decodeLocationMessage(
    ODID_Location_data *outData,
    ODID_Location_encoded *inEncoded);
int decodeAuthMessage(
    ODID_Auth_data *outData,
    ODID_Auth_encoded *inEncoded);
int decodeSelfIDMessage(
    ODID_SelfID_data *outData,
    ODID_SelfID_encoded *inEncoded);
int decodeSystemMessage(
    ODID_System_data *outData,
    ODID_System_encoded *inEncoded);
int decodeOperatorIDMessage(
    ODID_OperatorID_data *outData,
    ODID_OperatorID_encoded *inEncoded);
int decodeMessagePack(
    ODID_UAS_Data *uasData,
    ODID_MessagePack_encoded *pack);
int odid_message_process_pack(
    ODID_UAS_Data *UAS_Data,
    uint8_t *pack, size_t buflen);
```

Figure 12 - Open Drone ID reference implementation library decoding functions.

There is a particularly important decoding function called *odid_message_process_pack(...)*[19] which is used to parse a raw buffer of bytes (uint8_t *pack) which must point to an *ODID_MessagePack_encoded* and map it into an *ODID_UAS_Data* whose definition is shown in Figure 13. This data structure contains the normative form of all the data contained in the parsed ODID message pack.



```
typedef struct ODID_UAS_Data {
    ODID_BasicID_data BasicID[ODID_BASIC_ID_MAX_MESSAGES];
    ODID_Location_data Location;
    ODID_Auth_data Auth[ODID_AUTH_MAX_PAGES];
    ODID_SelfID_data SelfID;
    ODID_System_data System;
    ODID_OperatorID_data OperatorID;

    uint8_t BasicIDValid[ODID_BASIC_ID_MAX_MESSAGES];
    uint8_t LocationValid;
    uint8_t AuthValid[ODID_AUTH_MAX_PAGES];
    uint8_t SelfIDValid;
    uint8_t SystemValid;
    uint8_t OperatorIDValid;
} ODID_UAS_Data;
```

Figure 13 - Open Drone ID reference implementation library UAS aggregate data structure.

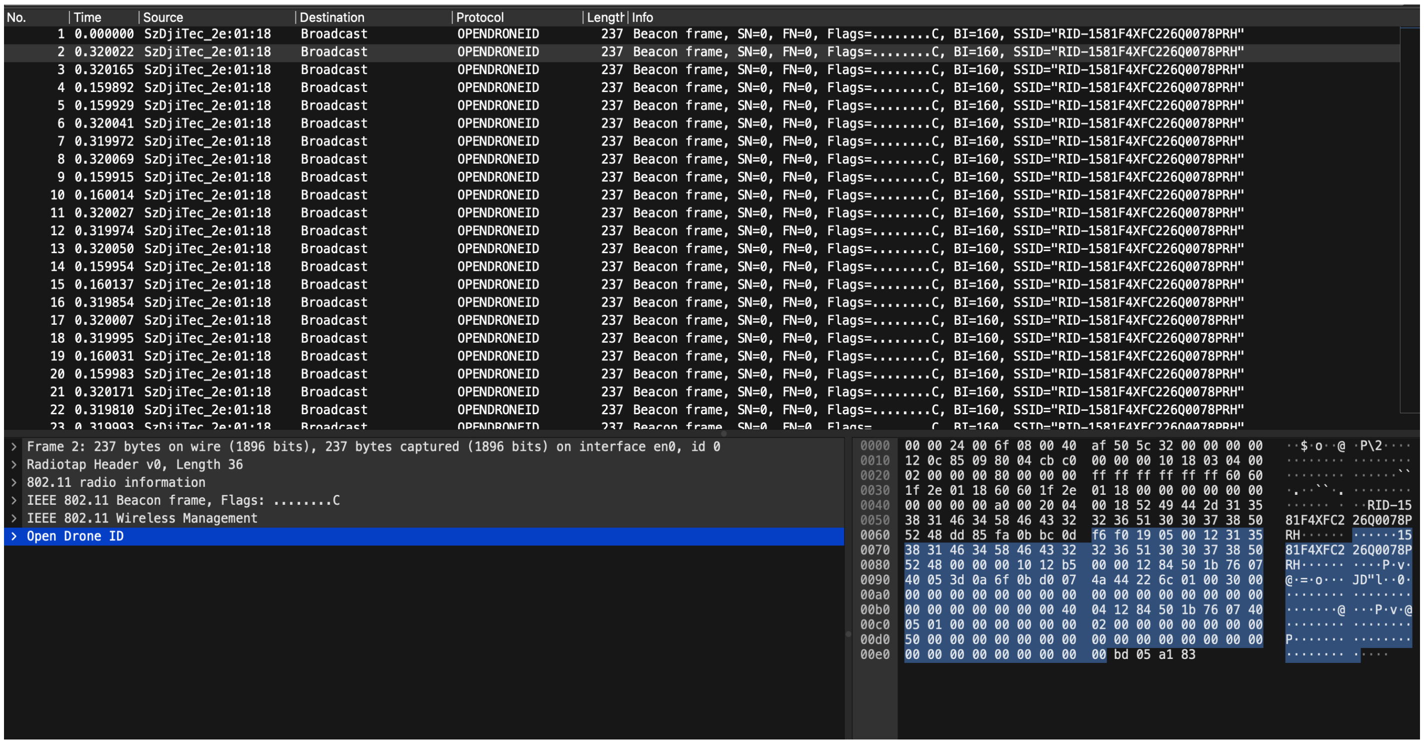
[19] Internally this function uses decodeMessagePack(...).

3.3 Wi-Fi ODID Traffic Capture Example

Now that we have seen how ODID is implemented in the reference library, we can look at an example of what real-world ODID traffic generated by a consumer drone looks like when monitoring Wi-Fi channels with nearby ODID transmitting drones. Figure 14 presents an example of Open Drone ID (RID) data capture on Wi-Fi channel 6 transmitted by a DJI Mini 3 Pro running firmware version 01.00.0150, which we used throughout this research project.[20] The Wireshark screenshot shows the ODID messages (protocol OPENDRONEID) transmitted by the drone.

Note that:

- The drone uses Wi-Fi beacon frames as its broadcasting method;
- In the beacon frames the SSID is set to the string "RID-" followed by the UAS ID of the drone[21];
- The ODID messages containing RID data are sent with a period that is either ~160ms or ~320ms.



```
No.  Time     Source             Destination  Protocol     Length Info
   1 0.000000 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
   2 0.320022 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
   3 0.320165 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
   4 0.159892 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
   5 0.159929 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
   6 0.320041 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
   7 0.319972 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
   8 0.320069 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
   9 0.159915 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  10 0.160014 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  11 0.320027 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  12 0.319974 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  13 0.320050 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  14 0.159954 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  15 0.160137 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  16 0.319854 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  17 0.320007 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  18 0.319995 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  19 0.160031 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  20 0.159983 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  21 0.320171 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  22 0.319810 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"
  23 0.319993 SzDjiTec_2e:01:18  Broadcast    OPENDRONEID    237 Beacon frame, SN=0, FN=0, Flags=........C, BI=160, SSID="RID-1581F4XFC226Q0078PRH"

> Frame 2: 237 bytes on wire (1896 bits), 237 bytes captured (1896 bits) on interface en0, id 0
> Radiotap Header v0, Length 36
> 802.11 radio information
> IEEE 802.11 Beacon frame, Flags: ........C
> IEEE 802.11 Wireless Management
> Open Drone ID

0000  00 00 24 00 6f 08 00 40  af 50 5c 32 00 00 00 00   ··$·o··@ ·P\2····
0010  12 0c 85 09 80 04 cb c0  00 00 00 10 18 03 04 00   ········ ········
0020  02 00 00 00 80 00 00 00  ff ff ff ff ff ff 60 60   ········ ······``
0030  1f 2e 01 18 60 60 1f 2e  01 18 00 00 00 00 00 00   ·.··``·. ········
0040  00 00 00 00 a0 00 20 04  00 18 52 49 44 2d 31 35   ········ ··RID-15
0050  38 31 46 34 58 46 43 32  32 36 51 30 30 37 38 50   81F4XFC2 26Q0078P
0060  52 48 dd 85 fa 0b bc 0d  f6 f0 19 05 00 12 31 35   RH······ ······15
0070  38 31 46 34 58 46 43 32  32 36 51 30 30 37 38 50   81F4XFC2 26Q0078P
0080  52 48 00 00 00 10 12 b5  00 00 12 84 50 1b 76 07   RH······ ····P·v·
0090  40 05 3d 0a 6f 0b d0 07  4a 44 22 6c 01 00 30 00   @·=·o··· JD"l··0·
00a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
00b0  00 00 00 00 00 00 00 40  04 12 84 50 1b 76 07 40   ·······@ ···P·v·@
00c0  05 01 00 00 00 00 00 00  02 00 00 00 00 00 00 00   ········ ········
00d0  50 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   P······· ········
00e0  00 00 00 00 00 00 00 00  00 bd 05 a1 83            ········ ·····
```

Figure 14 - DJI Mini 3 Pro transmits ODID messages with a period of ~160 or ~320 ms.

[20] In this firmware version DJI has enabled ODID (RID) transmission by default, independent of the geographic location of the drone. So, for example, it was possible to perform these experiments in Europe where the RID regulation was not yet mandatory.
[21] With DJI drones, at the time of writing, it was possible to configure a drone's UAS ID through the DJI Fly app.

The ODID message periodicity is reflected in the corresponding I/O graph (Figure 15) where we can see that the tested drone transmitted between 3 and 5 ODID messages per second.

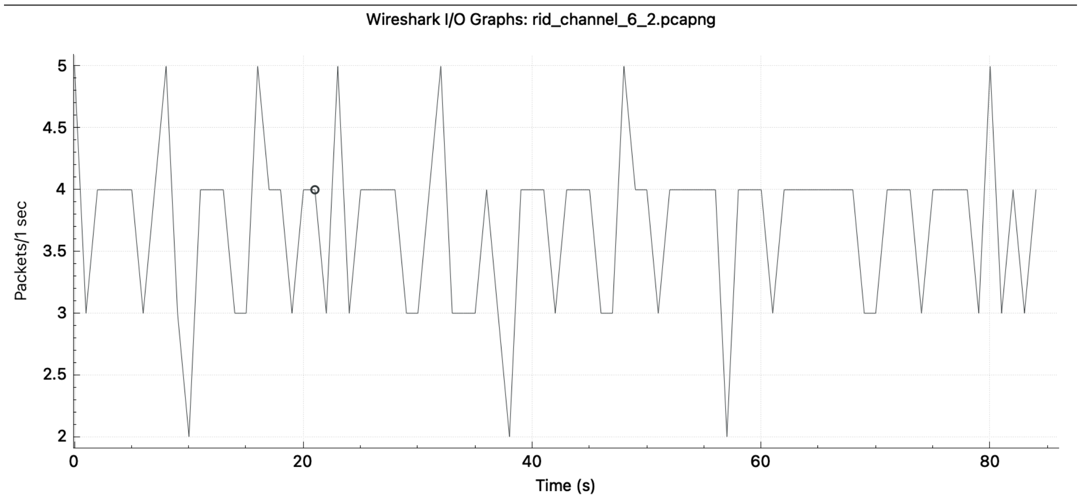
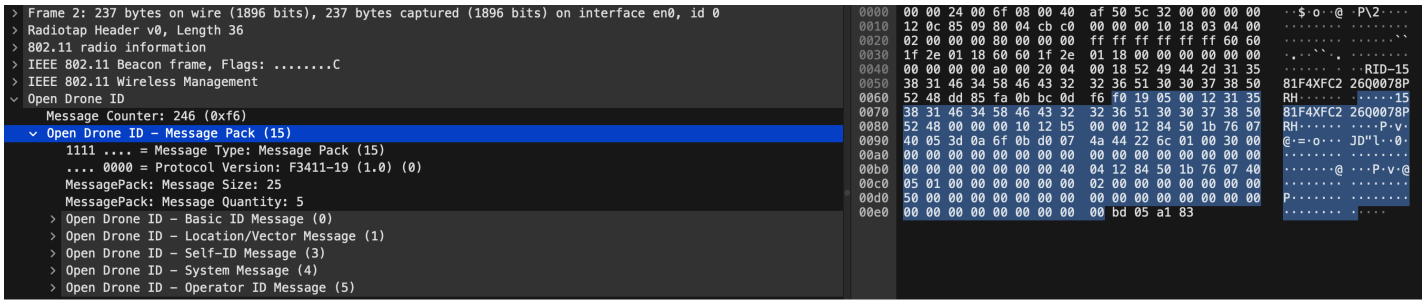


Figure 15 - I/O graph for ODID messages transmitted by the DJI Mini 3 Pro.

Going into more detail, Figure 16 shows that for the DJI Mini 3 Pro, each beacon frame is transmitting an ODID message pack containing five ODID message blocks: Basic ID, Location/Vector, Self-ID, System and Operator.



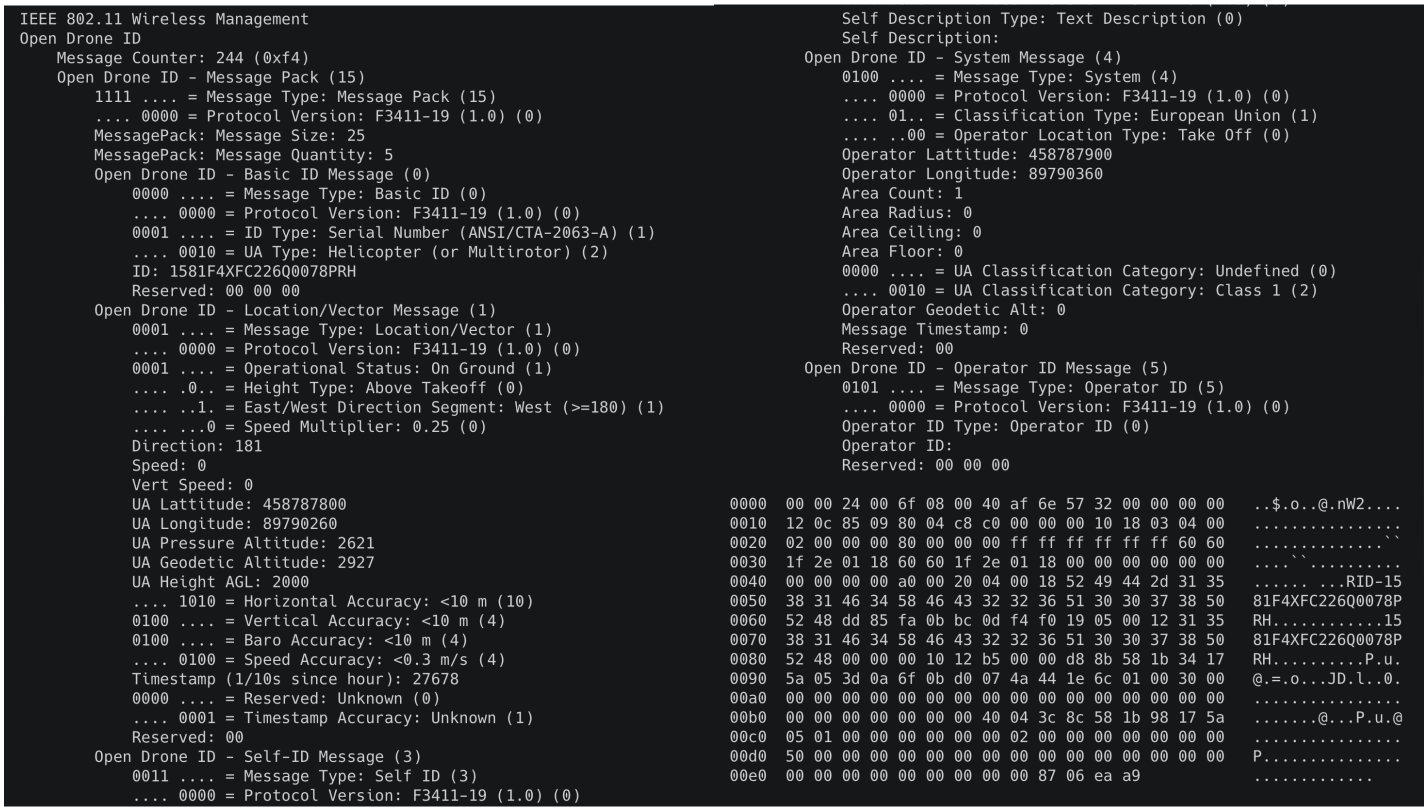
```
> Frame 2: 237 bytes on wire (1896 bits), 237 bytes captured (1896 bits) on interface en0, id 0
> Radiotap Header v0, Length 36
> 802.11 radio information
> IEEE 802.11 Beacon frame, Flags: ........C
> IEEE 802.11 Wireless Management
v Open Drone ID
    Message Counter: 246 (0xf6)
  v Open Drone ID – Message Pack (15)
      1111 .... = Message Type: Message Pack (15)
      .... 0000 = Protocol Version: F3411–19 (1.0) (0)
      MessagePack: Message Size: 25
      MessagePack: Message Quantity: 5
    > Open Drone ID – Basic ID Message (0)
    > Open Drone ID – Location/Vector Message (1)
    > Open Drone ID – Self–ID Message (3)
    > Open Drone ID – System Message (4)
    > Open Drone ID – Operator ID Message (5)
```

```
0000  00 00 24 00 6f 08 00 40  af 50 5c 32 00 00 00 00   ··$·o··@ ·P\2····
0010  12 0c 85 09 80 04 cb c0  00 00 00 10 18 03 04 00   ········ ········
0020  02 00 00 00 80 00 00 00  ff ff ff ff ff ff 60 60   ········ ······``
0030  1f 2e 01 18 60 60 1f 2e  01 18 00 00 00 00 00 00   ·.··``·. ········
0040  00 00 00 00 a0 00 20 04  00 18 52 49 44 2d 31 35   ······ · ··RID-15
0050  38 31 46 34 58 46 43 32  32 36 51 30 30 37 38 50   81F4XFC2 26Q0078P
0060  52 48 dd 85 fa 0b bc 0d  f6 f0 19 05 00 12 31 35   RH······ ······15
0070  38 31 46 34 58 46 43 32  32 36 51 30 30 37 38 50   81F4XFC2 26Q0078P
0080  52 48 00 00 00 10 12 b5  00 00 12 84 50 1b 76 07   RH······ ····P·v·
0090  40 05 3d 0a 6f 0b d0 07  4a 44 22 6c 01 00 30 00   @·=·o··· JD"l··0·
00a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
00b0  00 00 00 00 00 00 00 40  04 12 84 50 1b 76 07 40   ·······@ ···P·v·@
00c0  05 01 00 00 00 00 00 00  02 00 00 00 00 00 00 00   ········ ········
00d0  50 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   P······· ········
00e0  00 00 00 00 00 00 00 00  00 bd 05 a1 83            ········ ·····
```

Figure 16 - ODID message pack transmitted by a DJI Mini 3 Pro.

Finally, a complete dissection (followed by the hexadecimal dump) with all the details of the message blocks contained in the message pack is shown in Figure 17. We can see from the dissection that latitude and longitude values, for example, contained in the Location/Vector message block have values of 458787800 and 89790260. In the hexadecimal dump this corresponds to the little-endian order bytes “0xd8 0x8b 0x58 0x1b” and “0x34 0x17 0x5a 0x05”. Also, as described in the previous sections, these two values are encoded; to obtain the real latitude and longitude the receiver must divide them by 10^7 which results in 45.87878 and 8.979026.

Finally, it should be noted that the UAS ID contained in the Basic ID ODID message block is 1581F4XFC226Q0078PRH, which the Mini 3 Pro also uses for creating the SSID value of the beacon frame. This is not something required by the Open Drone ID protocol or the RID regulation, but it characterizes this drone model and the firmware version used during testing. For more details, we refer the reader to the official Open Drone ID documentation and GitHub organization.



```
IEEE 802.11 Wireless Management
Open Drone ID
    Message Counter: 244 (0xf4)
    Open Drone ID - Message Pack (15)
        1111 .... = Message Type: Message Pack (15)
        .... 0000 = Protocol Version: F3411-19 (1.0) (0)
        MessagePack: Message Size: 25
        MessagePack: Message Quantity: 5
        Open Drone ID - Basic ID Message (0)
            0000 .... = Message Type: Basic ID (0)
            .... 0000 = Protocol Version: F3411-19 (1.0) (0)
            0001 .... = ID Type: Serial Number (ANSI/CTA-2063-A) (1)
            .... 0010 = UA Type: Helicopter (or Multirotor) (2)
            ID: 1581F4XFC226Q0078PRH
            Reserved: 00 00 00
        Open Drone ID - Location/Vector Message (1)
            0001 .... = Message Type: Location/Vector (1)
            .... 0000 = Protocol Version: F3411-19 (1.0) (0)
            0001 .... = Operational Status: On Ground (1)
            .... .0.. = Height Type: Above Takeoff (0)
            .... ..1. = East/West Direction Segment: West (>=180) (1)
            .... ...0 = Speed Multiplier: 0.25 (0)
            Direction: 181
            Speed: 0
            Vert Speed: 0
            UA Lattitude: 458787800
            UA Longitude: 89790260
            UA Pressure Altitude: 2621
            UA Geodetic Altitude: 2927
            UA Height AGL: 2000
            .... 1010 = Horizontal Accuracy: <10 m (10)
            0100 .... = Vertical Accuracy: <10 m (4)
            0100 .... = Baro Accuracy: <10 m (4)
            .... 0100 = Speed Accuracy: <0.3 m/s (4)
            Timestamp (1/10s since hour): 27678
            0000 .... = Reserved: Unknown (0)
            .... 0001 = Timestamp Accuracy: Unknown (1)
            Reserved: 00
        Open Drone ID - Self-ID Message (3)
            0011 .... = Message Type: Self ID (3)
            .... 0000 = Protocol Version: F3411-19 (1.0) (0)
```

```
            Self Description Type: Text Description (0)
            Self Description:
        Open Drone ID - System Message (4)
            0100 .... = Message Type: System (4)
            .... 0000 = Protocol Version: F3411-19 (1.0) (0)
            .... 01.. = Classification Type: European Union (1)
            .... ..00 = Operator Location Type: Take Off (0)
            Operator Lattitude: 458787900
            Operator Longitude: 89790360
            Area Count: 1
            Area Radius: 0
            Area Ceiling: 0
            Area Floor: 0
            0000 .... = UA Classification Category: Undefined (0)
            .... 0010 = UA Classification Category: Class 1 (2)
            Operator Geodetic Alt: 0
            Message Timestamp: 0
            Reserved: 00
        Open Drone ID - Operator ID Message (5)
            0101 .... = Message Type: Operator ID (5)
            .... 0000 = Protocol Version: F3411-19 (1.0) (0)
            Operator ID Type: Operator ID (0)
            Operator ID:
            Reserved: 00 00 00

