NOZOMI NETWORKS LABS

WHITE PAPER

UWB Real Time Locating Systems: How Secure Radio Communications May Fail in Practice

AUTHORS

Andrea Palanca

Luca Cremona

Roya Gordon

About Nozomi Networks Labs

Nozomi Networks Labs is dedicated to reducing cyber risk for the world's industrial and critical infrastructure organizations. Through its cybersecurity research and collaboration with industry and institutions, it helps defend the operational systems that support everyday life.

The Labs team conducts investigations into industrial device vulnerabilities and, through a responsible disclosure process, contributes to the publication of advisories by recognized authorities.

To help the security community with current threats, they publish timely blogs, research papers and free tools.

The **Threat Intelligence** and **Asset Intelligence** services of Nozomi Networks are supplied by ongoing data generated and curated by the Labs team.

To find out more, and subscribe to updates, visit **nozominetworks/labs**

Table of Contents

1. Introduction

As the world becomes more connected, companies have found that wireless technology can be leveraged to increase efficiency and overall productivity while reducing unnecessary costs associated with cabling infrastructure. Wireless systems also allow for quicker sharing of information than wired networks by reducing wait times on data transfers between different devices within an organization. While these benefits are apparent, wireless communication systems are susceptible to various security threats that can compromise their reliability and impact production operations.

In an effort to strengthen the security of devices utilizing Ultra-wideband (UWB) radio waves, Nozomi Networks Labs conducted a security assessment of two popular UWB Real Time Locating Systems (RTLS) available on the market. In our research, we discovered zero-day vulnerabilities and other weaknesses that, if exploited, could allow an attacker to gain full access to all sensitive location data exchanged over-the-air.

In this white paper, we demonstrate how an attacker may exploit RTLS to locate and target people and objects, hinder safety geofencing rules, and interfere with contact tracing. We also present key actions that companies can take to help mitigate these risks and implement a secure wireless network infrastructure.

1.1 Ultra-wideband (UWB) and Real Time Locating Systems (RTLS)

UWB is a wireless communication protocol that uses radio waves to determine precision and ensure communication of peer devices. It is ideal for short-range devices because it has a relatively small wavelength, meaning it can transmit information quickly over short distances.[1]

UWB is used in many different types of applications ranging from consumer electronics to medical devices to industrial automation. Many companies are now using UWB technology in their products to take advantage of its unique properties, including its ability to send data through solid objects, like walls and other barriers, without losing quality or slowing down transmission speeds. This is opposed to other radio frequencies (RFs), such as Bluetooth or Wi-Fi, which use narrow-band radio waves for more line-of-sight precision over longer distances.

[1] **"What UWB Does,"** FiRa Consortium.

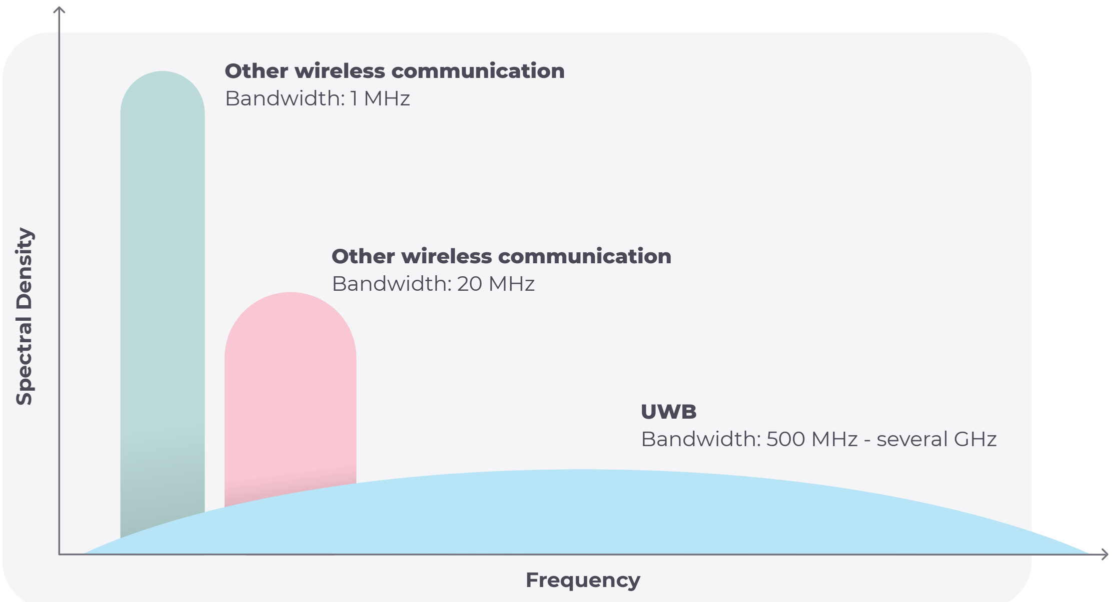


Figure 1 - Spectral density for UWB and narrowband. (Source: FiRa Consortium)