0000  00 00 24 00 6f 08 00 40 af 6e 57 32 00 00 00 00   ..$.o..@.nW2....
0010  12 0c 85 09 80 04 c8 c0 00 00 00 10 18 03 04 00   ................
0020  02 00 00 00 80 00 00 00 ff ff ff ff ff ff 60 60   ..............``
0030  1f 2e 01 18 60 60 1f 2e 01 18 00 00 00 00 00 00   ....``..........
0040  00 00 00 00 a0 00 20 04 00 18 52 49 44 2d 31 35   ...... ...RID-15
0050  38 31 46 34 58 46 43 32 32 36 51 30 30 37 38 50   81F4XFC226Q0078P
0060  52 48 dd 85 fa 0b bc 0d f4 f0 19 05 00 12 31 35   RH............15
0070  38 31 46 34 58 46 43 32 32 36 51 30 30 37 38 50   81F4XFC226Q0078P
0080  52 48 00 00 00 10 12 b5 00 00 d8 8b 58 1b 34 17   RH..........P.u.
0090  5a 05 3d 0a 6f 0b d0 07 4a 44 1e 6c 01 00 30 00   @.=.o...JD.l..0.
00a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00b0  00 00 00 00 00 00 00 40 04 3c 8c 58 1b 98 17 5a   .......@...P.u.@
00c0  05 01 00 00 00 00 00 00 02 00 00 00 00 00 00 00   ................
00d0  50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   P...............
00e0  00 00 00 00 00 00 00 00 00 87 06 ea a9            .............
```

Figure 17 - Dissection of an ODID message pack transmitted by a DJI Mini 3 Pro.

4. Reverse Engineering the BlueMark DroneScout ds230

The BlueMark DroneScout ds230 is a fixed ground station that receives and parses RID signals based on the ODID protocol. It is compliant with recent EU and U.S. Direct/Broadcast RID standards and supports all the communication technologies required by existing regulations. Being one of the first commercially available ODID-based ground station receivers, it represents an interesting security research target as its internal functions —and therefore vulnerabilities—will potentially be shared with other receivers that also rely on ODID.

Nozomi Networks Labs completely reverse engineered both the hardware and software of the DroneScout ds230 appliance. This chapter presents the results of our activity which include the discovery of multiple vulnerabilities, allowing the implementation of new attack scenarios such as the possibility of "hijacking" legitimate drone trajectories, which we present in Chapter 6.

4.1 DroneScout ds230 Characteristics

The BlueMark DroneScout ds230 appliance, shown in Figure 18, is a Broadcast RID outdoor fixed ground station receiver[22] capable of receiving and interpreting telemetry messages broadcast by drones. Internally the DroneScout is based on the ODID open-source framework (see Chapter 3), which makes it compatible with both EU and U.S. standards for remote identification and tracking (the DIN EN 4709-002 and ASTM F3411-22a-RID-B standards respectively).

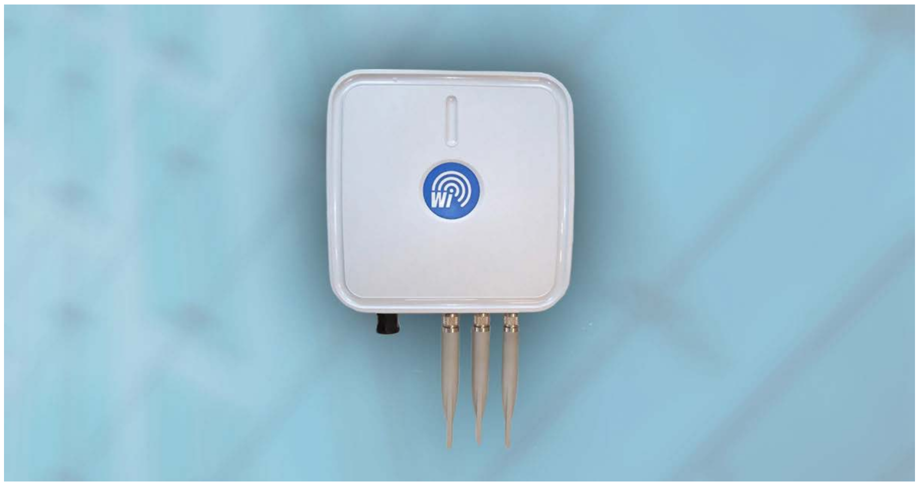


Figure 18 - BlueMark DroneScout ds230 RID receiver.

[22] Also called sensor or radar.

For RID functionality, the DroneScout appliance is equipped with two independent Wi-Fi interfaces and one Bluetooth interface with a dedicated antenna. The appliance supports all communication mechanisms and frequency bands currently required by various RID regulators around the world.[23] The wireless interfaces are configured in monitor mode and continually listen for incoming frames containing RID information packed according to the ODID protocol specification. We refer the reader to the official product manual for any technical specification details not covered in this chapter.[24]

Because RID signals can be broadcast on several different frequencies, the wireless interfaces are managed to cover as many frequency bands as possible. If no ODID payload is detected, both the Wi-Fi radio interfaces will continually loop over all Wi-Fi channels, switching to a new channel every second. If an ODID signal is detected, one Wi-Fi radio will continue hopping over all Wi-Fi channels and scanning for new ODID signals while the other Wi-Fi interface will only loop over Wi-Fi channels where ODID signals have previously been detected. A channel is removed from this list if no new ODID messages are received for a configurable amount of time, set to 60 seconds by default.

The DroneScout ds230 is not a device intended for an end user, it is instead designed for system integrators who want to integrate the functionalities provided by the DroneScout into their own products. Also, the DroneScout is not a standalone device; it requires an MQTT broker (typically provided by the system integrator) to collect detected RID information. Finally, the DroneScout is also equipped with a Power over Ethernet interface which provides power and allows asset owners to connect it to their own networks. The device uses this interface to communicate with the MQTT broker.

From a high-level point of view the DroneScout works as follows:

- It uses its internal wireless interfaces to continuously scan both Wi-Fi and Bluetooth channels. From the wireless point of view the DroneScout is a completely passive device. The wireless interfaces, configured in monitor mode, never transmit anything over wireless.
- When a frame (Wi-Fi or Bluetooth) containing RID information (packed as specified by the ODID protocol) is detected, it parses the content of the ODID message and associates the parsed RID information to the source MAC address of the drone sending the message.
- Collected RID information is periodically transmitted over the Ethernet interface to the third-party managed MQTT broker managed by the system integrators or asset owners. The content of the MQTT messages is JSON formatted.

[23] The communication mechanisms are Bluetooth 4.x legacy advertisement, Bluetooth 5.0 Extended advertisement, Wi-Fi NaN and Wi-Fi beacon. The frequency bands are 2.4GHz, 5.2GHz and 5.8GHz.
[24] DroneScout 230 manual is available at: **download.bluemark.io/dronescout_sensor_manual_230.pdf**

4.2 DroneScout ds230 Hardware

Looking at the DroneScout ds230’s hardware, Figure 19 shows the external enclosure of the appliance. The side views on the right show a PoE ethernet port and the three N-Type antenna connectors.

Opening the enclosure reveals that the DroneScout ds230 does not use a custom-made board but is instead based on multiple commercial-off-the-shelf (COTS) hardware components (Figure 20).

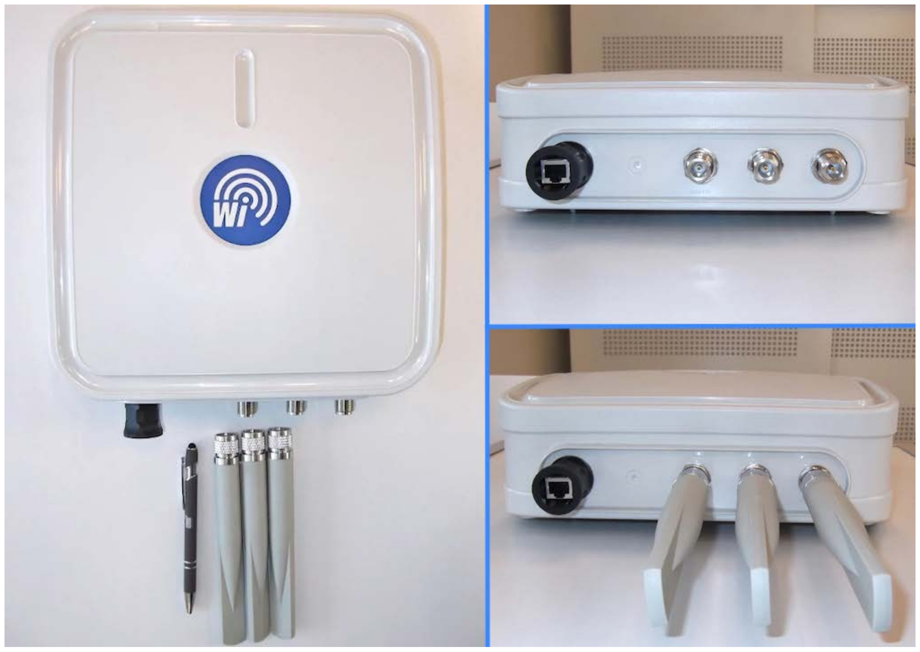


Figure 19 - BlueMark DroneScout ds230 external view.

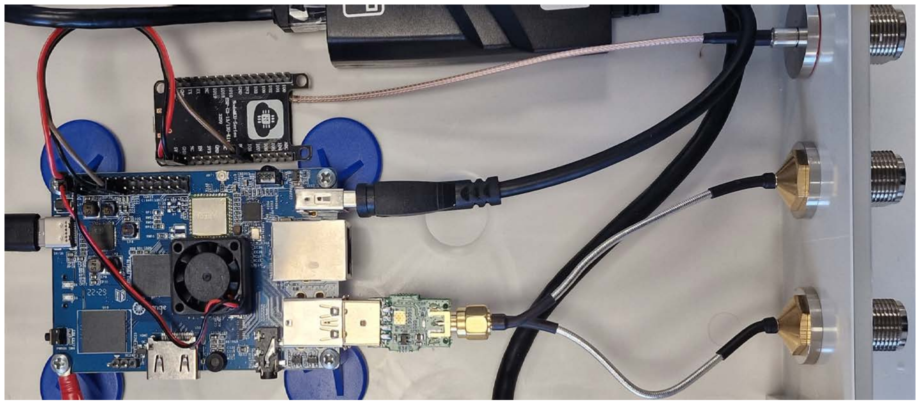


Figure 20 - BlueMark DroneScout ds230 internals.

In particular, the DroneScout uses an Orange Pi3 (Allwinner SUNXI64) as the main board (Figure 21). A PoE to USB-C converter is used to provide alimentation and ethernet connectivity to the board.

For the Wi-Fi interfaces, Figure 22 shows that the ds230 uses two USB dongles based on Realtek 8812AU/8821AU 802.11ac WLAN chipsets connected to the Orange Pi 3 board.

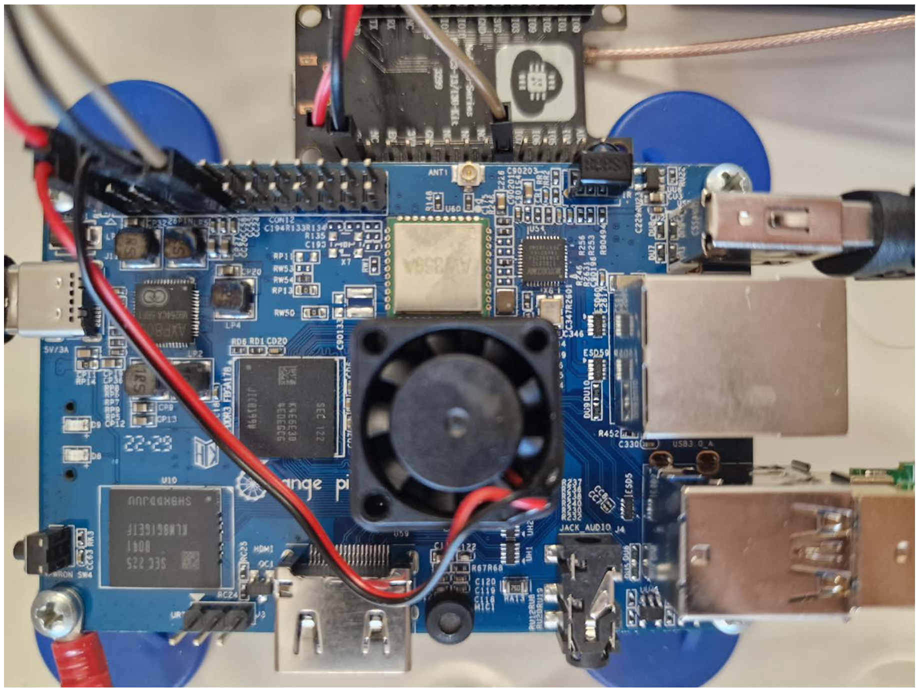


Figure 21 - BlueMark DroneScout ds230 main board.

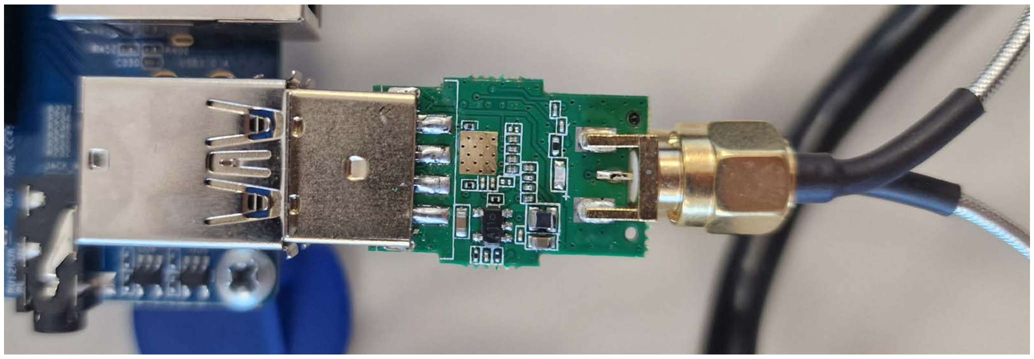
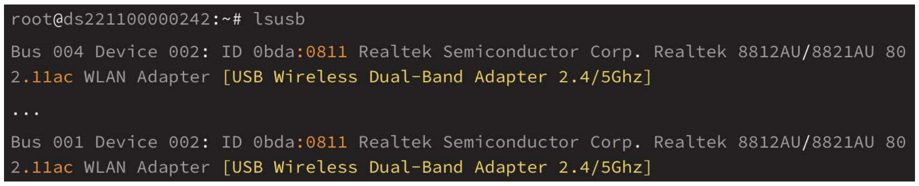


Figure 22 - BlueMark DroneScout ds230 Wi-Fi SoC.

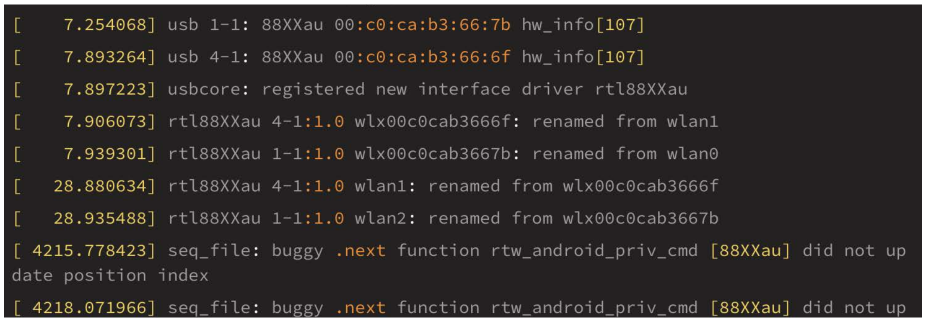
This can also be confirmed by listing the USB devices connected to the main board using the *lsusb* command whose output is shown in Figure 23.

The two wireless interfaces are managed by the kernel module 88XXau, confirmed by the output of the *dmesg* command shown in Figure 24.



```
root@ds221100000242:~# lsusb
Bus 004 Device 002: ID 0bda:0811 Realtek Semiconductor Corp. Realtek 8812AU/8821AU 80
2.11ac WLAN Adapter [USB Wireless Dual-Band Adapter 2.4/5Ghz]
...
Bus 001 Device 002: ID 0bda:0811 Realtek Semiconductor Corp. Realtek 8812AU/8821AU 80
2.11ac WLAN Adapter [USB Wireless Dual-Band Adapter 2.4/5Ghz]
```

Figure 23 - BlueMark DroneScout ds230 lsusb output.



```
[    7.254068] usb 1-1: 88XXau 00:c0:ca:b3:66:7b hw_info[107]
[    7.893264] usb 4-1: 88XXau 00:c0:ca:b3:66:6f hw_info[107]
[    7.897223] usbcore: registered new interface driver rtl88XXau
[    7.906073] rtl88XXau 4-1:1.0 wlx00c0cab3666f: renamed from wlan1
[    7.939301] rtl88XXau 1-1:1.0 wlx00c0cab3667b: renamed from wlan0
[   28.880634] rtl88XXau 4-1:1.0 wlan1: renamed from wlx00c0cab3666f
[   28.935488] rtl88XXau 1-1:1.0 wlan2: renamed from wlx00c0cab3667b
[ 4215.778423] seq_file: buggy .next function rtw_android_priv_cmd [88XXau] did not up
date position index
[ 4218.071966] seq_file: buggy .next function rtw_android_priv_cmd [88XXau] did not up
```

Figure 24 - BlueMark DroneScout ds230 dmesg output.

Finally, the Bluetooth sniffing is handled by a development kit based on the ESP32-C3 SoC. As shown in Figure 25, the DroneScout uses a NodeMCU series ESP-C3-13 development kit[25] connected to the Orange Pi 3 board through a serial connection (PIN IO8) which is used by the ESP32 SoC for transmitting the captured RID data to the main board.

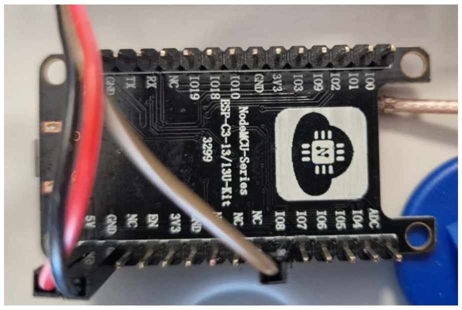
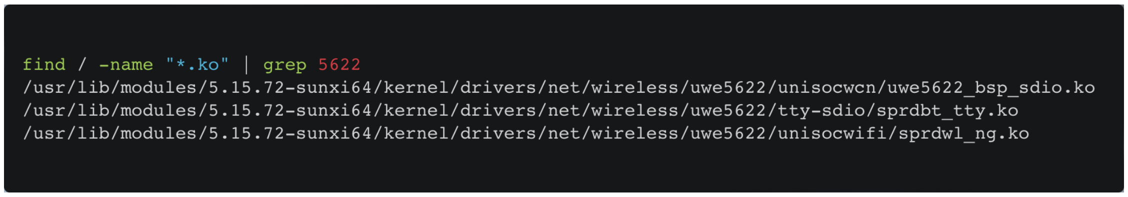


Figure 25 - . BlueMark DroneScout ds230 Bluetooth SoC.

[25] ESP-C3-13-Kit specifications can be found at: **docs.ai-thinker.com/_media/esp32/docs/esp-c3-13-kit-v1.0_specification.pdf**

On the DroneScout there is also a Wi-Fi/Bluetooth chipset based on a Unisoc UWE5622[26] and equipped directly on the Orange Pi 3 board. This chipset is managed by out-of-tree kernel drivers whose names are shown in Figure 26. However, this chip is not used by the DroneScout ds230 and its antenna is disconnected.



```
find / -name "*.ko" | grep 5622
/usr/lib/modules/5.15.72-sunxi64/kernel/drivers/net/wireless/uwe5622/unisocwcn/uwe5622_bsp_sdio.ko
/usr/lib/modules/5.15.72-sunxi64/kernel/drivers/net/wireless/uwe5622/tty-sdio/sprdbt_tty.ko
/usr/lib/modules/5.15.72-sunxi64/kernel/drivers/net/wireless/uwe5622/unisocwifi/sprdwl_ng.ko
```

Figure 26 - BlueMark DroneScout drivers for Unisoc UWE5622 Wi-Fi SoC.

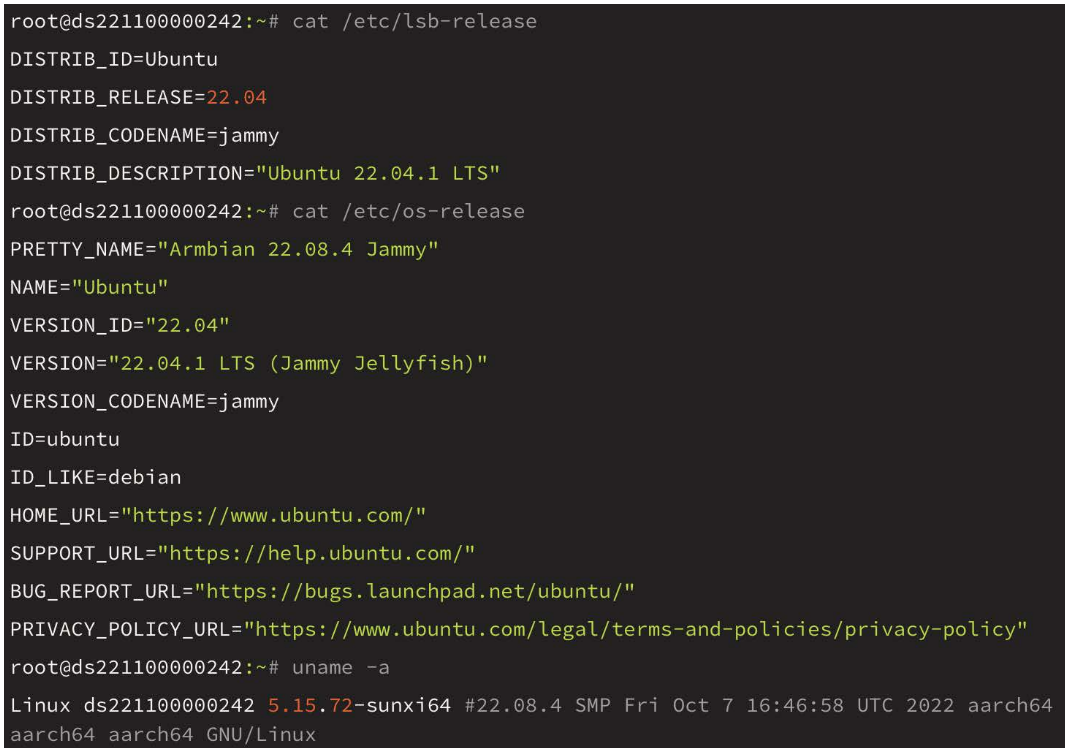
4.3 DroneScout Firmware Reverse Engineering

In order to understand the internal functioning and hunt for potential vulnerabilities, Nozomi Networks Labs completely reverse engineered the firmware running on the DroneScout ds230 appliance.[27]

At the time of writing BlueMark does not provide a way to download the device firmware directly from their website. However, by looking at the file /*root/update.sh*[28] it was easy to determine that the firmware is downloaded from the following endpoint: **https://download.bluemark.io/dronescout/firmware/stable/ds230.tar.bz2**

The firmware download package does not contain a full disk image but just a few configuration files, bash scripts and one executable file called *dronescout.arm64* which is the main firmware component. The firmware base image is based on an Ubuntu 22.04.1 LTS for ARM devices with Linux kernel version 5.15.72. The details of the operating system installed on the DroneScout are provided in Figure 27.

[26] For more information on the Unisoc UWE5622: **unisoc.com/cn_zh/home/TJUWLW-56XX-2**.
[27] Information presented in this section has been extracted by reverse engineering DroneScout firmware version 20220608-1239. It is mostly applicable to later firmware versions unless otherwise stated.
[28] The DroneScout ds230 filesystem can be freely explored as it can be accessed through SSH using the default credential: username root and password bluemark. This is a functionality provided by the vendor and documented in the official product manual. Also, by performing a nma scan on the ethernet interface, SSH on port 22 is the only open port.



```
root@ds221100000242:~# cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=22.04
DISTRIB_CODENAME=jammy
DISTRIB_DESCRIPTION="Ubuntu 22.04.1 LTS"
root@ds221100000242:~# cat /etc/os-release
PRETTY_NAME="Armbian 22.08.4 Jammy"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.1 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
root@ds221100000242:~# uname -a
Linux ds221100000242 5.15.72-sunxi64 #22.08.4 SMP Fri Oct 7 16:46:58 UTC 2022 aarch64
aarch64 aarch64 GNU/Linux
```

Figure 27 - BlueMark DroneScout ds230 system information.

Most of the files provided as part of the DroneScout firmware are located in the /root directory. We list them below with brief descriptions:

- ***dronescout.arm64 (SHA256: f9a519632273ecafe52-661d017876e8ebaddedf626d dc22c66c8e4e991fa e3c6):*** main executable taking care of sniffing wire-less traffic (Wi-Fi and Bluetooth), channel scanning, ODID message parsing and MQTT publishing.
- ***dronescout.conf (SHA256: 92af13b5af6b52675e-2904d0149e59cbc5b32407d3eb58fd07226ze592 67d4f):*** configuration file for *dronescout.arm64*.
- ***remote.sh (SHA256: cd7e885bd951a4139ed75e-2631ed066ddf90474ccf1209b32fcefbf01f16daef):*** script for setting up reverse SSH tunnel to remotely access the DroneScout when placed behind a router or firewall. Disabled by default.
- ***remote.conf (SHA256: e58b7170f7323d71341ff3c-bd256221216723215 44b48b2eac216fdf38b83fab):*** configuration file for *remote.sh*.
- ***run.sh (SHA256: cf00d1556861b69dd68d524ae37-e1310633ebfb5547d6108c6c122cf6c950cab):*** script for setting up the Wi-Fi and Bluetooth interfaces and running *dronescout.arm64*.
- ***start.sh (SHA256: cb3c31a6e0d2633b802056e6b-cda7a73a503a72ced435a570bded47017c42f1f):*** main script for setting up and starting the system. It starts *run.sh*, wath*dog.sh* and *remote.sh* (each in its own screen session).

- ***update.sh (SHA256: 97536f07adffc1327795cc6c-fb9b417ba0dd69c113ce50e27eb646c2dacad6c9):*** script for starting the firmware update procedure.
- ***watchdog.sh (SHA256: cf3985dbba11be1d020b311c-08bc4eda5dae8111d80b96ee1f8feedd48682832):*** checks if *dronescout.arm64* is still running correctly every 10 seconds. If this is not the case, it reboots the DroneScout.
- ***wlan.conf (SHA256: c6f77e8d2440d7ae8a35f830-6473e7c9993c3c2826085c6baed9114dcddeca5a):*** configuration file containing the initial names of the Wi-Fi interfaces that are then changed to wlan1 and wlan2 during initialization by *run.sh*.
- ***wlan_channels.conf (SHA256: 7e3dd8d174ce554-35073887281e51019a6d6621515e0f7dace0551281d-8be6):*** configuration file containing the list of Wi-Fi channels to scan.

The firmware also comes with a crontab file placed in */etc/cron.weekly/reboot* which reboots the system every week and with the file */etc/rc.local* which calls */root/start.sh* at system boot. The main executable *dronescout.arm64* is executed as root in a non-sandboxed environment.

Also, while investigating the device file system, Nozomi Networks Labs found the public SSH key shown in Figure 28, likely a leftover from device firmware development.

Finally, the ESP32-C3 SoC runs its own customized firmware which is in charge of handling the Bluetooth sniffing and parsing.

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC8JYJPJHRTMWTQD1TNGtW0MkO5pdmmeUsJPhS+ycv6bR3s7E
mmyp1Kv8T7lLjO70nD237Ucs7+RNEKWPt0fm401Gb+Q17OaJmB6KZRW/URh4H/fkzQTD9BLQVU+g6/PQ0e/cjL
cKSzSI5YI29rl0fSsiUh81fwR1HY/gi00lm+od8XkrKrLEpo+LPL+g0jRKYoPfgvuLprOjr4W2jW9ngfYFeMg7
JcRf+QQjwqr3YmfiGWZXl8XVGe09aJhlyjTmIxbzN9uYCz4anU0QfbnEFD4RV7nJAQu9ti9rZyxGzxDk/PFRAd
2oJV+j+QhQnmsbcpr0Ed3p/WIAfFrApKd/k/ roel@roel-laptop
```

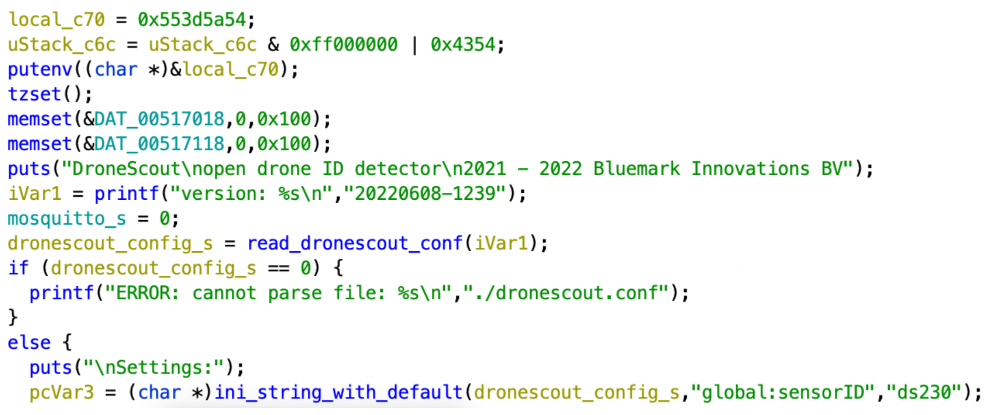
Figure 28 - BlueMark DroneScout public key found on the device.

4.3.1 DroneScout Firmware Main Executable: *dronescout.arm64*

The DroneScout firmware's main executable is called *dronescout.arm64*. This is the most important executable, as it handles all the functionalities of the product and is automatically executed when the device boots. This executable requires a "license" file to work correctly, which is located in */root/.ssh/serial*. license and its content on the analyzed device is "*sn: 82c00007bffb6b5c usb: 0000 0000 check: 0375 86ed*".[29]

At a high-level, the DroneScout's firmware operations can be divided in two macro blocks:

- The main function in charge of reading the configuration, setting up the system (e.g., configuring Wi-Fi interfaces) and spawning multiple threads.
- Multiple threads that run forever, each in charge of its own sub-functionality (e.g., communication with MQTT broker, Wi-Fi frame parsing, etc.).



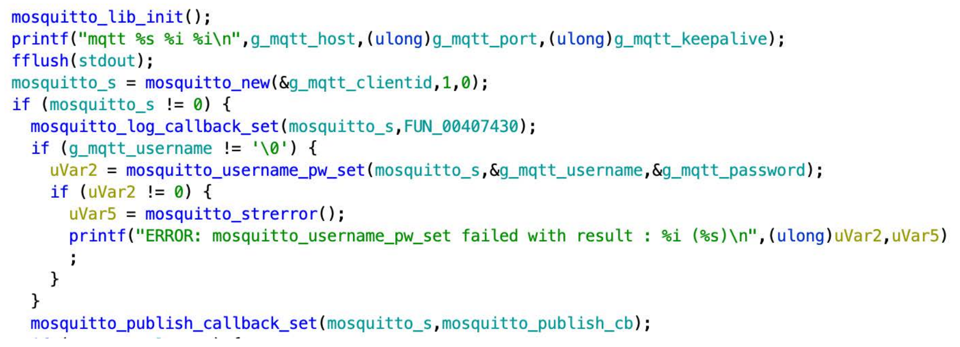
```
local_c70 = 0x553d5a54;
uStack_c6c = uStack_c6c & 0xff000000 | 0x4354;
putenv((char *)&local_c70);
tzset();
memset(&DAT_00517018,0,0x100);
memset(&DAT_00517118,0,0x100);
puts("DroneScout\nopen drone ID detector\n2021 - 2022 Bluemark Innovations BV");
iVar1 = printf("version: %s\n","20220608-1239");
mosquitto_s = 0;
dronescout_config_s = read_dronescout_conf(iVar1);
if (dronescout_config_s == 0) {
  printf("ERROR: cannot parse file: %s\n","./dronescout.conf");
}
else {
  puts("\nSettings:");
  pcVar3 = (char *)ini_string_with_default(dronescout_config_s,"global:sensorID","ds230");
```

Figure 29 - ds230 firmware: configuration file reading.

The *dronescout.arm64* process starts by reading the configuration file (*dronescout.conf*) and saving the various setting parameters in a global data structure that is then accessed in various parts of the code (Figure 29). We will not go into details, but the configuration contains things like IP address and port of the MQTT broker, Wi-Fi interface names, thresholds for Wi-Fi frame RSSI, etc. and can be used to tweak the DroneScout's behavior.

[29] The license checking procedure is straightforward and it is easy to generate "fake" licenses that are accepted by *dronescout.arm64*.

The process then continues by setting up the (Mosquitto-based) MQTT stack (Figure 30). In fact, *dronscout.arm64* acts like an MQTT publisher and various options can be configured in *dronescout.conf*, like the MQTT broker host/IP and port, username, password and, in case SSL is used, keys and certificates.

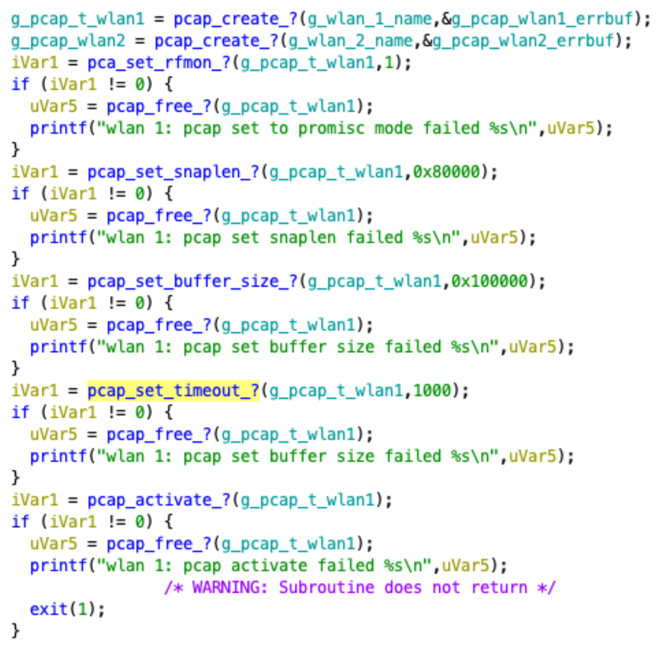


```
mosquitto_lib_init();
printf("mqtt %s %i %i\n",g_mqtt_host,(ulong)g_mqtt_port,(ulong)g_mqtt_keepalive);
fflush(stdout);
mosquitto_s = mosquitto_new(&g_mqtt_clientid,1,0);
if (mosquitto_s != 0) {
  mosquitto_log_callback_set(mosquitto_s,FUN_00407430);
  if (g_mqtt_username != '\0') {
    uVar2 = mosquitto_username_pw_set(mosquitto_s,&g_mqtt_username,&g_mqtt_password);
    if (uVar2 != 0) {
      uVar5 = mosquitto_strerror();
      printf("ERROR: mosquitto_username_pw_set failed with result : %i (%s)\n",(ulong)uVar2,uVar5)
      ;
    }
  }
  mosquitto_publish_callback_set(mosquitto_s,mosquitto_publish_cb);
```

Figure 30 - ds230 firmware: mosquitto setup.

Then the two Wi-Fi interfaces are configured to work in monitor mode. *dronescout.arm64* relies on *libpcap* (statically compiled) for sniffing the Wi-Fi traffic. *libpcap* capture handler is set the same way for both Wi-Fi interfaces (see Figure 31):

- Monitor mode enabled
- Snapshot length: 524288 bytes
- Buffer size: 1MB
- Buffer timeout: 1s

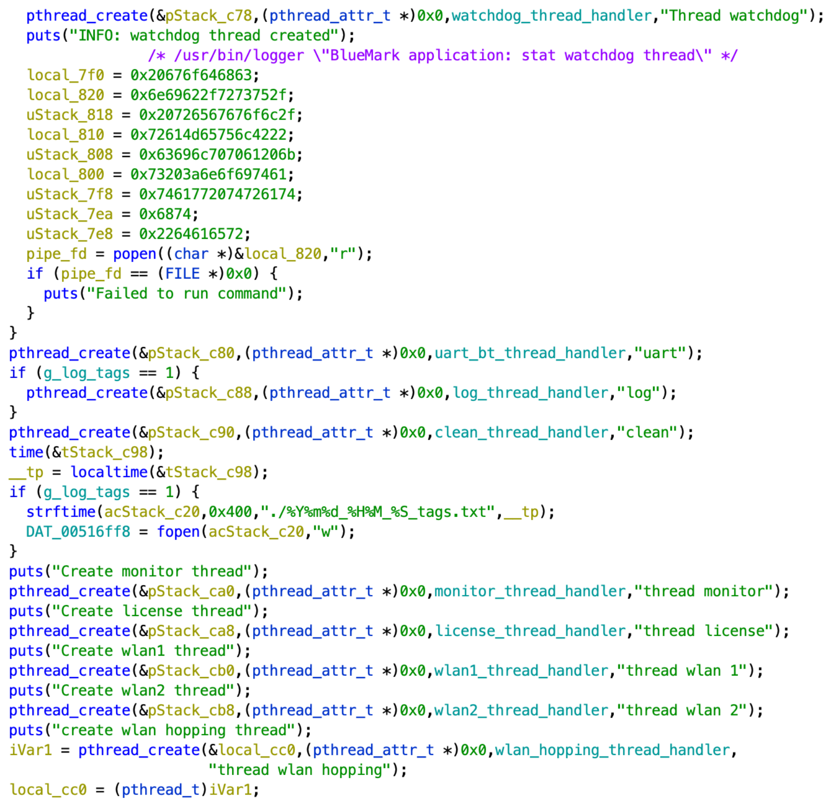


```
g_pcap_t_wlan1 = pcap_create_?(g_wlan_1_name,&g_pcap_wlan1_errbuf);
g_pcap_wlan2 = pcap_create_?(g_wlan_2_name,&g_pcap_wlan2_errbuf);
iVar1 = pca_set_rfmon_?(g_pcap_t_wlan1,1);
if (iVar1 != 0) {
  uVar5 = pcap_free_?(g_pcap_t_wlan1);
  printf("wlan 1: pcap set to promisc mode failed %s\n",uVar5);
}
iVar1 = pcap_set_snaplen_?(g_pcap_t_wlan1,0x80000);
if (iVar1 != 0) {
  uVar5 = pcap_free_?(g_pcap_t_wlan1);
  printf("wlan 1: pcap set snaplen failed %s\n",uVar5);
}
iVar1 = pcap_set_buffer_size_?(g_pcap_t_wlan1,0x100000);
if (iVar1 != 0) {
  uVar5 = pcap_free_?(g_pcap_t_wlan1);
  printf("wlan 1: pcap set buffer size failed %s\n",uVar5);
}
iVar1 = pcap_set_timeout_?(g_pcap_t_wlan1,1000);
if (iVar1 != 0) {
  uVar5 = pcap_free_?(g_pcap_t_wlan1);
  printf("wlan 1: pcap set buffer size failed %s\n",uVar5);
}
iVar1 = pcap_activate_?(g_pcap_t_wlan1);
if (iVar1 != 0) {
  uVar5 = pcap_free_?(g_pcap_t_wlan1);
  printf("wlan 1: pcap activate failed %s\n",uVar5);
                /* WARNING: Subroutine does not return */
  exit(1);
}
```

Figure 31 - ds230 firmware: Wi-Fi interface setup.

Finally, as shown in Figure 32, the main function of *dronescout.arm64* starts multiple threads, each in charge of handling a specific functionality. The most important threads are:

- License Thread (*license_thread_handler(), 0x407998*): periodically checks if the license file is valid. The period is a random number of seconds each time.
- Wi-Fi Sniffing Threads (*wlan{1,2}_thread_handler(), 0x407dd0, 0x407e48*): there are two of these, one for each Wi-Fi interface. They are in charge of handling the Wi-Fi frame sniffing and parsing and possibly the parsing of the ODID payload contained in them.
- Wi-Fi Channel Hopping Thread (*wlan_hopping_thread_handler(), 0x408150*): handles the Wi-Fi channel scanning. It uses mutexes to synchronize with the Wi-Fi sniffing threads.
- Bluetooth Sniffing Thread (*uart_bt_thread_handler(), 0x409468*): handles the communication with the external ESP32-C3 SoC and periodically parses the JSON data received from the microcontroller.
- Cleaning Thread (*clean_thread_handler(), 0x407708*): parses and cleans the global data structures containing the received RID data every few hours.
- Logging Thread (*log_thread_handler(), 0x408040*): periodical log of DroneScout operations.
- Monitor Thread (*monitor_thread_handler(), 0x407760*): periodically checks the DroneScout's running state and send a JSON status message to the MQTT broker.



```
    pthread_create(&pStack_c78,(pthread_attr_t *)0x0,watchdog_thread_handler,"Thread watchdog");
    puts("INFO: watchdog thread created");
                  /* /usr/bin/logger \"BlueMark application: stat watchdog thread\" */
    local_7f0 = 0x20676f646863;
    local_820 = 0x6e69622f7273752f;
    uStack_818 = 0x20726567676f6c2f;
    local_810 = 0x72614d65756c4222;
    uStack_808 = 0x63696c707061206b;
    local_800 = 0x73203a6e6f697461;
    uStack_7f8 = 0x7461772074726174;
    uStack_7ea = 0x6874;
    uStack_7e8 = 0x2264616572;
    pipe_fd = popen((char *)&local_820,"r");
    if (pipe_fd == (FILE *)0x0) {
      puts("Failed to run command");
    }
  }
  pthread_create(&pStack_c80,(pthread_attr_t *)0x0,uart_bt_thread_handler,"uart");
  if (g_log_tags == 1) {
    pthread_create(&pStack_c88,(pthread_attr_t *)0x0,log_thread_handler,"log");
  }
  pthread_create(&pStack_c90,(pthread_attr_t *)0x0,clean_thread_handler,"clean");
  time(&tStack_c98);
  __tp = localtime(&tStack_c98);
  if (g_log_tags == 1) {
    strftime(acStack_c20,0x400,"./%Y%m%d_%H%M_%S_tags.txt",__tp);
    DAT_00516ff8 = fopen(acStack_c20,"w");
  }
  puts("Create monitor thread");
  pthread_create(&pStack_ca0,(pthread_attr_t *)0x0,monitor_thread_handler,"thread monitor");
  puts("Create license thread");
  pthread_create(&pStack_ca8,(pthread_attr_t *)0x0,license_thread_handler,"thread license");
  puts("Create wlan1 thread");
  pthread_create(&pStack_cb0,(pthread_attr_t *)0x0,wlan1_thread_handler,"thread wlan 1");
  puts("Create wlan2 thread");
  pthread_create(&pStack_cb8,(pthread_attr_t *)0x0,wlan2_thread_handler,"thread wlan 2");
  puts("create wlan hopping thread");
  iVar1 = pthread_create(&local_cc0,(pthread_attr_t *)0x0,wlan_hopping_thread_handler,
                         "thread wlan hopping");
  local_cc0 = (pthread_t)iVar1;
```

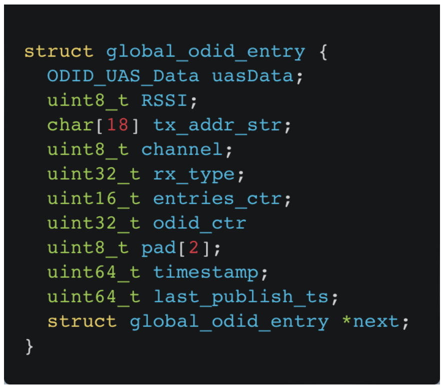
Figure 32 - ds230 firmware: threads.

[30] After the threads have been spawned, the main function enters an infinite loop.

4.3.1.1 UAS Data Structure

Every time a new ODID message is received by the DroneScout, it is parsed by a function provided by the ODID reference implementation API and decoded into an *ODID_UAS_Data* data structure (see section 3.2.1).

Then, this data structure is copied by the function *copy_odid_uas_data_in_global_ll() (0x4065b8)* into a global linked list which we call *g_data*. Each entry of this linked list is of type *global_odid_entry* which is defined as shown in Figure 33.[31]



```
struct global_odid_entry {
  ODID_UAS_Data uasData;
  uint8_t RSSI;
  char[18] tx_addr_str;
  uint8_t channel;
  uint32_t rx_type;
  uint16_t entries_ctr;
  uint32_t odid_ctr
  uint8_t pad[2];
  uint64_t timestamp;
  uint64_t last_publish_ts;
  struct global_odid_entry *next;
}
```

Figure 33 - ds230 firmware: reversed UAS Data structure.

The fields of global_odid_entry are:

- *uasData*: a copy of the *ODID_UAS_Data* data structure.
- *RSSI*: the RSSI value of the received frame containing ODID data.
- *tx_addr_str*: the string on the MAC address that broadcasts the frame containing the ODID data.
- *channel*: the channel that the frame containing the ODID data was received on.
- *rx_type*: identifies the type of broadcast message (Wi-Fi beacon, Wi-Fi NaN, Bluetooth 4.x legacy advertisement, Bluetooth 5.0 extended advertisement).
- *entries_ctr*: a counter tracking how much ODID data has been received from the *tx_addr_str* MAC address.
- *odid_ctr*: the ODID message counter.
- *timestamp*: timestamp in milliseconds of when the ODID data was received.
- *last_publish_ts*: timestamp in milliseconds of the last time the ODID data for this particular *tx_addr_str* was published to the MQTT broker.
- *next*: pointer to the next *global_odid_entry*.

When the *ODID_UAS_Data* refers to a transmitter device (*tx_addr_str*) not already present in the *g_data*, a new *global_odid_entry* is created and appended at the end of the existing *g_data* (or put at the head if this is the first entry). When the transmitter device (*tx_addr_str*) already exists, no new entries are created. In this case the *global_odid_entry* referring to that particular *tx_addr_str* is retrieved from the *g_data* list. Its *global_odid_entry→uasData* field is updated with the info contained in the new *ODID_UAS_Data*.

[31] Note that this definition has been reverse engineered, so a few details could be wrong.

This means that for a given *tx_addr_str* no historical values are maintained.

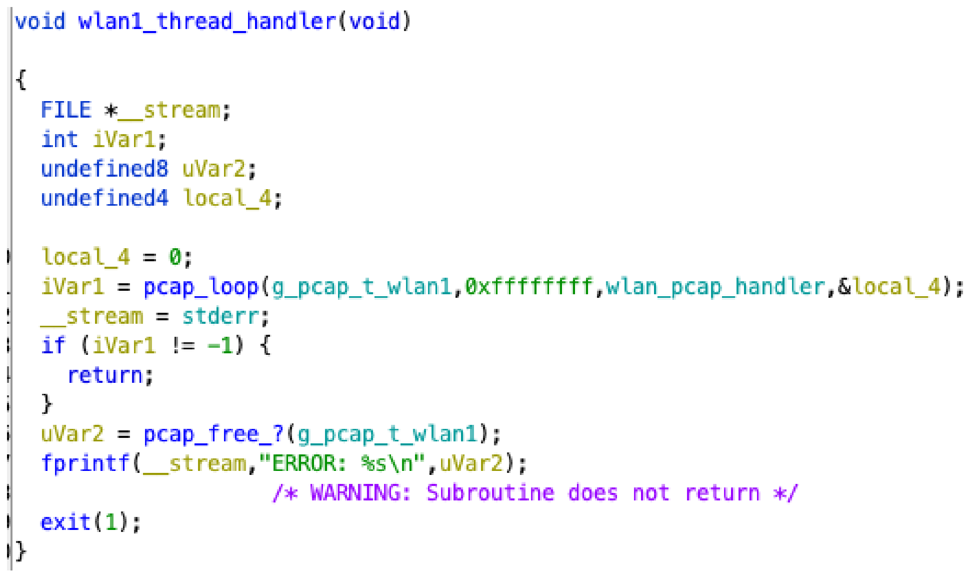
Of particular interest is the function *log_odid_entry() (0x4084c0)* (which is called for every *ODID_UAS_Data* received) that does not publish to the MQTT broker the new data for a given *tx_addr_str* if

(global_odid_entry→timestamp - global_odid_entry→last_publish_ts)

is less than 500 *ms* (or 1000 *ms* depending on the configuration). This means that injecting a lot of spoofed traffic for a given *tx_addr_str* is not directly reflected in what is published to the MQTT broker. Later in this chapter we will show how it is possible to exploit this DroneScout firmware behavior to "hijack" legitimate drone trajectories.