UWB is the preferred communication protocol for RTLS, which is a technology that uses radio-frequency signals to locate both stationary and mobile objects. RTLS consists of three components: tags that are attached to assets, anchors that receive the wireless signal, and a computer system that processes and stores tag positions. When an asset passes through an area with a tag attached to it, the tag sends out a signal which is received by computers connected to the system. The computers analyzes the signal's time of arrival to determine its distance from the asset, and then information is stored into the database.

UWB utilizes the following positioning techniques:

1. **Two Way Ranging (TWR):** This method calculates the Time of Flight (TOF) of an electromagnetic wave by measuring the time it takes for a wave to travel from one point to another. This method is mostly used for hands-free access control or locating lost items.[2]
2. **Time Difference of Arrival (TDoA):** This method uses multiple anchors deployed within a given facility. When the anchors receive a beacon from a tagged device, the timestamp of the beacon will be analyzed to correlate the position of the device.[3] This method is mostly used when tracking personnel in facilities.

This research will focus on the latter technique, TDoA.

[2] **"Why UWB Is the Premier Location Technology,"** Qorvo.
[3] Ibid.

1.2 Use Cases

UWB RTLS are used in manifold use cases: from smart building and mobility to industrial, from smart retail to smart home and consumer. Two examples of use cases follow.

In production environments, RTLS uses radio frequency technology to locate components in various stages of production from the time they are created until they are delivered to the customer. The system allows for a precise positioning of each component in its own unique location, ensuring that the component does not get mixed up with others or placed incorrectly during assembly. The system also allows for automatic release of components when they reach their designated areas so that they do not have to be handled manually by workers; this reduces the possibility of errors during final assembly.[4]

RTLS is also used in access control systems, which have traditionally been cumbersome and inconvenient. They required the user to either wave their credential in front of a sensor or insert it into a lock, which can be difficult to do if a person is carrying items or wants to just walk through a door without stopping. UWB RTLS allows the lock and unlock functions to happen in response to movements and positioning, making accessing buildings and vehicles hands-free and hassle-free.[5]

1.3 Cyber Threats to Wireless Communications

While there are many benefits to these technologies, when it comes to industrial environments, there is no shortage of potential security risks. With the growing use of wireless networks in the industrial space comes an increased likelihood that those networks will be vulnerable to attacks from cyber criminals who are seeking to exploit vulnerabilities in order to gain access to sensitive data or disrupt operations.

[4] **"Efficient, reliable, paperless: Full transparency in the automative assembly,"** Siemens, 2019.
[5] **"UWB Use Cases,"** FiRa Consortium.

1.4 Motivation

According to the Fine Ranging, or FiRa, consortium, there was an increased demand in 2018 for "improvements to existing modulations to increase the integrity and accuracy of ranging measurements."[6] In 2020, the Institute of Electrical and Electronic Engineers (IEEE) released standard 802.15.4 which provides guidance (protocols, specifications, etc.) for low-rate wireless network communications, replacing the outdated 2015 version. IEEE quickly followed up with the 802.15.4z amendment in 2020, which adds requirement to achieve security in wireless transmissions.

The new physical layer (PHY), was added to the 802.15.4z specification to make it harder for attackers to access or manipulate UWB communications. The extra portion of the PHY acts as a kind of shield between the network and any external devices trying to access it.[7] The addition of cryptography and random number generation was to ensure that no one can eavesdrop on or manipulate UWB communications.[8]

While these updates are an important step towards securing UWB, upon further review, we noticed that the synchronization and exchange of location data are considered out-of-scope by the standard, despite being critical aspects in RTLS. These communications, whose design is left entirely to vendors, are critical aspects for the overall posture of TDoA RTLS.

To the best of our knowledge, research on UWB RTLS focusing on the security of communications via Ethernet, Wi-Fi, or other media for the synchronization and exchange of location data has never been done in literature or appeared in a security conference.

For this reason, we decided to focus our research solely on these specific communications, to evaluate their security posture in an effort to strengthen the overall security of UWB RTLS.

[6] **"What UWB Does,"** FiRa Consortium.
[7] **"UWB Technology Comparison,"** FiRa Consortium.
[8] Ibid.