4.3.1.2 Wi-Fi Sniffing and Parsing

In the Dronescout's firmware, Wi-Fi sniffing is handled by *wlan1_thread_handler()* for the "first" Wi-Fi interface and by *wlan2_thread_handler()* for the "second" Wi-Fi interface. These two functions do exactly the same thing: they call *pcap_loop()* passing *wlan_pcap_handler() (0x408708)* as callback function. *wlan_pcap_handler()* is the function that actually takes care of parsing the sniffed Wi-Fi frames (Figure 34).



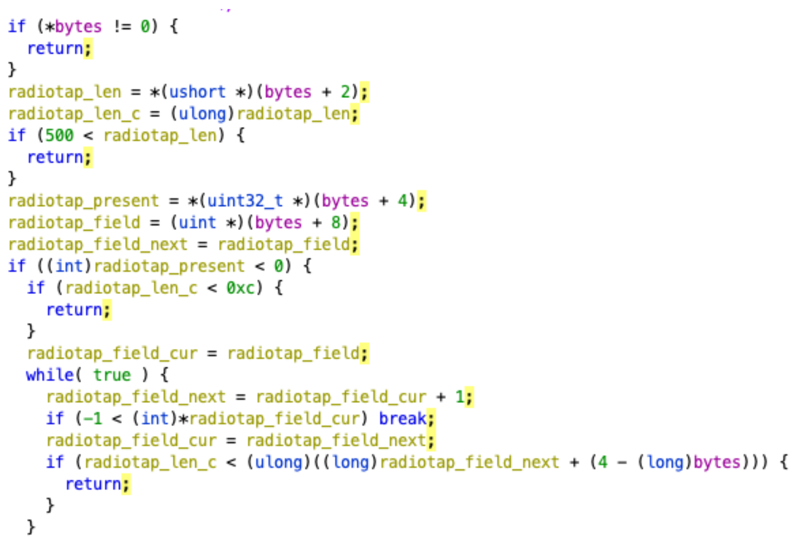
```
void wlan1_thread_handler(void)

{
  FILE *__stream;
  int iVar1;
  undefined8 uVar2;
  undefined4 local_4;

  local_4 = 0;
  iVar1 = pcap_loop(g_pcap_t_wlan1,0xffffffff,wlan_pcap_handler,&local_4);
  __stream = stderr;
  if (iVar1 != -1) {
    return;
  }
  uVar2 = pcap_free_?(g_pcap_t_wlan1);
  fprintf(__stream,"ERROR: %s\n",uVar2);
                    /* WARNING: Subroutine does not return */
  exit(1);
}
```

Figure 34 - ds230 firmware: Wi-Fi thread handler.

wlan_pcap_handler() starts by parsing the radiotap header (Figure 35). "This is done to extract information like the frequency and channel the frame was received on and the measured RSSI (frames whose RSSI is below a configurable threshold are dropped and not parsed by the code that follows).

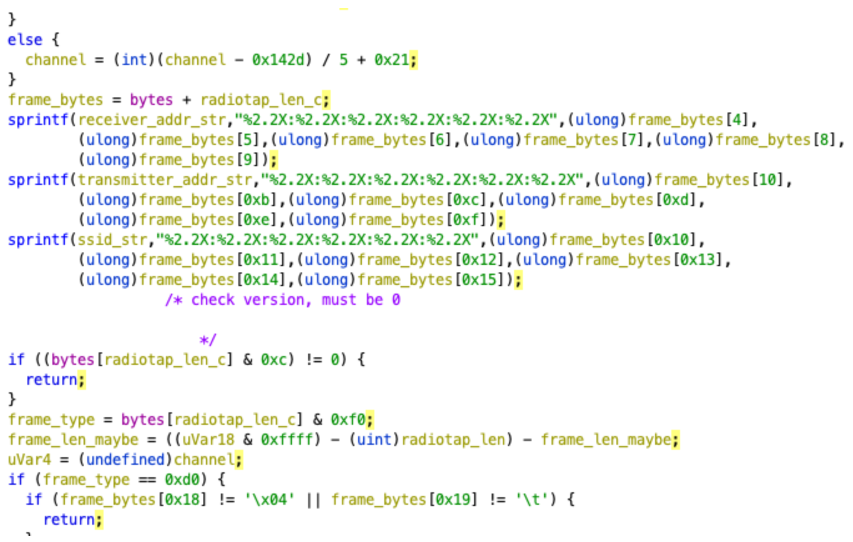
Once the radiotap header parsing is complete, the function starts parsing the actual Wi-Fi frame (Figure 36). Without going too far into details, the code performs the following actions:

- Parses and saves the receiver, transmitter and BSSID addresses.
- Continues with the parsing code only if the frame Type is 0 (i.e., management).[32]
- It then extracts the frame sub-type and continues only if the sub-type is beacon (*0x08*) or action NaN



```
if (*bytes != 0) {
  return;
}
radiotap_len = *(ushort *)(bytes + 2);
radiotap_len_c = (ulong)radiotap_len;
if (500 < radiotap_len) {
  return;
}
radiotap_present = *(uint32_t *)(bytes + 4);
radiotap_field = (uint *)(bytes + 8);
radiotap_field_next = radiotap_field;
if ((int)radiotap_present < 0) {
  if (radiotap_len_c < 0xc) {
    return;
  }
  radiotap_field_cur = radiotap_field;
  while( true ) {
    radiotap_field_next = radiotap_field_cur + 1;
    if (-1 < (int)*radiotap_field_cur) break;
    radiotap_field_cur = radiotap_field_next;
    if (radiotap_len_c < (ulong)((long)radiotap_field_next + (4 - (long)bytes))) {
      return;
    }
  }
```

Figure 35 - ds230 firmware: radiotap header parsing.



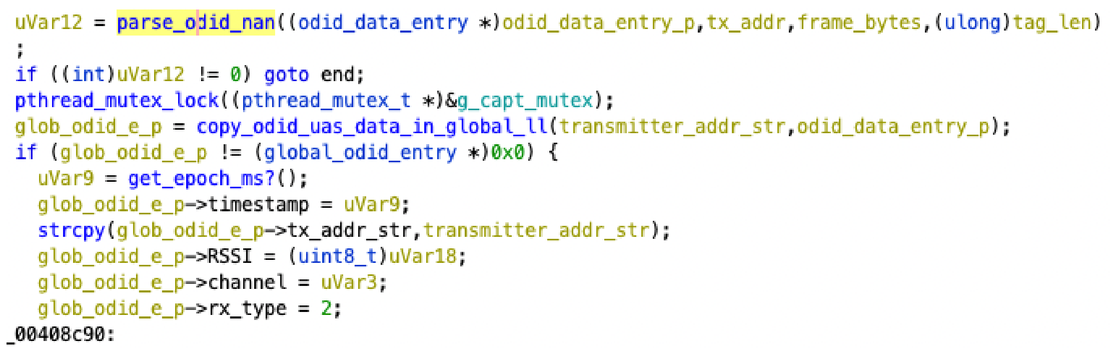
```
}
else {
  channel = (int)(channel - 0x142d) / 5 + 0x21;
}
frame_bytes = bytes + radiotap_len_c;
sprintf(receiver_addr_str,"%2.2X:%2.2X:%2.2X:%2.2X:%2.2X:%2.2X",(ulong)frame_bytes[4],
        (ulong)frame_bytes[5],(ulong)frame_bytes[6],(ulong)frame_bytes[7],(ulong)frame_bytes[8],
        (ulong)frame_bytes[9]);
sprintf(transmitter_addr_str,"%2.2X:%2.2X:%2.2X:%2.2X:%2.2X:%2.2X",(ulong)frame_bytes[10],
        (ulong)frame_bytes[0xb],(ulong)frame_bytes[0xc],(ulong)frame_bytes[0xd],
        (ulong)frame_bytes[0xe],(ulong)frame_bytes[0xf]);
sprintf(ssid_str,"%2.2X:%2.2X:%2.2X:%2.2X:%2.2X:%2.2X",(ulong)frame_bytes[0x10],
        (ulong)frame_bytes[0x11],(ulong)frame_bytes[0x12],(ulong)frame_bytes[0x13],
        (ulong)frame_bytes[0x14],(ulong)frame_bytes[0x15]);
                 /* check version, must be 0

                     */
if ((bytes[radiotap_len_c] & 0xc) != 0) {
  return;
}
frame_type = bytes[radiotap_len_c] & 0xf0;
frame_len_maybe = ((uVar18 & 0xffff) - (uint)radiotap_len) - frame_len_maybe;
uVar4 = (undefined)channel;
if (frame_type == 0xd0) {
  if (frame_bytes[0x18] != '\x04' || frame_bytes[0x19] != '\t') {
    return;
```

Figure 36 - ds230 firmware: Wi-Fi frame parsing.

(*0xd*). Then the code flow changes slightly depending on the frame sub-type:

- If the frame sub-type is beacon, the code starts parsing the vendor-specific IEs. The code will loop over all vendor-specific IEs present in the beacon until the end of the frame or until it finds an IE that matches the ODID specifications.[33] In order to be compliant with the specifications, the ODID IE must be formatted in the following way:
  - *0xdd*: vendor-specific IE.
  - *0xXX*: IE length (maximum 255 bytes).
  - *0xFA 0x0B 0xB 0x0D*: ASD-STAN specific IE.
  - *0xYY*: ODID message counter.
  - *0xF0*: ODID message pack.
  - *0x19*: each message block in the message pack must have a length of 25 bytes.
  - *0x0N*: number of message blocks in the message pack (maximum 9),
  - Then *25 * N* bytes (25 bytes for each ODID message block).
- Instead, if the frame sub-type is NaN, with the help of function *parse_odid_nan() (0x430418)* (Figure 37), the code checks that the mandatory values in the frame are set as defined in the specification.

- The code continues by passing the content of the ODID message pack to function *odid_message_process_pack() (0x43e38c)* which fills and returns an *ODID_UAS_Data* data structure containing the received ODID data.[34]
- At this point, as discussed in section 4.3.1.1, the obtained *ODID_UAS_Data* data structure is passed to function *copy_odid_uas_data_in_global_ll()* and inserted into *g_data*.
- The updated entry in *g_data* is then filled with additional information like a timestamp, the transmitter address, RSSI, channel, counter and reception type (see Figure 33).
- Finally, the Wi-Fi Sniffing Thread synchronizes with the Wi-Fi Channel Hopping Thread to update the list of channels that must be monitored by the "second" Wi-Fi interface.


```
uVar12 = parse_odid_nan((odid_data_entry *)odid_data_entry_p,tx_addr,frame_bytes,(ulong)tag_len)
;
if ((int)uVar12 != 0) goto end;
pthread_mutex_lock((pthread_mutex_t *)&g_capt_mutex);
glob_odid_e_p = copy_odid_uas_data_in_global_ll(transmitter_addr_str,odid_data_entry_p);
if (glob_odid_e_p != (global_odid_entry *)0x0) {
  uVar9 = get_epoch_ms?();
  glob_odid_e_p->timestamp = uVar9;
  strcpy(glob_odid_e_p->tx_addr_str,transmitter_addr_str);
  glob_odid_e_p->RSSI = (uint8_t)uVar18;
  glob_odid_e_p->channel = uVar3;
  glob_odid_e_p->rx_type = 2;
_00408c90:
```
Figure 37 - ds230 firmware: Wi-Fi frame parsing.

[32] This makes sense considering that the ODID messages can only be contained in beacon or NaN frames which are both management.

[33] During our reverse engineering activity, we noticed that the length of the vendor-specific IE (the second byte in each element) is trusted by the code and is used for advancing to the next element. It is also used for computing the length of the ODID element that is then passed to function located at 0x43238c. This means it is possible to inject beacon frames with forged IE lengths and force the code to read content after the actual frame. However, during our experiments, we noticed that even if we are able to force the code to read past the actual frame length, the memory read is part of a buffer managed by *libcap* and memory mapped, so it was not possible to cause a segmentation fault that could have led to a Denial-of-Service attack.

[34] Both *odid_message_process_pack()* and *ODID_UAS_Data* have been introduced in the previous chapter and are part of the ODID reference implementation library API.

4.3.1.3 Bluetooth Sniffing and Parsing

As already discussed, Bluetooth sniffing in the DroneScout ds230 is handled by an ESP32-C3 SoC.[35] As we focused on Wi-Fi ODID broadcasting and injection in this research, we will not detail how Bluetooth sniffing works on the DroneScout. However, for a high-level overview of how this functionality works:

- The ESP32-C3 SoC runs a custom firmware responsible for sniffing Bluetooth advertisements (4.x legacy and 5.0 extended).
- When an advertisement with an ODID payload is detected, the custom firmware extracts the ODID RID information and prepares a JSON message that will be transmitted to the main board through the UART interface.
- On the main board, *dronescout.arm64* is in charge of setting up the UART communication and receiving and parsing the JSON data generated by the custom firmware running on the ESP32-C3 SoC.
- From the JSON, a properly encoded stream of bytes is extracted and passed to the function *decodeOpenDroneID() (0x43aa84)*. This function is part of the ODID reference implementation API and returns an *ODID_UAS_Data* data structure. From here on, the code proceeds in the same way as it does for Wi-Fi (i.e., *copy_odid_uas_data_in_global_ll()*, etc.).

4.4 DroneScout ds230 Vulnerabilities

While reverse engineering the DroneScout ds230 appliance we found three distinct vulnerabilities, listed below. The manufacturer BlueMark Innovations has, upon discovery, solved the vulnerabilities in firmware version 20230605-1350 released on June 5, 2023.

Critical Risk

CVE-2023-31191:
Information Loss or Omission (CWE-221)

Base Score:
9.3

CVSS 3.1 Vector:
CVSS:3.1/AV:A/AC:L/PR:N/UI:N/S:C/C:N/I:H/A:H

High Risk

CVE-2023-31190:
Improper Authentication (CWE-287)

Base Score:
8.1

CVSS 3.1 Vector:
CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H

Medium Risk

CVE-2023-29156:
Information Loss or Omission (CWE-221)

Base Score:
4.7

CVSS 3.1 Vector:
VSS:3.1/AV:A/AC:H/PR:N/UI:N/S:C/C:N/I:L/A:L

[35] The firmware running on the ESP32 SoC handles the differences between Bluetooth 4.x legacy advertisement and Bluetooth 5.0 extended advertisement making the difference between the two communication mechanisms transparent to *dronescout.arm64*.

Two of the vulnerabilities we discovered (CVE-2023-31191 and CVE-2023-29156) could allow an attacker to spoof RID information, forcing the DroneScout ds230 to drop RID information transmitted by legitimately communicating drones. Consequentially, an attacker could inject fake locations associated with the legitimate drone detected by the DroneScout.

Apart from the technicalities which are discussed in the next sections, the difference between the two vulnerabilities is that CVE-2023-21156 is probabilistic and the attack's success rate is around 90%, which is why this CVE is classified as medium risk. Meanwhile, CVE-2023-31191, classified as critical, is deterministic and attack success is guaranteed when it is exploited.

CVE-2023-31190 demonstrates the capability to install malicious firmware updates on the DroneScout appliance. The crafted update could contain arbitrary files which, in turn, could lead the attacker to gain administrative privileges on the underlying Linux operating system. We will not discuss this vulnerability in here.[36] Instead, we will focus on the other two vulnerabilities, which are more related to the RID functionalities of the DroneScout.

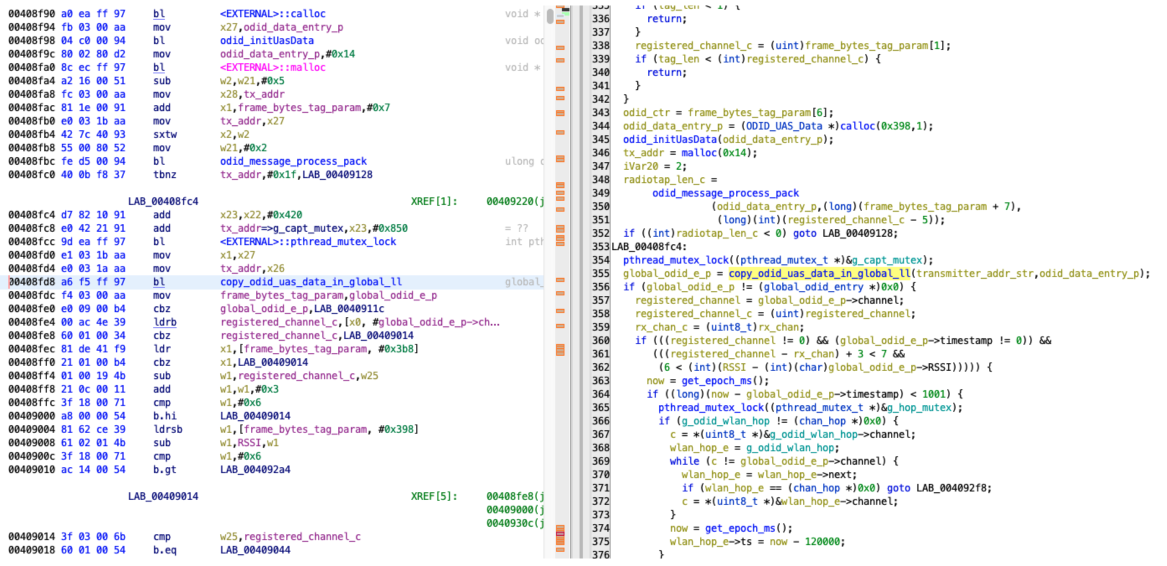
4.4.1 Vulnerability Analysis: CVE-2023-29156

With this vulnerability, an attacker can force the DroneScout receiver to drop real RID information and instead generate and transmit JSON encoded MQTT messages containing fake RID information. Consequently, the system integrator running MQTT broker will have no access to the RID information of the real drones. To trigger the vulnerability, the attacker must inject ODID messages with spoofed source MAC addresses at the right time in order to overwrite the in-memory RID information stored by the DroneScout main executable (*dronescout.arm64*). This CVE affects the DroneScout ds230 appliance firmware 20211210-1627 and later versions with default configuration.

As introduced in the previous sections, the behavior of *dronescout.arm64* can be tweaked by modifying the configuration file dronescout.conf. We are interested in the option *"transmit_mode"* under section *"mqtt"*. When this option is set to 1, as in the default configuration, *dronescout.arm64* performs a throttling of the MQTT messages and does not generate and publish an MQTT message for each ODID message received and correctly parsed.

Instead, the executable behaves as follows: when a new ODID message is received from a legitimate drone D with MAC address MD, the behavior follows the sequence presented in section 4.3.1.2, "Wi-Fi Sniffing and Parsing". In short, the RID data (RIDD) contained in the message is parsed into a structure of type *ODID_UAS_Data* (defined by the ODID framework). Then RIDD and MD are passed to function *copy_odid_uas_data_in_global_ll* (note that MD is used by *dronescout.arm64* for identifying D). An example of this is shown in Figure 38, which reports a snippet of function *wlan_pcap_handler (0x408708)* that handles the sniffed Wi-Fi frames.

[36] More information about this CVE can be found in this blog published by Nozomi Networks Labs: **nozominetworks.com/blog/nozomi-networks-discovers-three-vulnerabilities-affecting-bluemark-dronescout-ds230-remote-id-receiver**.

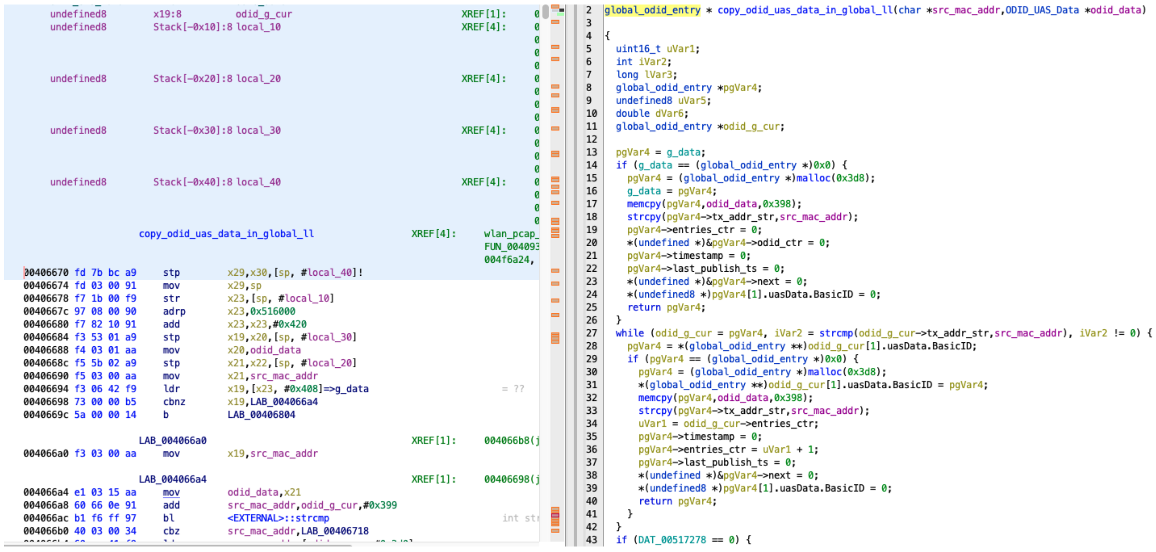


```
00408f90 a0 ea ff 97     bl       <EXTERNAL>::calloc                            void *
00408f94 fb 03 00 aa     mov      x27,odid_data_entry_p
00408f98 04 c0 00 94     bl       odid_initUasData                              void oc
00408f9c 80 02 80 d2     mov      odid_data_entry_p,#0x14
00408fa0 8c ec ff 97     bl       <EXTERNAL>::malloc                            void *
00408fa4 a2 16 00 51     sub      w2,w21,#0x5
00408fa8 fc 03 00 aa     mov      x28,tx_addr
00408fac 81 1e 00 91     add      x1,frame_bytes_tag_param,#0x7
00408fb0 e0 03 1b aa     mov      tx_addr,x27
00408fb4 42 7c 40 93     sxtw     x2,w2
00408fb8 55 00 80 52     mov      w21,#0x2
00408fbc fe d5 00 94     bl       odid_message_process_pack                     ulong c
00408fc0 40 0b f8 37     tbnz     tx_addr,#0x1f,LAB_00409128

                     LAB_00408fc4                          XREF[1]:     00409220(j
00408fc4 d7 82 10 91     add      x23,x22,#0x420
00408fc8 e0 42 21 91     add      tx_addr=>g_capt_mutex,x23,#0x850              = ??
00408fcc 9d ea ff 97     bl       <EXTERNAL>::pthread_mutex_lock                int pth
00408fd0 e1 03 1b aa     mov      x1,x27
00408fd4 e0 03 1a aa     mov      tx_addr,x26
00408fd8 a6 f5 ff 97     bl       copy_odid_uas_data_in_global_ll               global_
00408fdc f4 03 00 aa     mov      frame_bytes_tag_param,global_odid_e_p
00408fe0 e0 09 00 b4     cbz      global_odid_e_p,LAB_0040911c
00408fe4 00 ac 4e 39     ldrb     registered_channel_c,[x0, #global_odid_e_p->ch…
00408fe8 60 01 00 34     cbz      registered_channel_c,LAB_00409014
00408fec 81 de 41 f9     ldr      x1,[frame_bytes_tag_param, #0x3b8]
00408ff0 21 01 00 b4     cbz      x1,LAB_00409014
00408ff4 01 00 19 4b     sub      w1,registered_channel_c,w25
00408ff8 21 0c 00 11     add      w1,w1,#0x3
00408ffc 3f 18 00 71     cmp      w1,#0x6
00409000 a8 00 00 54     b.hi     LAB_00409014
00409004 81 62 ce 39     ldrsb    w1,[frame_bytes_tag_param, #0x398]
00409008 61 02 01 4b     sub      w1,RSSI,w1
0040900c 3f 18 00 71     cmp      w1,#0x6
00409010 ac 14 00 54     b.gt     LAB_004092a4

                     LAB_00409014                          XREF[5]:     00408fe8(j
                                                                        00409000(j
                                                                        0040930c(j
00409014 3f 03 00 6b     cmp      w25,registered_channel_c
00409018 60 01 00 54     b.eq     LAB_00409044
```

```
      return;
    }
    registered_channel_c = (uint)frame_bytes_tag_param[1];
    if (tag_len < (int)registered_channel_c) {
      return;
    }
  }
  odid_ctr = frame_bytes_tag_param[6];
  odid_data_entry_p = (ODID_UAS_Data *)calloc(0x398,1);
  odid_initUasData(odid_data_entry_p);
  tx_addr = malloc(0x14);
  iVar20 = 2;
  radiotap_len_c =
       odid_message_process_pack
                 (odid_data_entry_p,(long)(frame_bytes_tag_param + 7),
                  (long)(int)(registered_channel_c - 5));
  if ((int)radiotap_len_c < 0) goto LAB_00409128;
LAB_00408fc4:
  pthread_mutex_lock((pthread_mutex_t *)&g_capt_mutex);
  global_odid_e_p = copy_odid_uas_data_in_global_ll(transmitter_addr_str,odid_data_entry_p);
  if (global_odid_e_p != (global_odid_entry *)0x0) {
    registered_channel = global_odid_e_p->channel;
    registered_channel_c = (uint)registered_channel;
    rx_chan_c = (uint8_t)rx_chan;
    if (((registered_channel != 0) && (global_odid_e_p->timestamp != 0)) &&
       (((registered_channel - rx_chan) + 3 < 7 &&
        (6 < (int)(RSSI - (int)(char)global_odid_e_p->RSSI))))) {
      now = get_epoch_ms();
      if ((long)(now - global_odid_e_p->timestamp) < 1001) {
        pthread_mutex_lock((pthread_mutex_t *)&g_hop_mutex);
        if (g_odid_wlan_hop != (chan_hop *)0x0) {
          c = *(uint8_t *)&g_odid_wlan_hop->channel;
          wlan_hop_e = g_odid_wlan_hop;
          while (c != global_odid_e_p->channel) {
            wlan_hop_e = wlan_hop_e->next;
            if (wlan_hop_e == (chan_hop *)0x0) goto LAB_004092f8;
            c = *(uint8_t *)&wlan_hop_e->channel;
          }
          now = get_epoch_ms();
          wlan_hop_e->ts = now - 120000;
        }
```

Figure 38 - ODID message parsing for Wi-Fi.

The function *copy_odid_uas_data_in_global_ll (0x4065b8)* copies *RID*$_D$ into a global linked list that contains an entry for each known *M*$_D$ (i.e., a drone from which an ODID message was previously received). A snippet of this function is shown in Figure 39: if the global linked list already contains an entry for *M*$_D$, then that entry is updated with the new *RID*$_D$. Otherwise, a new entry for the previously unknown *M*$_D$ address is created and filled with the new *RID*$_D$. This means that for a given *M*$_D$ (i.e., drone), the DroneScout does not keep a history of the received *RID*$_D$. Instead, only the more recent *RID*$_D$ is kept in memory. In the following we call *RID*$_M$ the most recent *RID*$_D$ for entry *M*$_D$.



```
        undefined8        x19:8         odid_g_cur                  XREF[1]:     0
        undefined8        Stack[-0x10]:8 local_10                   XREF[4]:     0
                                                                                 0
                                                                                 0
                                                                                 0
        undefined8        Stack[-0x20]:8 local_20                   XREF[4]:     0
                                                                                 0
                                                                                 0
                                                                                 0
        undefined8        Stack[-0x30]:8 local_30                   XREF[4]:     0
                                                                                 0
                                                                                 0
                                                                                 0
        undefined8        Stack[-0x40]:8 local_40                   XREF[4]:     0
                                                                                 0
                                                                                 0
                                                                                 0
                      copy_odid_uas_data_in_global_ll          XREF[4]:     wlan_pcap_
                                                                            FUN_00409З
                                                                            004f6a24,
00406670 fd 7b bc a9     stp      x29,x30,[sp, #local_40]!
00406674 fd 03 00 91     mov      x29,sp
00406678 f7 1b 00 f9     str      x23,[sp, #local_10]
0040667c 97 08 00 90     adrp     x23,0x516000
00406680 f7 82 10 91     add      x23,x23,#0x420
00406684 f3 53 01 a9     stp      x19,x20,[sp, #local_30]
00406688 f4 03 01 aa     mov      x20,odid_data
0040668c f5 5b 02 a9     stp      x21,x22,[sp, #local_20]
00406690 f5 03 00 aa     mov      x21,src_mac_addr
00406694 f3 06 42 f9     ldr      x19,[x23, #0x408]=>g_data                     = ??
00406698 73 00 00 b5     cbnz     x19,LAB_004066a4
0040669c 5a 00 00 14     b        LAB_00406804

                      LAB_004066a0                          XREF[1]:     004066b8(j
004066a0 f3 03 00 aa     mov      x19,src_mac_addr

                      LAB_004066a4                          XREF[1]:     00406698(j
004066a4 e1 03 15 aa     mov      odid_data,x21
004066a8 60 66 0e 91     add      src_mac_addr,odid_g_cur,#0x399
004066ac b1 f6 ff 97     bl       <EXTERNAL>::strcmp                            int str
004066b0 40 03 00 34     cbz      src_mac_addr,LAB_00406718
```

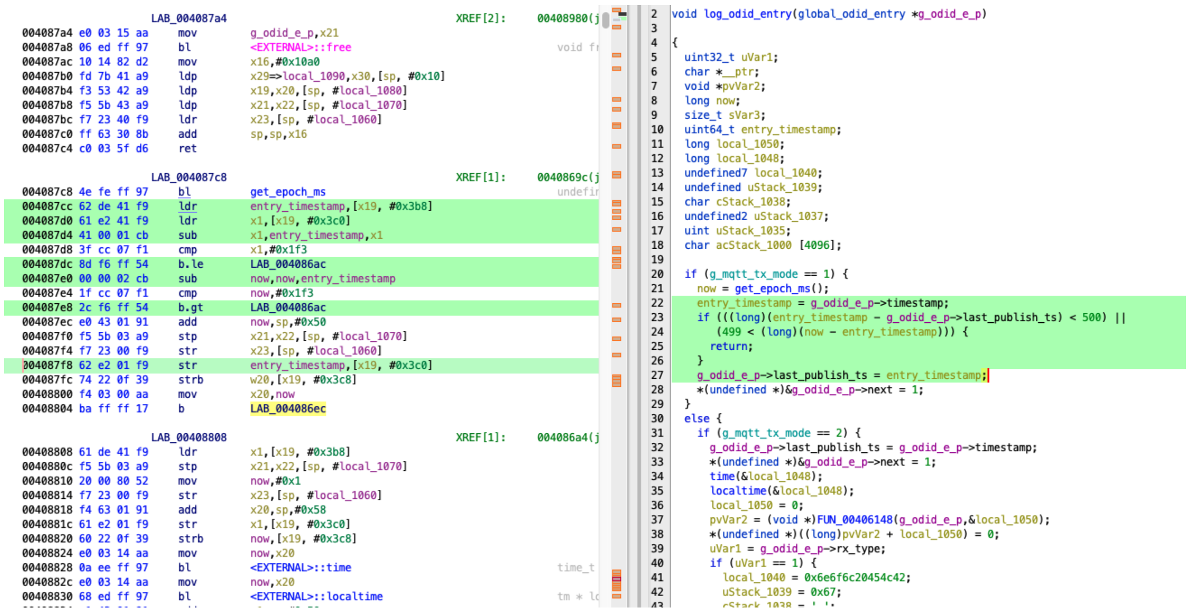
```
global_odid_entry * copy_odid_uas_data_in_global_ll(char *src_mac_addr,ODID_UAS_Data *odid_data)

{
  uint16_t uVar1;
  int iVar2;
  long lVar3;
  global_odid_entry *pgVar4;
  undefined8 uVar5;
  double dVar6;
  global_odid_entry *odid_g_cur;

  pgVar4 = g_data;
  if (g_data == (global_odid_entry *)0x0) {
    pgVar4 = (global_odid_entry *)malloc(0x3d8);
    g_data = pgVar4;
    memcpy(pgVar4,odid_data,0x398);
    strcpy(pgVar4->tx_addr_str,src_mac_addr);
    pgVar4->entries_ctr = 0;
    *(undefined *)&pgVar4->odid_ctr = 0;
    pgVar4->timestamp = 0;
    pgVar4->last_publish_ts = 0;
    *(undefined *)&pgVar4->next = 0;
    *(undefined8 *)pgVar4[1].uasData.BasicID = 0;
    return pgVar4;
  }
  while (odid_g_cur = pgVar4, iVar2 = strcmp(odid_g_cur->tx_addr_str,src_mac_addr), iVar2 != 0) {
    pgVar4 = *(global_odid_entry **)odid_g_cur[1].uasData.BasicID;
    if (pgVar4 == (global_odid_entry *)0x0) {
      pgVar4 = (global_odid_entry *)malloc(0x3d8);
      *(global_odid_entry **)odid_g_cur[1].uasData.BasicID = pgVar4;
      memcpy(pgVar4,odid_data,0x398);
      strcpy(pgVar4->tx_addr_str,src_mac_addr);
      uVar1 = odid_g_cur->entries_ctr;
      pgVar4->timestamp = 0;
      pgVar4->entries_ctr = uVar1 + 1;
      pgVar4->last_publish_ts = 0;
      *(undefined *)&pgVar4->next = 0;
      *(undefined8 *)pgVar4[1].uasData.BasicID = 0;
      return pgVar4;
    }
  }
  if (DAT_00517278 == 0) {
```

Figure 39 - ds230 firmware: *copy_odid_uas_data_in_global.*

Then, each entry in the global linked list has two fields containing timestamps (see Figure 33): *"timestamp"*, which is the timestamp in milliseconds of when *RID_M* was last updated (which is more or less the time when the latest ODID message was received from *M_D*) and *"last_publish_ts"* which is the timestamp of the last RIDM transmitted to the MQTT broker (i.e., when RIDM is published over MQTT, the *"timestamp"* field of entry MD is copied over its *"last_publish_ts"*. This is shown in Figure 40).

Finally, once the global linked list has been updated, the function *log_odid_entry (0x4084c0)* is called. This function, among other things, decides when to generate and transmit a new JSON message containing *RID_M* to the MQTT broker. In Figure 40, lines 20-29 of the decompiled code show that, when option *"transmit_mode"* is set to 1, the function does not always publish *RID_M* over MQTT. Instead, *RID_M* is dropped (i.e., the function returns) if it was last published less than 500 milliseconds before. Below is the code snippet of interest:

```
entry_timestamp = g_odid_e_p >timestamp;

 if (((long)(entry_timestamp - g_odid_e_p->last_publish_ts) < 500) ||

      (499 < (long)(now - entry_timestamp))) {

     return;

 }
```



```
                        LAB_004087a4                              XREF[2]:     00408980(j
004087a4 e0 03 15 aa     mov        g_odid_e_p,x21
004087a8 06 ed ff 97     bl         <EXTERNAL>::free                            void fr
004087ac 10 14 82 d2     mov        x16,#0x10a0
004087b0 fd 7b 41 a9     ldp        x29=>local_1090,x30,[sp, #0x10]
004087b4 f3 53 42 a9     ldp        x19,x20,[sp, #local_1080]
004087b8 f5 5b 43 a9     ldp        x21,x22,[sp, #local_1070]
004087bc f7 23 40 f9     ldr        x23,[sp, #local_1060]
004087c0 ff 63 30 8b     add        sp,sp,x16
004087c4 c0 03 5f d6     ret

                        LAB_004087c8                              XREF[1]:     0040869c(j
004087c8 4e fe ff 97     bl         get_epoch_ms                                undefin
004087cc 62 de 41 f9     ldr        entry_timestamp,[x19, #0x3b8]
004087d0 61 e2 41 f9     ldr        x1,[x19, #0x3c0]
004087d4 41 00 01 cb     sub        x1,entry_timestamp,x1
004087d8 3f cc 07 f1     cmp        x1,#0x1f3
004087dc 8d f6 ff 54     b.le       LAB_004086ac
004087e0 00 00 02 cb     sub        now,now,entry_timestamp
004087e4 1f cc 07 f1     cmp        now,#0x1f3
004087e8 2c f6 ff 54     b.gt       LAB_004086ac
004087ec e0 43 01 91     add        now,sp,#0x50
004087f0 f5 5b 03 a9     stp        x21,x22,[sp, #local_1070]
004087f4 f7 23 00 f9     str        x23,[sp, #local_1060]
004087f8 62 e2 01 f9     str        entry_timestamp,[x19, #0x3c0]
004087fc 74 22 0f 39     strb       w20,[x19, #0x3c8]
00408800 f4 03 00 aa     mov        x20,now
00408804 ba ff ff 17     b          LAB_004086ec

                        LAB_00408808                              XREF[1]:     004086a4(j
00408808 61 de 41 f9     ldr        x1,[x19, #0x3b8]
0040880c f5 5b 03 a9     stp        x21,x22,[sp, #local_1070]
00408810 20 00 80 52     mov        now,#0x1
00408814 f7 23 00 f9     str        x23,[sp, #local_1060]
00408818 f4 63 01 91     add        x20,sp,#0x58
0040881c 61 e2 01 f9     str        x1,[x19, #0x3c0]
00408820 60 22 0f 39     strb       now,[x19, #0x3c8]
00408824 e0 03 14 aa     mov        now,x20
00408828 0a ee ff 97     bl         <EXTERNAL>::time                            time_t
0040882c e0 03 14 aa     mov        now,x20
00408830 68 ed ff 97     bl         <EXTERNAL>::localtime                       tm * lc
```

```
2   void log_odid_entry(global_odid_entry *g_odid_e_p)
3
4   {
5     uint32_t uVar1;
6     char *__ptr;
7     void *pvVar2;
8     long now;
9     size_t sVar3;
10    uint64_t entry_timestamp;
11    long local_1050;
12    long local_1048;
13    undefined7 local_1040;
14    undefined uStack_1039;
15    char cStack_1038;
16    undefined2 uStack_1037;
17    uint uStack_1035;
18    char acStack_1000 [4096];
19
20    if (g_mqtt_tx_mode == 1) {
21      now = get_epoch_ms();
22      entry_timestamp = g_odid_e_p->timestamp;
23      if (((long)(entry_timestamp - g_odid_e_p->last_publish_ts) < 500) ||
24         (499 < (long)(now - entry_timestamp))) {
25        return;
26      }
27      g_odid_e_p->last_publish_ts = entry_timestamp;
28      *(undefined *)&g_odid_e_p->next = 1;
29    }
30    else {
31      if (g_mqtt_tx_mode == 2) {
32        g_odid_e_p->last_publish_ts = g_odid_e_p->timestamp;
33        *(undefined *)&g_odid_e_p->next = 1;
34        time(&local_1048);
35        localtime(&local_1048);
36        local_1050 = 0;
37        pvVar2 = (void *)FUN_00406148(g_odid_e_p,&local_1050);
38        *(undefined *)((long)pvVar2 + local_1050) = 0;
39        uVar1 = g_odid_e_p->rx_type;
40        if (uVar1 == 1) {
41          local_1040 = 0x6e6f6c20454c42;
42          uStack_1039 = 0x67;
```

Figure 40 - ds230 firmware: MQTT throttling.

In other words, when RID data received from a certain MAC address M_D is published over MQTT, all the RID data received from the same MAC address in the following 500 milliseconds is discarded and never published by the DroneScout. This means that an attacker can force the DroneScout to never publish real RID_D by following these steps:

- Create an ODID message with spoofed source MAC address M_D and containing crafted RID data RID_{D1};
- Inject the ODID message 500ms or more after RID_M has been published over MQTT and before *D* transmits its next ODID message containing the real RID_D.

If the condition above is satisfied, what happens in *dronescout.arm64* is the following:

- ODID message transmitted by the attacker and containing crafted RID_{D1} is received,
- Crafted RID_{D1} is used to update RIDM,
- RID_M is put in a new JSON message transmitted to the MQTT broker,
- ODID message transmitted by *D* and containing real RID_D is received,
- Real RID_D is used to update RID_M,
- RID_M is NOT published over MQTT (i.e., RID_D is discarded).

If the attacker repeats the steps above, the MQTT broker of the system integrator using DroneScout will never receive real RID_D transmitted by drone *D*. In Chapter 6 we will see an example of attack scenario where this vulnerability is exploited and what kind of impacts it can have.

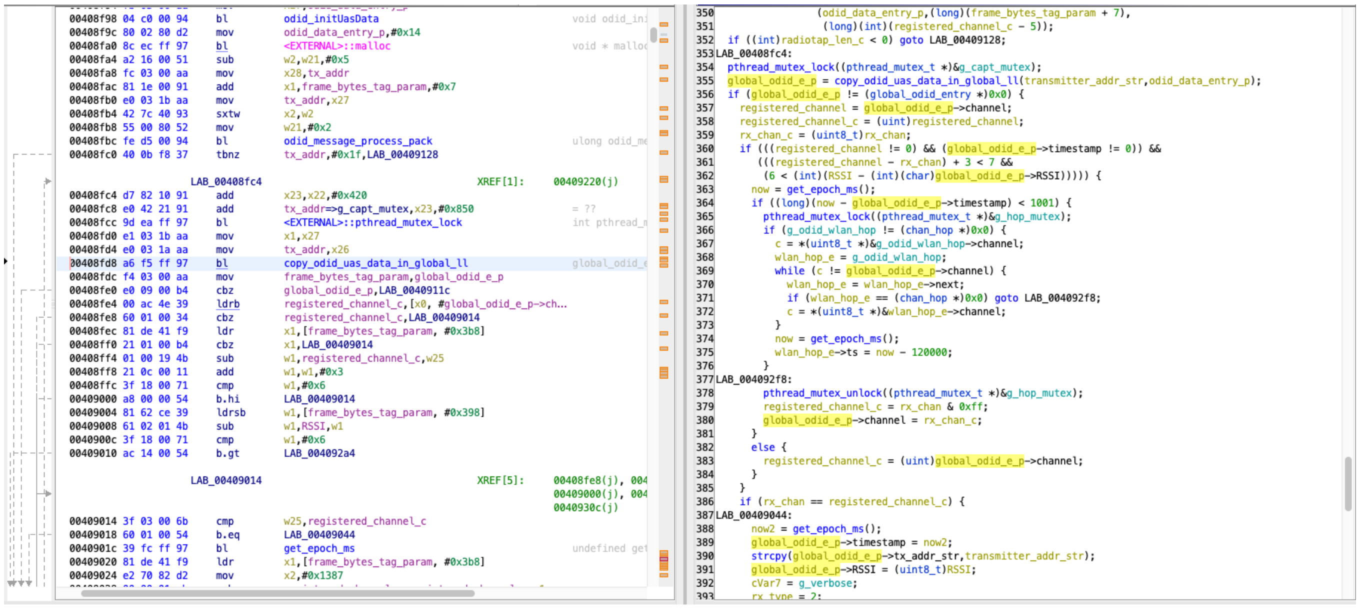
4.4.2 Vulnerability Analysis: CVE-2023-31191

The effect of this vulnerability is the same as the previous one: its exploitation can force the DroneScout receiver to drop real RID information and instead generate and transmit JSON encoded MQTT messages containing fake RID information. Consequently, the system integrator running the MQTT broker will have no access to the RID information of the real drones. This CVE affects the DroneScout ds230 appliance firmware 20230104-1650 and later versions with default configuration.

The difference is that with this vulnerability the attacker does not need to rely on the DroneScout's internal unknown timer states. Instead, the attacker only needs to inject high power messages (e.g., by using directional antennas) containing RID information on Wi-Fi channels adjacent to the ones used by real drones.

This vulnerability was made possible by a new algorithm introduced in DroneScout firmware version 20230104-1650 that, as stated in the official release notes: *"implement[s] an algorithm to suppress WLAN transponder signals on neighboring channels in case the RSSI is very strong. (If for instance a transponder is detected on channel 6 at -45 dBm, it will also be detected at channel 4, 5 7 and 8. The algorithm will suppress those detections on adjacent channels.)"*

In firmware 20230104-1650 the algorithm has been added to function *wlan_pcap_handler() (0x408a80)* and a decompiled snippet is shown in Figure 41.



```
00408f98 04 c0 00 94     bl        odid_initUasData                          void odid_ini
00408f9c 80 02 80 d2     mov       odid_data_entry_p,#0x14
00408fa0 8c ec ff 97     bl        <EXTERNAL>::malloc                        void * malloc
00408fa4 a2 16 00 51     sub       w2,w21,#0x5
00408fa8 fc 03 00 aa     mov       x28,tx_addr
00408fac 81 1e 00 91     add       x1,frame_bytes_tag_param,#0x7
00408fb0 e0 03 1b aa     mov       tx_addr,x27
00408fb4 42 7c 40 93     sxtw      x2,w2
00408fb8 55 00 80 52     mov       w21,#0x2
00408fbc fe d5 00 94     bl        odid_message_process_pack                 ulong odid_me
00408fc0 40 0b f8 37     tbnz      tx_addr,#0x1f,LAB_00409128

                     LAB_00408fc4                          XREF[1]:     00409220(j)
00408fc4 d7 82 10 91     add       x23,x22,#0x420
00408fc8 e0 42 21 91     add       tx_addr=>g_capt_mutex,x23,#0x850          = ??
00408fcc 9d ea ff 97     bl        <EXTERNAL>::pthread_mutex_lock            int pthread_m
00408fd0 e1 03 1b aa     mov       x1,x27
00408fd4 e0 03 1a aa     mov       tx_addr,x26
00408fd8 a6 f5 ff 97     bl        copy_odid_uas_data_in_global_ll           global_odid_e
00408fdc f4 03 00 aa     mov       frame_bytes_tag_param,global_odid_e_p
00408fe0 e0 09 00 b4     cbz       global_odid_e_p,LAB_0040911c
00408fe4 00 ac 4e 39     ldrb      registered_channel_c,[x0, #global_odid_e_p->ch…
00408fe8 60 01 00 34     cbz       registered_channel_c,LAB_00409014
00408fec 81 de 41 f9     ldr       x1,[frame_bytes_tag_param, #0x3b8]
00408ff0 21 01 00 b4     cbz       x1,LAB_00409014
00408ff4 01 00 19 4b     sub       w1,registered_channel_c,w25
00408ff8 21 0c 00 11     add       w1,w1,#0x3
00408ffc 3f 18 00 71     cmp       w1,#0x6
00409000 a8 00 00 54     b.hi      LAB_00409014
00409004 81 62 ce 39     ldrsb     w1,[frame_bytes_tag_param, #0x398]
00409008 61 02 01 4b     sub       w1,RSSI,w1
0040900c 3f 18 00 71     cmp       w1,#0x6
00409010 ac 14 00 54     b.gt      LAB_004092a4

                     LAB_00409014                          XREF[5]:     00408fe8(j), 004
                                                                        00409000(j), 004
                                                                        0040930c(j)
00409014 3f 03 00 6b     cmp       w25,registered_channel_c
00409018 60 01 00 54     b.eq      LAB_00409044
0040901c 39 fc ff 97     bl        get_epoch_ms                              undefined get
00409020 81 de 41 f9     ldr       x1,[frame_bytes_tag_param, #0x3b8]
00409024 e2 70 82 d2     mov       x2,#0x1387
```

```
                  (odid_data_entry_p,(long)(frame_bytes_tag_param + 7),
                   (long)(int)(registered_channel_c - 5));
  if ((int)radiotap_len_c < 0) goto LAB_00409128;
LAB_00408fc4:
  pthread_mutex_lock((pthread_mutex_t *)&g_capt_mutex);
  global_odid_e_p = copy_odid_uas_data_in_global_ll(transmitter_addr_str,odid_data_entry_p);
  if (global_odid_e_p != (global_odid_entry *)0x0) {
    registered_channel = global_odid_e_p->channel;
    registered_channel_c = (uint)registered_channel;
    rx_chan_c = (uint8_t)rx_chan;
    if (((registered_channel != 0) && (global_odid_e_p->timestamp != 0)) &&
       (((registered_channel - rx_chan) + 3 < 7 &&
        (6 < (int)(RSSI - (int)(char)global_odid_e_p->RSSI))))) {
      now = get_epoch_ms();
      if ((long)(now - global_odid_e_p->timestamp) < 1001) {
        pthread_mutex_lock((pthread_mutex_t *)&g_hop_mutex);
        if (g_odid_wlan_hop != (chan_hop *)0x0) {
          c = *(uint8_t *)&g_odid_wlan_hop->channel;
          wlan_hop_e = g_odid_wlan_hop;
          while (c != global_odid_e_p->channel) {
            wlan_hop_e = wlan_hop_e->next;
            if (wlan_hop_e == (chan_hop *)0x0) goto LAB_004092f8;
            c = *(uint8_t *)&wlan_hop_e->channel;
          }
          now = get_epoch_ms();
          wlan_hop_e->ts = now - 120000;
        }
LAB_004092f8:
        pthread_mutex_unlock((pthread_mutex_t *)&g_hop_mutex);
        registered_channel_c = rx_chan & 0xff;
        global_odid_e_p->channel = rx_chan_c;
      }
      else {
        registered_channel_c = (uint)global_odid_e_p->channel;
      }
    }
    if (rx_chan == registered_channel_c) {
LAB_00409044:
      now2 = get_epoch_ms();
      global_odid_e_p->timestamp = now2;
      strcpy(global_odid_e_p->tx_addr_str,transmitter_addr_str);
      global_odid_e_p->RSSI = (uint8_t)RSSI;
      cVar7 = g_verbose;
      rx_type = 2;
```

Figure 41 - ds230 firmware: adjacent channel suppression algorithm.

When a new drone (let's call it *D*) with source MAC address M_D is detected by the DroneScout (i.e., the DroneScout captures ODID messages transmitted by *D*), *dronescout.arm64* creates an in-memory global linked list entry (see Figure 33) indexed by M_D. Among other things, this list contains:

- M_D*->channel*: the Wi-Fi channel *C* where the ODID messages transmitted by *D* have been captured.
- M_D*->RSSI*: the Wi-Fi frame reception RSSI.

When a new Wi-Fi frame containing an ODID message is received from MAC address M_D on channel *C1* with RSSI *R*, the adjacent channel suppression algorithm implements the following logic:

- If *C1* is an adjacent channel (i.e., *((M_D->channel – C1) + 3) < 7*) and the reception RSSI *R* is greater than the last RSSI registered for M_D (i.e., *(R – M_D->RSSI) > 6*), then M_D*->channel* is set to *C1*, M_D*->RSSI* is set to *R* and the new ODID message is accepted and used to fill the RID information in the global entry corresponding to M_D.
- Otherwise, the new ODID message is accepted only if *C1* is the same as *C* (i.e., M_D*->channel*). In other words, the new Wi-Fi frames are accepted only if received on the same channel that was previously registered for M_D.

The description above is a simplified version of the algorithm which handles other details like the timestamps of the ODID messages received from M_D and the case of a first ODID message received from a specific drone *D*. Additional details have been omitted as they do not impact the vulnerability discussed here.

An attacker can exploit the algorithm described above with the following procedure:

- Create an ODID message with spoofed source MAC address M_D containing crafted RID data.
- Inject the ODID message with the spoofed source MAC address on an adjacent channel *Ca* (e.g., if the drone *D* is transmitting on channel 6, the attacker can transmit on channel 8).
- Transmit the Wi-Fi frames with high enough power that they are received by the DroneScout with an RSSI Ra that satisfies the condition discussed above (i.e., *((Ra – M_D->RSSI) > 6)*). This can be achieved by using a transmitter amplifier or a high gain/ directional antenna.

If the conditions above are satisfied, when the DroneScout receives the ODID message spoofed by the attacker it will set *M_D->channel* to *Ca* and *M_D->RSSI* to *Ra*. From this moment forward it will start dropping the ODID messages received from drone *D* on channel *C* and the MQTT broker will never receive real RID data transmitted by drone *D*. As we mentioned in the analysis of the previous CVE, we will see an example of attack scenario where this vulnerability is exploited and what kind of impacts it can have in Chapter 6.

5. DJI, OcuSync and DroneID Protocol

Before the introduction of RID standards, rules and policies, DJI, which owns roughly 70% of the global drone market, developed and deployed its own proprietary RID protocol called droneID. The telemetry information transported by droneID[37] is similar to what is contained in ODID messages. However, in contrast to ODID, DJI's RID protocol is broadcast by its drones using a radio protocol called OcuSync[38] which requires specialized hardware in order to be received and decoded.

OcuSync[39] is a proprietary and undocumented[40] protocol designed and developed by DJI for the purpose of providing a better communication range with respect to the standard Wi-Fi transmission technology. Nowadays, all modern drones and related equipment from DJI support OcuSync as a communication technology. To enable end users (who are mostly military and critical infrastructure) to detect its drones, DJI also developed a specialized device, called Aeroscope, designed specifically to receive droneID protocol transmitted on top of OcuSync and visualize the detected drones on a map.

As part of our research, Nozomi Networks Labs analyzed the behavior of the Aeroscope appliance and reverse engineered the OcuSync signal and the droneID protocol transmitted on top of it. From this analysis, we developed an injection framework based on Software Defined Radios (SDRs) for injecting OcuSync-based telemetry data (i.e. droneID packets). Given that OcuSync telemetry packets suffer from the same security weaknesses found in the ODID protocol (they are neither authenticated nor encrypted) the injection framework allowed us to develop attack scenarios against DJI's Aeroscope that resemble those based on ODID used against the DroneScout.

5.1 DJI Aeroscope Appliance

The Aeroscope is a specialized device developed by DJI which uses SDR-based hardware for receiving and decoding droneID telemetry packets transmitted on top of OcuSync Radio Frequency (RF) signals. DJI produces two models of this device: a stationary unit and a mobile unit (which has since been discontinued). Nozomi Networks Labs had the opportunity to test and analyze the behavior of the mobile unit.

[37] This includes the status, speed, altitude of the drone, user identification code and license information, GPS position of the drone and the takeoff, landing and operator positions.

[38] In this white paper, with OcuSync we refer both to the radio frequency (RF) signal at the physical layer and to the protocol transported on top of the RF signal.

[39] This research is based on OcuSync version 3.

[40] We used the paper "DJI droneIDs are not encrypted" by Conner Bender (**https://arxiv.org/pdf/2207.10795.pdf**) and the dji_droneid GitHub project as starting points for our activity.

The DJI Aeroscope mobile unit is based on an Android tablet with dedicated SDR hardware that is able to detect all DJI telemetry signals. Figure 42 shows a picture of the Aeroscope mobile unit.

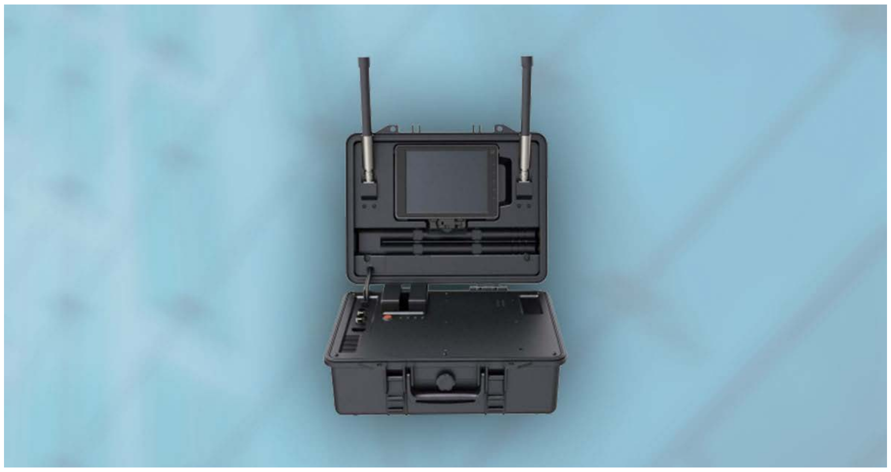


Figure 42 - A mobile Aeroscope unit used during Nozomi Networks Labs experiments, consisting of an Android tablet with dedicated USB hardware.

The Android tablet on the mobile Aeroscope unit is customized by DJI and provides a simple user interface to execute the drone monitoring application or show documentation. The drone monitoring application is composed of a world map and an information panel. When the application is restarted the map is clean. A sound plays every time a new drone is detected and the map updates with live drone positions, flight paths, operator and home (landing) position. After some time, the old drones are removed from the display to avoid clutter. Figure 43 shows an example of a map on the Aeroscope's drone monitoring application with two icons reporting detected drones.

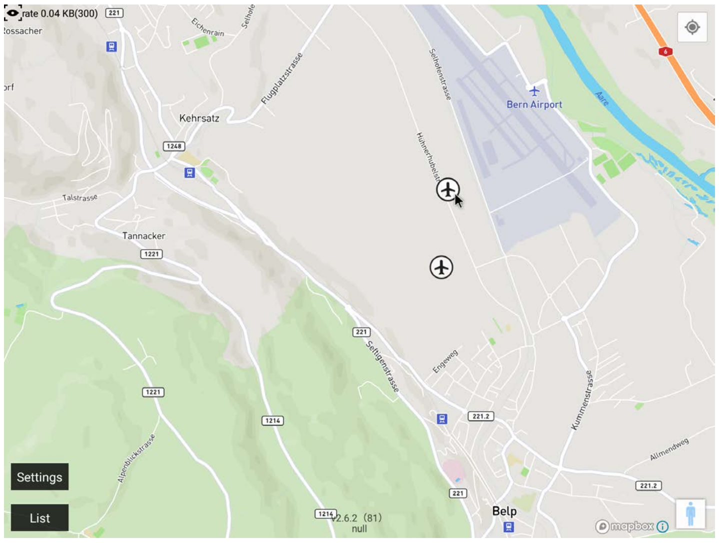
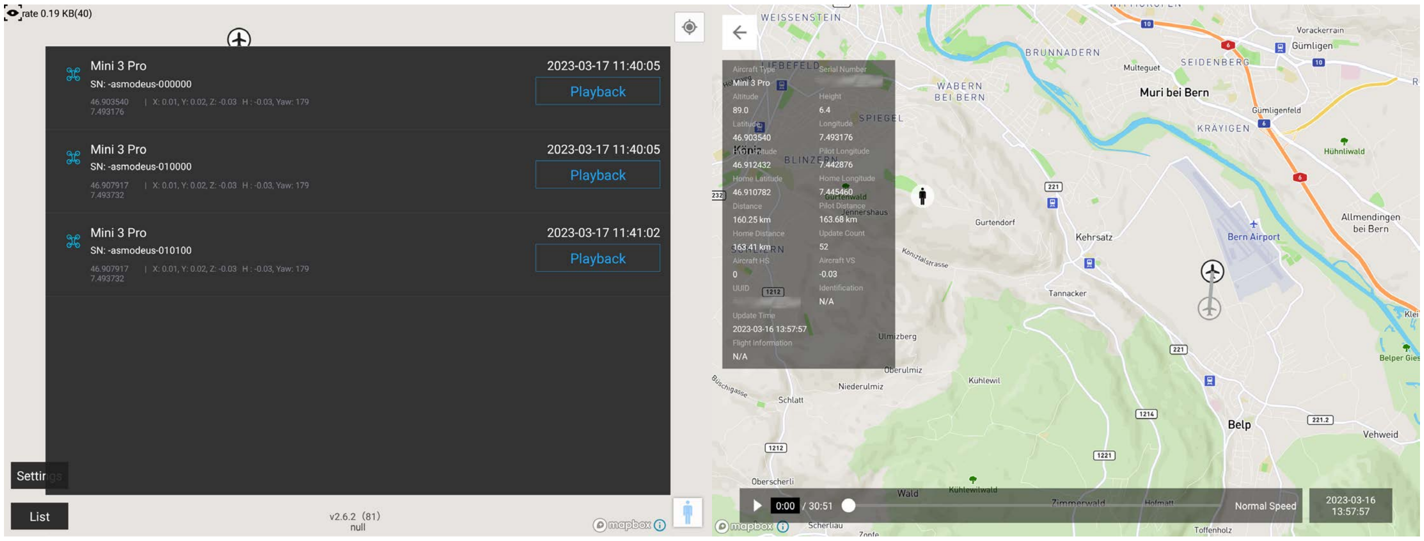


Figure 43 - DJI Aeroscope map interface showing the detected drone locations.

When one of the drones visualized on the map is selected, corresponding information is shown in a panel that appears at the bottom of the map. The "List" button (bottom left of Figure 44) allows a user to see a list of previously detected drones. Selecting a drone from this list will show it on the map with captured information and flight path (right of Figure 44). The drone's movement can be played back using the play button or a sliding bar control.


rate 0.19 KB(40)

Mini 3 Pro 2023-03-17 11:40:05
SN: -asmodeus-000000
46.903540 | X: 0.01, Y: 0.02, Z: -0.03 H : -0.03, Yaw: 179
7.493176

Mini 3 Pro 2023-03-17 11:40:05
SN: -asmodeus-010000
46.907917 | X: 0.01, Y: 0.02, Z: -0.03 H : -0.03, Yaw: 179
7.493732

Mini 3 Pro 2023-03-17 11:41:02
SN: -asmodeus-010100
46.907917 | X: 0.01, Y: 0.02, Z: -0.03 H : -0.03, Yaw: 179
7.493732

v2.6.2 (81)
null

Aircraft Type	Serial Number
Mini 3 Pro	[redacted]
Altitude	Height
89.0	6.4
Latitude	Longitude
46.903540	7.493176
Pilot Latitude	Pilot Longitude
46.912432	7.442876
Home Latitude	Home Longitude
46.910782	7.445460
Distance	Pilot Distance
160.25 km	163.68 km
Home Distance	Update Count
163.41 km	52
Aircraft HS	Aircraft VS
0	-0.03
UUID	Identification
[redacted]	N/A
Update Time	
2023-03-16 13:57:57	
Flight Information	
N/A	

Figure 44 - DJI Aeroscope, example showing the list of drones (left) and a detailed view of one of them (right).

5.2 Ocysync Communication Methods and Technologies

DJI uses the OcuSync protocol for different types of communications, including video streaming from the drone to the Remote Controller (RC), sending commands from the RC to the drone and broadcasting telemetry data from the drone. The OcuSync RF signal at the physical layer uses a different modulation scheme depending on the type of information transported. The rest of this section focuses on our analysis of the OcuSync signal used to broadcast telemetry packets and analysis of the format of the telemetry data (droneID) it contains.[41]

[41] In particular, our analysis is based on the OcuSync telemetry signal broadcast by the DJI Mini 3 Pro drone.

The OcuSync telemetry signal is modulated using an Orthogonal Frequency Division Multiplexing (OFDM). Several parameters must be known to correctly decode or encode the telemetry. The signal bandwidth is 10 MHz, while the duration is around 0.65 milliseconds and is broadcast on a frequency that can hop on 2.4 GHz or 5 GHz ISM bands. The hopping logic is unknown, but it seems to be related to channel quality. Table 5 shows the center frequencies where the DJI telemetry signal can be spotted. These frequencies have been confirmed by our RF spectrum observation of DJI Mini 3 Pro transmissions.

	Center Frequency of Telemetry Bands						
2.4 GHz	2.3995	2.4145	2.4295	2.4445	2.4595	2.4745	
5 GHz	5.7415	5.7565	5.7715	5.7865	5.8015	5.8165	5.8315

Table 5 - OcuSync channel center frequencies.

5.2.1 OcuSync Message Format and Encoding

OcuSync signal structure differs between DJI drone models. Figure 45 shows the spectrogram of one telemetry signal sent by the DJI Mini 3 Pro drone.

The telemetry signal is characterized by two synchronization symbols based on the Zadoff-Chu sequence with root of 600 (ZC600) and 147 (ZC147). The signal structure is composed by a fixed number of OFDM symbols with two of them carrying the ZC600 and ZC147 sequences.

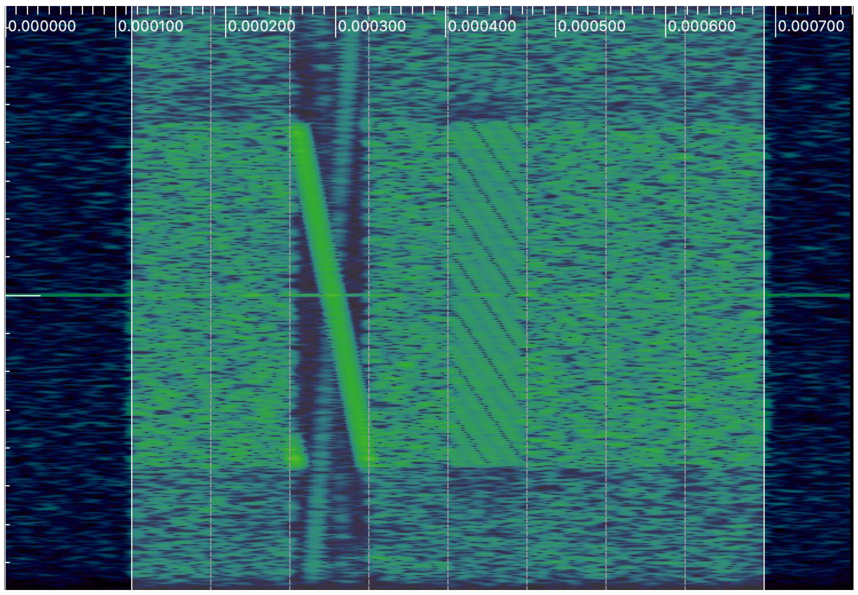


Figure 45 - Spectrogram of DJI Mini 3 Pro telemetry signal (10 MHz bandwidth, 643 microseconds).

There can be eight or nine symbols in a telemetry packet depending on the DJI drone model. Table 6 shows the two possible symbol sequences composing an OcuSync telemetry packet. The case with eight symbols is a special case where the first symbol is missing. The DJI Mini 3 Pro uses the eight-symbol packet format. The fourth symbol contains a ZC sequence with root set to 600 (ZC600) and the sixth symbol contains a ZC sequence with root set to 147 (ZC147).

Count	1	2	3	4	5	6	7	8	9
9 symbols	S1	S2	S3	ZC 600	S5	ZC 147	S7	S8	S9
8 symbols	-	S2	S3	ZC 600	S5	ZC 147	S7	S8	S9

Table 6 - OFDM symbols and Zadoff-Chu sequences in OcuSync telemetry signals.[42]

Each symbol is composed of a cyclic prefix (CP) followed by the OFDM symbol bitstream. The cyclic prefix is a copy of the last samples of the OFDM symbol to the beginning of the same symbol. The size of each OFDM symbol is fixed at 1024 samples but the length of the CP changes depending on the symbol index. There can either be a short CP with a duration of 4.69 microseconds or a long CP with a duration of 5.2 microseconds. These durations are the same as those used in LTE standard for normal and Extended CP. The duration of each OFDM symbol is about 66.67 microseconds (which corresponds to a 15KHz subcarrier spacing).

Table 7 shows the sample lengths of each cyclic prefix and OFDM symbol for the case of an OcuSync telemetry packet with nine symbols with the sampling rate is set to 15,360,000 Sample/s.

The total length of a telemetry signal is about 643 microseconds (or 9,880 samples with 15,360,000 Sample/s).

Count	1	2	3	4	5	6	7	8	9
cyclic prefix	80	72	72	72	72	72	72	72	80
OFDM symbol	1024	1024	1024	1024	1024	1024	1024	1024	1024

Table 7 - OFDM symbols and cyclic prefixes durations in OcuSync telemetry signals.

[42] Source paper “DJI drone IDs are not encrypted" available at **arxiv.org/pdf/2207.10795.pdf**.

5.2.2 Telemetry Signal Detection and Extraction

In order to detect the presence of an OcuSync telemetry signal broadcast by a DJI drone, the following three condition must be fulfilled:

1. The cross-correlation peak between the signal and a Zadoff-Chu (ZC) sequence with root 600 is above a threshold (see Figure 46)
2. The cross-correlation peak between the signal and a Zadoff-Chu (ZC) sequence with root 147 is above a threshold.
3. The difference in samples between the two correlation offsets is comparable with the expected distance of 2 OFDM symbols (2(1024 + 72) = 2,192 samples with 15,360,000 Sample/s).

The threshold must be empirically selected depending on the level of channel noise.

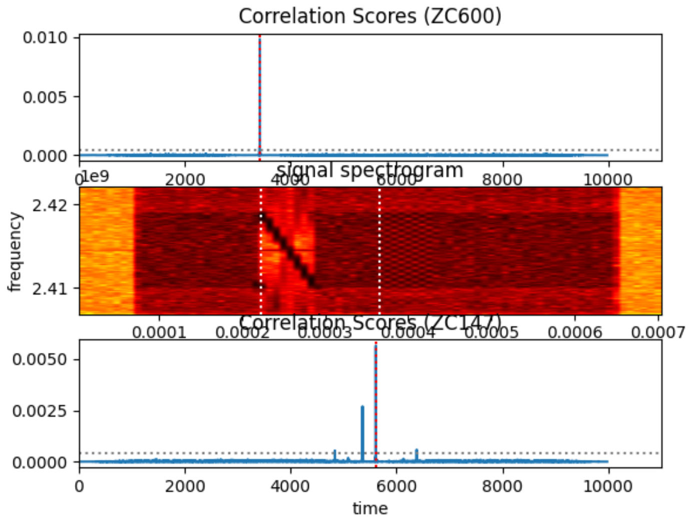


Figure 46 - Zadoff-Chu sequence correlation peaks on a signal with eight symbols (DJI Mini 3 Pro.

The correlation peak of the ZC600 sequence is near the fourth block of symbols. Starting from the sample offset where the correlation peak is detected, the beginning of the telemetry packet signal is detected by subtracting the length of the first three symbols (3*1024+80+2*72). Then the whole telemetry signal is extracted starting from the start to the end estimated using the total length of 9,880 samples (nine symbols 9*1024+7*72+2*80).

[43] The cross-correlation is a signal processing to compute the similarity of two data series and in function of the relative displacement between them.

$$= \sum_{m=-\infty}^{\infty} [\ ]\ \overline{[\ +\ ]}$$

Where , and ($\overline{[\ ]}$) complex conjugate of [ ]

5.2.3 Telemetry Signal Decoding

Depending on the precision in the frequency tuning of the SDR Card, the captured signal can present some frequency offset. In order to correctly decode the signal, the frequency offset must be removed and the signal must be centered in the frequency space. After the frequency correction, the signal can be safely filtered using a 10 MHz low pass filter. Then the samples are separated into the nine OFDM symbols with their relative cyclic prefix replicas, as shown in Figure 47. These replicas are used to perform a coarse frequency offset adjustment.

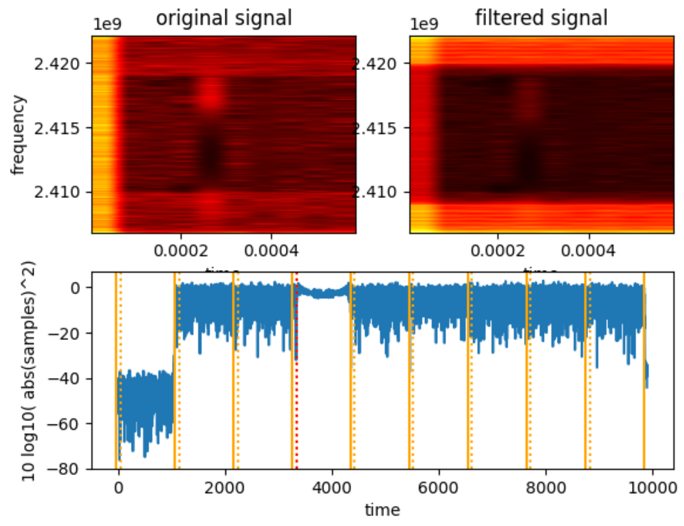


Figure 47 - OcuSync: centered signal (with symbol blocks) and filtered signal.

The first symbol (unused by DJI Mini 3 Pro), the fourth symbol (ZC600) and the sixth symbol (ZC147) are ignored. All other symbols are decoded as Quadrature Phase Shift Key (QPSK) modulated bits. The resulting bitstream is descrambled and the turbo codes are removed. The final bitstream contains a CRC code that is useful to check the validity of the extracted data.

5.2.4 Telemetry Packet Data Structure

The overall structure of an OcuSync telemetry packet is shown in Table 8.

The first field (Payload length) contains the length in bytes of Payload type and data. The second field (Payload type) contains the type of the structure contained in Payload data. The type can assume two different values: 0x10 when Payload data contains telemetry data or 0x11 when Payload data contains license plates. The last field contains a 16-bit CRC computed from the first field to the end of Payload data and can be used to check for possible decoding errors.

Byte Offset	Data Type	Value	Description
0x00	uint8	-	Payload length (N)
0x01	uint8	0x10 or 0x11	Payload type
0x02	N * uint8	-	Payload data
N+1	Uint16	-	CRC 16 bit

Table 8 - High-level OcuSync telemetry packet structure.

Table 9 shows an example of OcuSync packet content transmitted by a DJI Mini 3 Pro drone. The payload type is 0x10, as expected for a telemetry packet. The byte at offset 0x02 is the protocol version number – for the DJI Mini 3 Pro it is always 2.

Byte Offset	Data Type	Value	Description
0x00	uint8	0x58	payload length
0x01	uint16	0x10	packet type
0x02	uint8	0x02	version
0x03	uint16	0xD3 0x03	sequence number
0x05	uint16	0xF6 0x3F	status
0x07	char[16]	"ABCD1234EFGH5678"	serial number
0x17	uint32	0x12 0x7B 0x07 0x00	drone longitude
0x1A	uint32	0x8B 0x8E 0x55 0x00	drone latitude
0x1E	uint16	0xF8 0x02	altitude above sea
0x21	uint16	0xFE 0xFF	height from ground
0x23	uint16	0x03 0x00	speed north
0x25	uint16	0x06 0x00	speed east
0x27	uint16	0xF2 0xFF	speed up
0x29	uint16	0x88 0xBE	yaw
0x2B	uint64	0x45 0xF7 0x3F 0xCE 0x84 0x01 0x00 0x00	GPS timestamp
0x33	uint32	0x2B 0x8E 0x55 0x00	pilot latitude
0x37	uint32	0x22 0x7B 0x07 0x00	pilot longitude
0x3B	uint32	0x14 0x7B 0x07 0x00	home longitude
0x3F	uint32	0x81 0x8E 0x55 0x00	home latitude
0x43	uint8	0x49	model
0x44	uint8	0x13	length of UUID string
0x45	chars[]	"ABCD1234EFGH5"	UUID

Table 9 - Example of OcuSync telemetry packet content.

For telemetry packet protocol version 2, the python *unpack* format string is *'<BBBHH16siihhhhhhQiiiiBB19s'*. The GPS timestamp field contains the Unix epoch time multiplied by 1000 to reach millisecond granularity. The UUID field is a numerical string encoded in ASCII while the drone serial number is an alphanumerical field. The GPS coordinates can be extracted multiplying the longitude or latitude *int32* value by $\frac{180}{\pi\, 10^7}$.

5.3 Creating Fake DJI Telemetry Packets

Nozomi Networks built an OcuSync telemetry data injection framework based on the scripts made available by the open source dji_droneid project. This allowed us to artificially create OcuSync telemetry packets, giving the user complete control over the data written into the telemetry packet structure. The output generated by these scripts can be given as input to any SDR card able to output 10Mhz signals over 2.4 GHz or 5 GHz bands. Figure 48 reports the spectrum characteristics of the fake signals generated by these scripts.

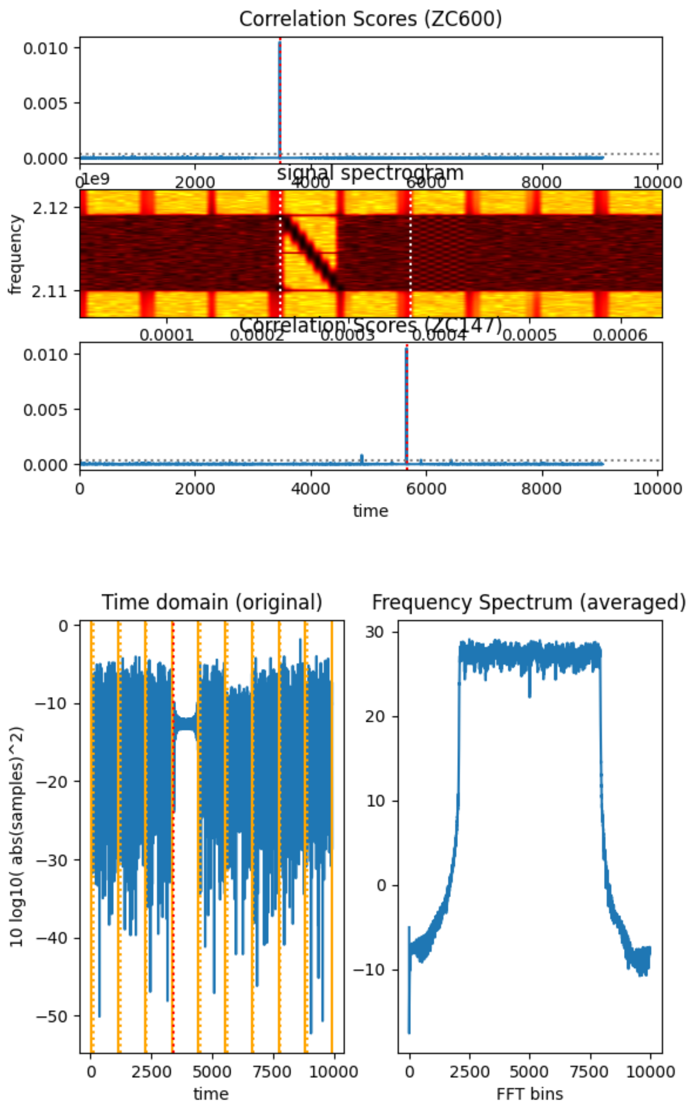


Figure 48 - Spectrum and Zadoff-Chu correlation peaks of a fake telemetry signal (left) and fake telemetry signal in time and frequency (right).

5.3.1 Replay of OcuSync Telemetry Packets

Beyond generating artificial OcuSync telemetry packets, another thing a potential attacker can do is capture (with an SDR) a legitimate OcuSync telemetry packet transmitted by a real DJI drone and later use the captured samples to replay it. For example, with a bladeRF SDR card, an OcuSync signal can be captured using the following commands (through the bladerf-cli tool):

```
set frequency rx1 2.3995Ghz
set samplerate rx1 15.36Mhz
set bandwidth rx1 10MHz
set agc rx1 on
rx config file=replay.sc16 n=10G
rx start
```

In the example above, the 10MHz signal is captured using 2.3995 as a central frequency and with a sample rate of 15.36 samples/s. The captured samples are then saved in the file replay.sc16 which uses a SC16 Q11 (signed complex 11bit) raw binary format where each sample requires 32 bits (4 bytes): 16 bits for the I component and 16 bits for the Q component (i.e., 61.44MB of data is required for each second of capture). At this point the file can be processed to extract the single telemetry packets, each one using more or less 40KB of data (39,520 bytes).

Otherwise, the attacker can replay the previously captured signals (saved in file replay.sc16) using the following commands:

```
set frequency tx1 2.3995Ghz
set samplerate tx1 15.36Mhz
set bandwidth tx1 10MHz
tx config file=replay.sc16
tx start
tx wait
```

Nozomi Networks Labs performed several experiments demonstrating that OcuSync telemetry data injected using replay techniques is actually received and correctly decoded by the Aeroscope; the Aeroscope map will show the drones that were present when the original signals were captured.

Our experiments also showed that the Aeroscope always considers the telemetry data valid, even when the drone location contained in the injected telemetry data is several thousand kilometers away from the actual position of the Aeroscope.

6. DroneScout ds230 Attack Scenarios

Nozomi Networks developed two frameworks that would allow us to implement attacks exploiting both the weaknesses in current RID standards as well as the vulnerabilities we discovered in ground station receivers. The first framework is based on standard Wi-Fi interfaces for receiving, parsing, and injecting ODID messages containing RID data. The second is based on SDR cards for receiving, decoding, encoding and injecting OcuSync signals containing DJI droneID protocol data.

The frameworks we developed would allow a potential attacker to inject crafted ODID and OcuSync telemetry data into ground station receivers. This fake data could be used to achieve different effects such as forging the presence of fake drones or hijacking the trajectories of legitimately flying drones, thereby impacting the security and reliability of RID systems.

This chapter introduces both our ODID injection framework and OcuSync telemetry data injection framework, then presents proof-of-concept attack scenarios designed to highlight the risks involved in RID systems as currently designed. The technical descriptions of the presented attacks will mostly focus on ODID protocol. However, except for differences in radio technologies and injection mechanism, the attacks similarly impact the DJI Aeroscope and its proprietary droneID protocol.

6.1 ODID Injection Framework

This injection framework takes advantage of both intrinsic ODID protocol weaknesses and vulnerabilities in the DroneScout ds230 ground station receiver, which we described in Chapter 4. In order to capture and analyze ODID traffic transmitted by consumer drones[44] and perform injection attacks, we created a python framework built on top of the ODID project's reference implementation library.[45]

The official ODID reference implementation library is implemented in C programming language. To create the required Python bindings, we leveraged the Binder project[46] to automatically generate as much bindings code as possible; we manually developed the missing code for parts of the reference implementation library that Binder was not able to support. Using Python programming language allowed us to quickly create different experimental ODID reception and injection prototypes.

[44] Specifically, for all our experiments we used DJI Mini 3 Pro drones equipped with firmware version 01.00.0150. With this firmware the drone transmitted Wi-Fi beacon frames containing Open Drone ID data by default, independent of the geographic location of the drone. So, for example, it was possible to perform experiments in Europe where, at the time of writing, it was not mandatory for a drone to transmit RID data.

[45] See section 3.3.3 in this white paper and **github.com/opendroneid/opendroneid-core-c**.

[46] Binder is a tool for automatic generation of Python bindings for C++11 projects using Pybind11 and Clang LibTooling libraries. See **github.com/RosettaCommons/binder**.

The resulting framework supports the following features:

- Support for multiple independent Wi-Fi interfaces to allow simultaneous ODID traffic monitoring and injection on different Wi-Fi channels[47];
- Real time asynchronous Wi-Fi traffic sniffing;
- Real time asynchronous ODID message injection;
- Support for different types of ODID (RID) oriented attacks like single drone emulation, multiple drone emulation, drone copycat (where the framework monitors for the presence of legitimate drones and injects identical ODID data but with fake locations), drone cloud (where the framework creates a cloud of emulated drones surrounding the location of a legitimate drone), DroneScout timer attack (which exploits CVE-2023-29156) and DroneScout adjacent channels attack (which exploits CVE-2023-31191);
- Drone visualization on a map. For this part, the framework leverages a locally deployed OpenStreetMap tiling server as a backend and a modified version of tar1090[48] for the frontend visualization.

We used this framework to implement all the proof-of-concept attack scenarios presented in the following sections.

6.2 OcuSync Telemetry Data Injection Framework

This Ocusync telemetry data injection framework was developed to receive and decode OcuSync signals containing DJI's proprietary telemetry data transmitted by the Mini 3 Pro drone, as well as perform OcuSync signal injection attacks against the DJI Aeroscope appliance. To do so, we created a python framework that directly controls an SDR card to perform the required tasks.

In particular, the framework has been fined tuned to work with the BladeRF SDR board. The framework can use one or more of these SDR cards to scan all possible wireless channels in search of DJI's OcuSync telemetry signals. The framework can also create parallel execution threads to create and transmit OcuSync signals containing forged telemetry data packets. For example, it can be configured to inject telemetry packets forging the presence of a DJI drone in a fixed and pre-selected location or to inject multiple telemetry packets, making multiple drones appear around the position of the most recently detected legitimate DJI drone (through OcuSync based telemetry).

[47] For our experiments we used multiple Alfa AWUS036ACH USB Wi-Fi dongles. However, any Wi-Fi network interface supporting monitor and injection mode is compatible with the framework.

[48] Web interface application originally designed for use with ADS-B decoders readsb / dump1090-fa. See **github.com/wiedehopf/tar1090**.

6.3 Attack Scenarios

Nozomi Networks Labs developed five different proof-of-concept attack scenarios targeting RID protocols (ODID and DJI's proprietary RID protocol) and compatible ground receivers (the DroneScout and Aeroscope). The attacks presented can be divided into two macro groups:

1. **Attacks that target the intrinsic weaknesses of current RID protocols.** For example, the lack of data authentication and encryption, as presented in Chapter 2 of this document. These attacks afflict any RID receiver and are independent of the actual ground station implementation.
2. **Attacks that take advantage of vulnerabilities present in the ground station receivers.** The attacks affecting the DroneScout ds230 ground station receiver presented in the Chapter 4 are an example. These attacks are not as portable as those in the first group because they are specific to each ground station and rely on vulnerabilities that can eventually be patched. However, they represent an interesting case study because they allow the attacker to obtain results that would not be possible just from leveraging the weaknesses of current RID protocols.

What follows is a brief description of the five attack scenarios developed by Nozomi Networks, divided into the two macro groups discussed above:

- **Generic RID attacks** (receiver implementation independent)[49]:
  - **Single drone forging:** this scenario showcases an attacker injecting ODID traffic to forge the presence of a single drone flying in a random trajectory in the airspace around the ground station receiver.
  - **Multiple drone forging:** similar to the previous scenario, but in this case the attacker injects ODID messages to emulate the presence of an arbitrary number of drones flying in random trajectories in the airspace around the ground station receiver.
  - **Drone cloud:** this scenario involves the attacker monitoring the airspace for ODID messages coming from real drones. When the real drone target is detected, the attacker injects ODID traffic to emulate the presence of fake drones flying in the airspace surrounding the legitimate drone.
- **Ground station-specific attacks:**
  - **DroneScout timer attack:** this scenario exploits CVE-2023-29156. The attacker injects spoofed ODID messages with the right timing to overwrite a legitimate drone's trajectory with a fake trajectory. In optimal conditions and using multiple attackers, the reliability of this attack can go above 90%, which we discuss below.
  - **DroneScout adjacent channels attack:** this scenario exploits CVE-2023-31191. The attacker injects high power spoofed ODID messages on a carefully selected Wi-Fi channel to overwrite a legitimate drone's trajectory with a fake trajectory. The attack can achieve 100% reliability through the use of directional antennas and a high gain transmitter.

[49] The same attack scenarios also apply to DJI's proprietary RID protocol and work against the Aeroscope. In this case the attacker must inject DroneID telemetry data using an SDR card capable of injecting OcuSync signals.

For all the presented proof-of-concept attacks targeting ODID or the DroneScout, we used the following equipment:

- **Target (legitimate):** DJI Mini 3 Pro drone with firmware version 01.00.0150.
- **Attack box:** Intel PC running our in-house developed ODID reception/injection framework and equipped with two ALFA AWUS036ACH USB Wi-Fi dongles.
- **ODID ground station receiver:** DroneScout ds230. As already discussed, the DroneScout is not a standalone device. So, we set up our own MQTT broker based on Mosquitto to receive the data captured by the DroneScout and leveraged our framework to visualize the real and fake drones on a map.

For DJI's proprietary RID protocol based on OcuSync and the Aeroscope, we used:

- **Target (legitimate):** DJI Mini 3 Pro drone with firmware version 01.00.0150.
- **Attack box:** Intel PC or MacBook running our in-house developed OcuSync reception/injection framework connected to a BladeRF SDR card.
- **Ground station receiver:** Aeroscope mobile unit. In contrast to the DroneScout, the Aeroscope is a standalone unit equipped with a monitor capable of visualizing detected DJI drones on a map.

In the following sections we assume that the attacker is positioned in such a way that the RID injected traffic is received by the ground stations.

The attacker can increase the reception rates of the injected wireless traffic by positioning themselves "close enough" to the ground station receivers, or in a location where they have line-of-sight towards the receivers, or by using high power transmitter or high gain directional antennas.

6.3.1 Attack Scenario 1: Single Drone Forging

Scenario Description

The attacker injects Wi-Fi beacons containing ODID message packs that emulate the presence of a single drone flying in the airspace surrounding the DroneScout ds230 ground station receiver.

The attack takes advantage of the fact that RID (ODID) data is not authenticated, and the ground station receiver has no way to discern forged RID data injected by an attacker from real RID data that is usually transmitted by a drone.

Impact

The DroneScout ds230 parses the injected ODID messages as if they are transmitted by a real drone and transmits the processed data to the MQTT broker.

The final user will see a drone appearing on the map as if a real drone is positioned in the reported location while, in reality, there are no drones flying in the monitored airspace.

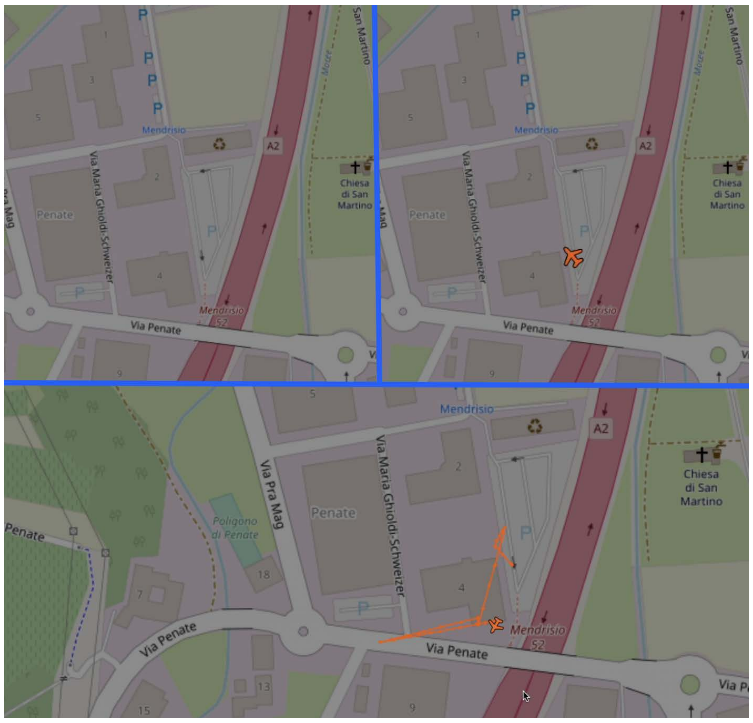
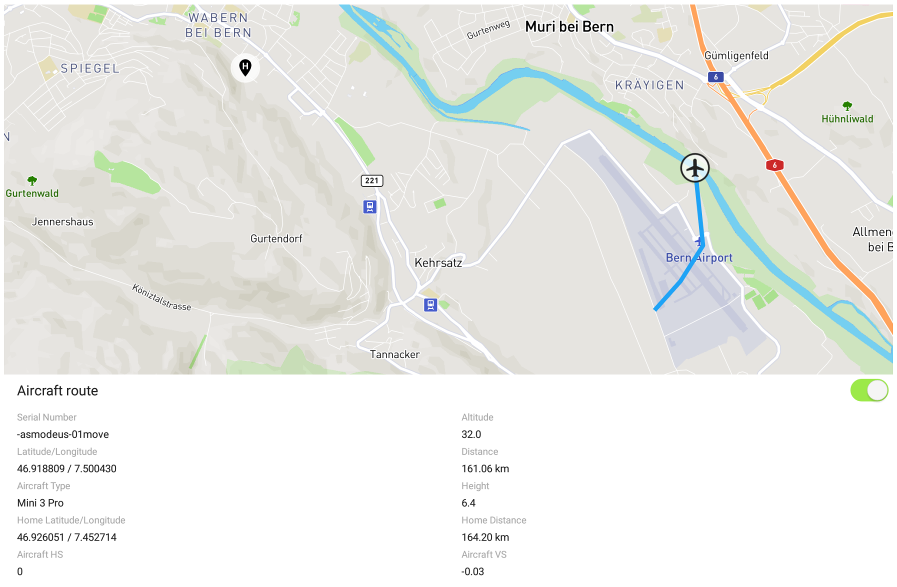


Figure 49 - Attack scenario 1: Forging of a single drone.

Here is a step-by-step description of the attack whose effects are shown in Figure 49:

- At the beginning (top left in the figure) no one is transmitting *RID* data, so the DroneScout does not detect any beacon frames containing ODID messages and does not transmit any data to the MQTT broker. The final user sees a map without any drone.
- The attacker selects an appropriate Wi-Fi channel. The channel selection logic can be based on different metrics such as how much a channel is occupied or detected noise levels. In general, it is better to choose channels on the 2.4GHz band because they allow the Wi-Fi transmitted frames to reach longer distances.
- The attacker generates a random MAC address *M* that will be used as source address for the Wi-Fi frames that are going to be injected. (Remember from Chapter 4 that the DroneScout uses the beacon source MAC address as drone identifier.)
- The attacker starts injecting Wi-Fi beacon frames containing an ODID message pack on the selected channel. The frames use *M* as source address. The interval between each beacon frame can be chosen arbitrarily by the attacker (in our PoC we used 160ms which is the same period used by the DJI Mini 3 Pro). In each subsequent ODID message pack, the attacker modifies latitude, longitude, altitude and speed to make the drone appear to follow a random trajectory.
- The DroneScout captures the first beacons containing ODID message packs. The ODID data is not authenticated, so the DroneScout accepts and parses it. It creates a new entry D_M for drone with MAC address M (see Chapter 4 for details on DroneScout internals) and associates the *RID* data contained in the ODID message to this entry (RID_M). Then it transmits a new JSON message containing RID_M to the MQTT broker.
- The user sees a drone appearing on the map (top right in the figure).
- The DroneScout keeps receiving and accepting the beacon frames containing ODID data injected by the attacker. The DroneScout periodically (every 500ms by default) transmits a JSON message containing the updated RID data for the drone with MAC address *M* (RID_M) to the MQTT broker.
- The user sees the drone following the random trajectory generated by the attacker (bottom image in Figure 49). The user has no way to tell if the drone is real or not just by looking at the map. They can attempt to visually confirm the presence of the drone, however this is not always feasible since drones are small and can easily fly at low altitudes or behind obstacles, making it extremely hard to see them. Even assuming the user can visually confirm the presence of the drone, it is possible to make their life even harder by emulating a vast number of drones. This is the attack scenario explored in the next section.

The same attack scenario can also be replicated in the case of DJI's proprietary RID protocol. In this situation, the attacker uses the SDR card to inject OcuSync signals containing forged telemetry packets. The Aeroscope will accept the injected telemetry data as valid and the forged drone will appear on the map on the Aeroscope's built-in display. Figure 50 shows the effect of this attack scenario against the Aeroscope. In this example, the attacker injected forged DJI telemetry data to make a drone flying over the Bern airport appear on the map.



Aircraft route

Serial Number	Altitude
-asmodeus-01move	32.0
Latitude/Longitude	Distance
46.918809 / 7.500430	161.06 km
Aircraft Type	Height
Mini 3 Pro	6.4
Home Latitude/Longitude	Home Distance
46.926051 / 7.452714	164.20 km
Aircraft HS	Aircraft VS
0	-0.03

Figure 50 - Attack scenario 1 in case of OcuSync: forging a single drone over the Bern airport.

6.3.2 Attack Scenario 2: Multiple Drone Forging

Scenario Description

The attacker injects Wi-Fi beacons containing ODID message packs that emulate the presence of an arbitrary number of drones flying in the airspace surrounding the DroneScout ds230 ground station receiver. Each emulated drone follows an independent random trajectory.

As in the previous case, the attack takes advantage of the fact that RID (ODID) data is not authenticated, and the ground station receiver has no way to discern forged RID data injected by an attacker from real RID data that is usually transmitted by a drone.

Impact

The DroneScout ds230 parses the injected ODID messages as if they are transmitted by real drones and transmits the processed data to the MQTT broker.

The final user will see multiple drones appearing on the map as if real drones are positioned in the reported locations, while in reality there are no drones flying in the monitored airspace. In this case it will be much more difficult, if not impossible, for the user to visually confirm the data reported on the map (e.g., tell if all the reported drones are fake or if there is at least one real drone).

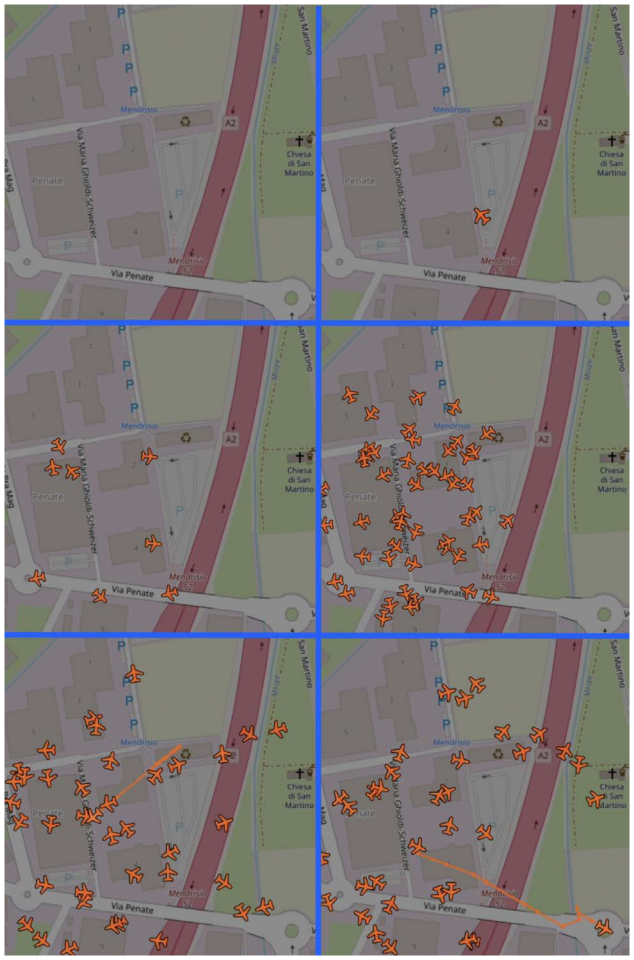


Figure 51 - Attack scenario 2: Forging of multiple drones.

Here is a step-by-step description of the attack whose effects are shown in Figure 51:

- At the beginning (top left in the figure) no one is transmitting RID data and the final user sees a map without any drones (the same as the previous attack scenario).
- The attacker selects an appropriate Wi-Fi channel (see attack scenario 1 for a discussion on channel selection). In this case the attacker could also decide to use multiple Wi-Fi channels and emulate different drones on different Wi-Fi channels. This would make the attack more realistic. However, in our PoC, to keep things simpler, we used a single Wi-Fi channel for all emulated drones.
- For each drone *i* that they want to emulate, the attacker generates a random MAC address M_i. Given that the MAC address is used by the DroneScout to identify a drone (i.e., associate the received RID data to the right drone) it is important that each emulated drone has its own unique MAC address.

- The attacker starts injecting beacon frames containing ODID messages. For each drone *i* the attacker must periodically (e.g., every 160ms) inject a separate beacon frame with source address M_i. Also in this case, for each drone and for each subsequent ODID message associated with that drone, the attacker modifies the latitude, longitude, altitude and speed to make the drone appear to follow a random trajectory.
- The DroneScout captures the first beacons containing ODID message packs. The ODID data is not authenticated, so the DroneScout accepts and parses it. It creates a new entry D_{Mi} for each drone with a MAC address Mi (see Chapter 4 for details on DroneScout internals) and associates the RID data contained in the ODID message to this entry (RID_{Mi}). Then it transmits a new JSON message containing RID_{Mi} to the MQTT broker.
- The user sees a drone appearing on the map (top right in Figure 51).
- The DroneScout keeps receiving and accepting the beacon frames containing ODID data injected by the attacker.
- When a new ODID message is received from a MAC address M_i that has not been seen before, the user will see a new drone appear on the map (middle left and right in Figure 51).
- When a newly received ODID message arrives from a known MAC address M_i the user will see a drone on the map moving. Each drone has its own independent trajectory (bottom left and right in the figure).
- In this case the user has no way to tell if the reported drones are real or not just by looking at the map. The user may be aware of being under attack by visually confirming that there are not so many drones flying around. However, it can be hard to visually confirm whether at least one of the drones reported on the map is real.

The same attack scenario can also be replicated in the case of DJI's proprietary RID protocol. The attacker uses the SDR card to inject OcuSync signals containing forged telemetry packet to fake the presence of multiple drones. The effect of this attack scenario against the Aeroscope is shown in Figure 52. All the forged OcuSync telemetry packets injected by the attacker are considered valid by the Aeroscope and the corresponding drones are visualized on the map.

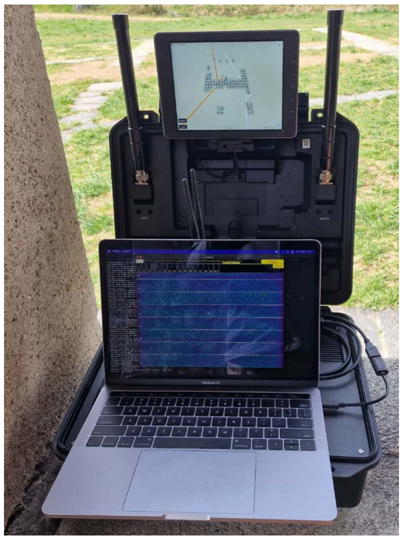


Figure 52 - Attack scenario 2 in case of OcuSync.

6.3. Attack Scenario 3: Drone Cloud

Scenario Description

This scenario assumes there is a real (legitimate) drone flying in the airspace monitored by the DroneScout receiver.

The attacker is interested in creating confusion by making an arbitrary number of fake drones appear to be flying around the real drone and following the same trajectory.

Impact

The DroneScout receives and parses both ODID messages transmitted by the real drone and ODID messages injected by the attacker.

The user sees a cloud of drones surrounding the legitimate drone on the map, making it difficult to determine the actual location of the real drone.

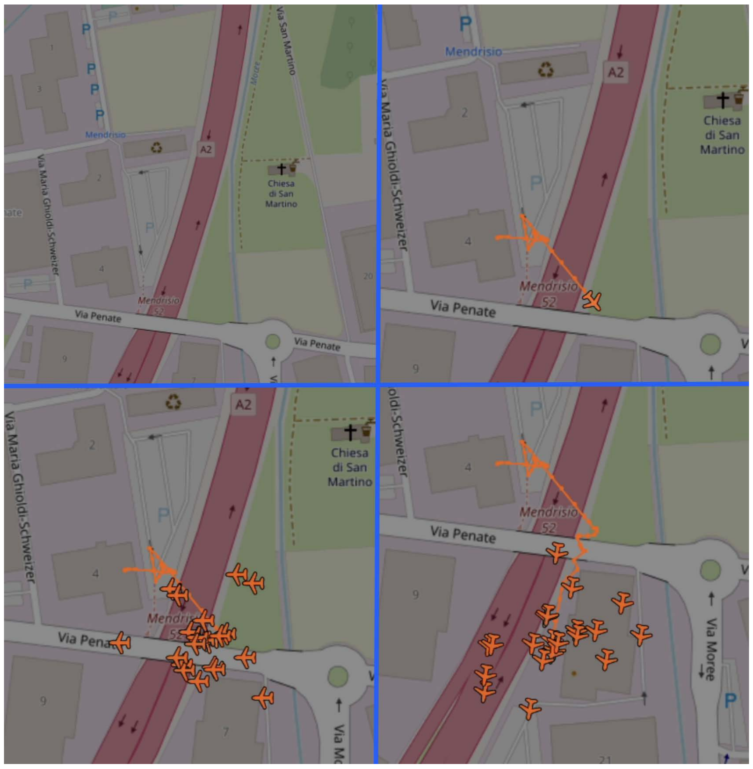


Figure 53 - Attack scenario 3: cloud of drones.

Here is a step-by-step description of the attack whose effects are shown in Figure 53:

- As in the previous scenarios, at the beginning (top left in the figure) no one is transmitting RID data and the final user sees a map without any drone reported.
- Then an operator starts flying their own legitimate drone. The drone periodically transmits ODID messages in Wi-Fi beacon frames containing real RID data. Those messages are captured and parsed by the DroneScout that transmits the corresponding JSON messages to the MQTT broker. The DroneScout's user sees the legitimate drone appearing on the map and following a certain trajectory (top right in Figure 53).
- The attacker, with Wi-Fi monitoring capabilities, receives the ODID messages transmitted by the real drone. Then for each drone *i* that they want to emulate (the number of drones to inject for creating the cloud is configurable), they generate a random MAC address M_i.
- For each drone they want to emulate, the attacker injects a beacon frame with source address M_i containing an ODID message whose latitude, longitude and latitude are near (where near is configurable, e.g., 20 meters) to the last location reported in the last captured ODID message coming from the real drone.
- The user sees a cloud of drones surrounding the legitimate drone (bottom left in Figure 53).
- For each newly captured ODID message coming from the real drone, the attacker injects an ODID message for each address M_i, updating the location information to maintain the emulated drones near the real drone.
- The user sees the cloud of drones following the same trajectory as the real drone (bottom right in Figure 53), making it difficult to identify the exact location of the real drone.

The same attack scenario can also be replicated in the case of DJI's proprietary RID protocol. In this situation, the attacker uses the SDR card to inject OcuSync signals containing forged telemetry packets. The Aeroscope will accept the injected telemetry data as valid and the forged drone will appear on the map on the Aeroscope's built-in display. Figure 54 shows the effect of this attack scenario against the Aeroscope. In this example, the attacker injected forged DJI telemetry data to make a drone flying over the Bern airport appear on the map.

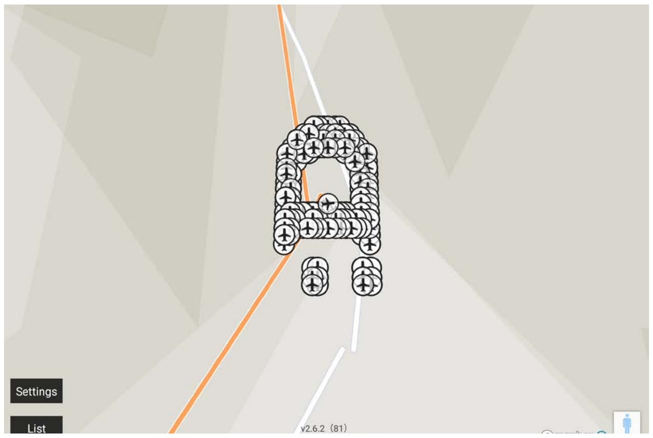


Figure 54 - Attack scenario 3 in the case of OcuSync.

6.3.4 Attack Scenario 4: DroneScout Timer (CVE-2023-29156)

Scenario Description

This scenario assumes there is a real drone flying in the airspace monitored by the DroneScout receiver. The attacker injects, at precise times, spoofed ODID messages, using the MAC address of the real drone as source address. This injection forces the DroneScout to drop RID data transmitted by the real drone and only report the false, injected RID data to the MQTT broker.

Note: This proof-of-concept attack exploits the DroneScout ds230's vulnerability identified by CVE-29156. We refer the reader to Chapter 4 for a technical description of how the vulnerability works and how it can be exploited.

Impact

The user sees a drone on the map, which is a legitimate drone flying in monitored airspace, however the trajectory followed by the real drone is different than the one visualized on the map.

The attacker can use this scenario to achieve the false prosecution of an innocent operator, for example, by making a legitimate drone appear to be in a no-fly zone.

Analogously, an attacker could make a drone in a no-fly zone appear to be outside of the no-fly zone.

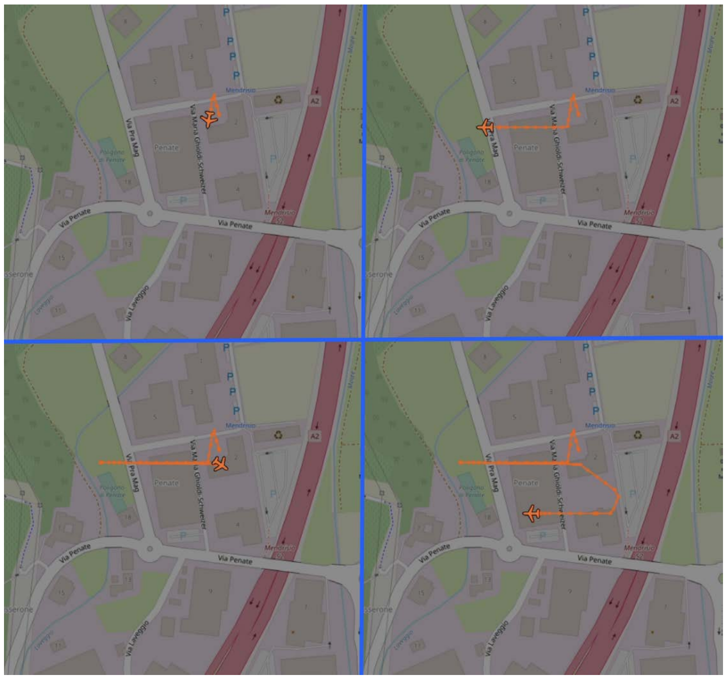


Figure 55 - Attack scenario 4: DroneScout timer.

Here is a step-by-step description of the attack whose effects are shown in Figure 55:

- An operator starts flying their own legitimate drone *D* with MAC address M_D. The drone periodically transmits ODID messages in Wi-fi beacon frames containing real RID data RID_M. Those messages are captured and parsed by the DroneScout that transmits the corresponding JSON messages to the MQTT broker. The DroneScout's user sees the legitimate drone appearing on the map and following a certain trajectory (top left in Figure 55).
- The attacker, with Wi-Fi monitoring capabilities, receives the ODID messages transmitted by the real drone. The attacker creates ODID messages with spoofed source MAC address M_D containing crafted RID data RID_{D1}. In our PoC, the crafted RID data RID_{D1} is generated in such a way that the drone appears to be moving west. The attacker then injects these messages in such a way that each is injected 500ms or more after RID_M has been transmitted to the MQTT broker and before *D* transmits its next ODID message containing the real RIM_D (see Chapter 4 to understand why this is done in this way).
- The user sees the real drone pointing west on the map (top right in Figure 55).
- When the attacker stops injecting the ODID message, the user sees the actual position of the real drone again (bottom left in the figure). In real attack scenarios the attacker does not have to stop the injection. In the PoC this was done to demonstrate that the real drone was not actually flying west.
- When the attacker starts injecting the ODID messages again, the user sees the real drone pointing west again (bottom right in the figure).

As explained in the previous chapter, for this attack to work the attacker must inject the ODID messages with crafted RID data with precise timing (500ms or more after RID_M has been transmitted to the MQTT broker and before *D* transmits its next ODID). The problem here is that the attacker does not know the internal state of DroneScout timestamps and timers. This means that the attacker does not know the right times to inject the spoofed ODID messages, so they can only try to predict the right times. This makes the attack probabilistic.

In order to increase the chance of success, the attacker can increase the number of injected ODID messages and inject them on multiple Wi-Fi channels at once. Preliminary non-optimized experiments performed with a single channel injection strategy showed that it is quite easy to achieve an attack success rate above 90%. However, when the predicted injection time of the attacker is wrong, the actual location of the drone will appear on the map for a few instants. This example is shown in Figure 56, where the attacker was injecting ODID messages to make the drone appear like it was going west but at a certain point the attack failed and the location of the drone briefly jumped back to its real position.

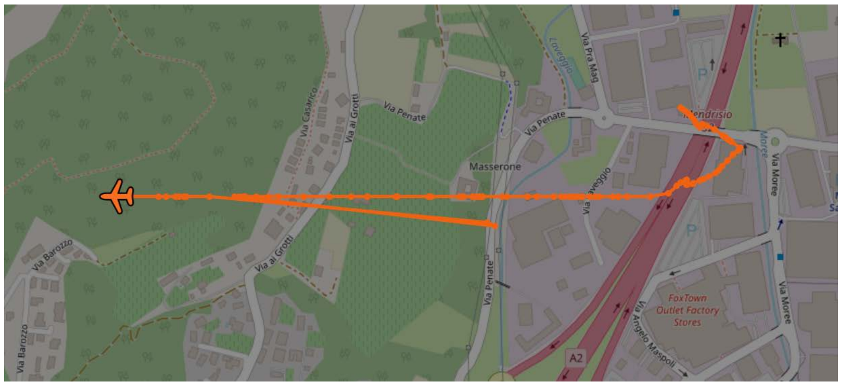


Figure 56 - Attack scenario 4: DroneScout timer attack is not 100% reliable.

6.3.5 Attack Scenario 5: DroneScout Adjacent Channels (CVE-2023-31191)

Scenario Description

This scenario assumes there is a real drone flying in the airspace monitored by the DroneScout receiver. The attacker injects, with precise timing, spoofed ODID messages, using the MAC address of the real drone as source address. This forces the DroneScout to drop RID data transmitted by the real drone and only report the RID data injected by the attacker to the MQTT broker. This attack scenario and impact are identical to the previous attack scenario, however, in this case the attack is 100% reliable if a high-power transmitter or high gain directional antennas are used.

Note: This proof-of-concept attack exploits the DroneScout ds230 vulnerability identified by CVE-31191. We refer the reader to Chapter 4 for a technical description of how the vulnerability works and how it can be exploited.

Impact

The user sees a drone on the map, which is a legitimate drone flying in monitored airspace, but the trajectory it is following in reality is different than what is visualized on the map.

The attacker can use this scenario to achieve the false prosecution of an innocent operator, for example, by making a legitimate drone appear to be in a no-fly zone.

Or, similarly, the attack could be used to make a drone in a no-fly zone appear to be outside the no-fly zone.

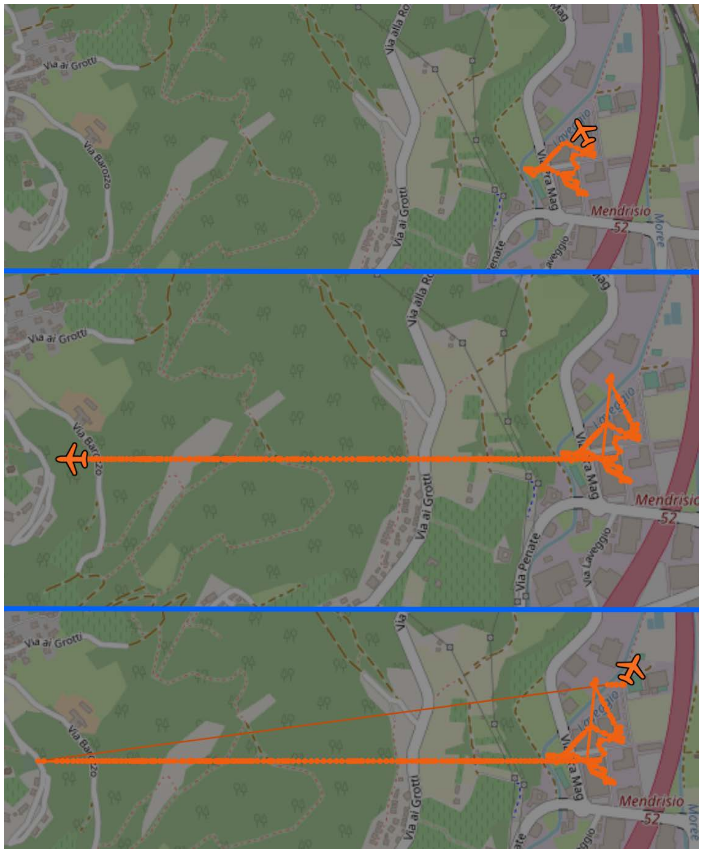


Figure 57 - Attack Scenario 5: DroneScout adjacent channels.

Here is a step-by-step description of the attack whose effects are shown in Figure 57:

- An operator starts flying their own real and legitimate drone *D* with MAC address M_D. The drone periodically transmits ODID messages in Wi-Fi beacon frames containing real RID data RID_M. Those messages are captured and parsed by the DroneScout, which transmits the corresponding JSON messages to the MQTT broker. The DroneScout's user sees the legitimate drone appearing on the map and following a certain trajectory (top of Figure 57). The beacon frames containing ODID data are received by the DroneScout with power M_D->RSSI.
- The attacker creates ODID messages with spoofed source MAC address M_D and containing crafted RID data RID_{D1}. In our PoC, the crafted RID data RID_{D1} is generated in such a way that the drone appears moving west. The attacker then injects the messages with spoofed source on an adjacent channel *Ca* (e.g., if the drone *D* is transmitting on channel 6, the attacker can transmit on channel 8). Also, the beacon frames containing RID_{D1} must be transmitted with high enough power that they are received by the DroneScout with an RSSI *Ra* that satisfies the condition ((*Ra* - M_D->RSSI) > 6) (see Chapter 4 to understand why this is done in this way).
- The user sees the real drone pointing west on the map (middle of Figure 57).
- When the attacker stops, the real location of the drone is revealed (bottom of the figure).

7. Conclusion

Unfortunately, policy requirements making drone RID systems “immediately actionable” led to the deployment of RID protocols that do not protect telemetry data confidentiality nor guarantee telemetry data integrity and authentication. This makes current RID systems open to injection attacks. Moreover, vulnerabilities in ground station receivers, like the ones we found affecting the DroneScout ds230, open the possibility of more sophisticated types of attacks, making the situation even worse.

In this research we studied how RID ground stations in charge of receiving drone telemetry data can be abused by a malicious user. By leveraging both RID system intrinsic weaknesses and the vulnerabilities we found on ground station receivers, we identified attack scenarios where an attacker could emulate the presence of fake drones, spoof legitimate telemetry data, inject fake drone trajectories and, ultimately, disrupt RID functionalities.

When critical infrastructure facilities are involved (airports, military bases, industrial areas, etc.), these attacks become particularly serious because they can lead to the disruption of critical operations. We believe that this work is important because it highlights the type of risks that a final user, like a law enforcement agency or a critical infrastructure provider, could face if they decided to rely exclusively on RID systems to monitor for the presence of drones in the surrounding airspace, or to make security and safety related decisions.

Drone RID regulations and standards, which require drones to periodically broadcast their telemetry information, will play an essential role in the future of aviation. This is true both in terms of airspace security and safety as drones allow entities such as law enforcement and critical infrastructure authorities to be aware of the drones flying surrounding a certain area. RID systems, as currently designed, should not be used as the one and only source of information for taking security and safety sensitive decisions.

Cybersecurity for OT, IoT and Critical Infrastructure

Nozomi Networks protects the world's critical infrastructure from cyber threats. Our platform uniquely combines network and endpoint visibility, threat detection, and AI-powered analysis for faster, more effective incident response. Customers rely on us to minimize risk and complexity while maximizing operational resilience.

nozominetworks.com

NN-WP-DRONES-8.5x11-001