2. Methodology and Attack Demos

In this chapter, we illustrate the entire methodology followed during our research, and the results obtained. We describe the scope of our investigation, illustrate the basic concepts behind the TDoA theory, explain all reverse engineering steps done during our analysis, show how an adversary can retrieve or estimate all information required for an attack, and demonstrate how they can abuse this knowledge to perform practical attacks against real-world scenarios.

2.1 Scope

UWB RTLS are pervasive technologies that can be deployed in a plethora of conditions and for a wide variety of use cases. Additionally, they comprise manifold components and protocols. This chapter of the document defines both the industry scope and technical scope of our research.

2.1.1 Industry Scope

From parking structures to hospitals, from airports to factories, from retail to sports fields, UWB RTLS enable sophisticated localization-based services in the most disparate environments.

Given the breadth of industries that utilize UWB, we decided to limit the scope of our research to those that were both highly targeted and highly critical. These are expected to be the industries where a security flaw is most likely be exploited by adversaries, and lead to the highest impacts.

Among the various industries utilizing UWB RTLS, we focused our research on both the industrial and healthcare sectors. We decided to focus on these sectors primarily because the industrial and healthcare sectors have seen a surge in cyberattacks in recent years,[9] and UWB RTLS are employed for safety-related purposes,[10,11] (Figure 2), such as:

- **Employee and patient tracking:** In factories, UWB RTLS help the facility's management system track and rescue any employees remaining onsite in the event of an emergency. In hospitals, they are used to track a patient's position and quickly provide medical assistance in case of sudden, serious medical symptoms;
- **Geofencing:** In both factories and hospitals, UWB RTLS enforce safety-geofencing rules. For instance, UWB RTLS can be configured to halt hazardous machinery in case a human is within close proximity, to prevent harmful consequences;
- **Contact tracing:** UWB RTLS enables centralized contact tracing during major pandemics like COVID-19. By monitoring and tracking contact between people, it can determine who came in contact with someone who tested positive for COVID-19, so that necessary quarantine measures can be taken.

[9] **"New OT/IoT Security Report: Trends and Countermeasures for Critical Infrastructure Attacks,"** Nozomi Networks Labs, February 2, 2022.
[10] **"Worker Safety,"** Ubisense.
[11] **"Simatic RTLS: How to create a safe working environment,"** Markus Weinlaender, Siemens Ingenuity, July 13, 2020.

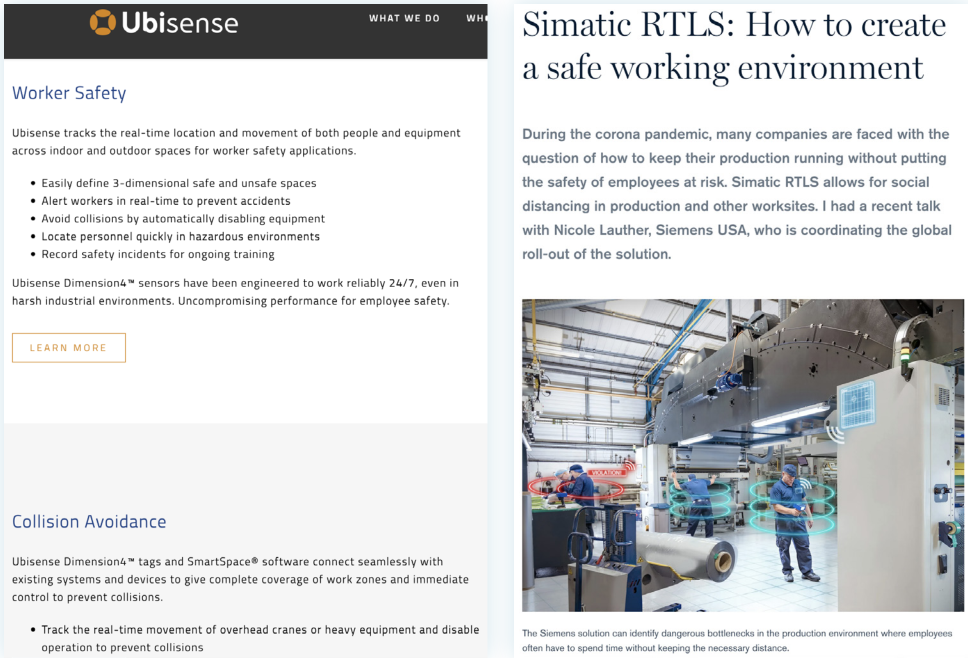


Figure 2 - Examples of safety-related use cases advertised by vendors for UWB RTLS.

It is thus paramount that the security of industrial and healthcare UWB RTLS is as robust as possible, to prevent adversaries from taking advantage of systems that cause safety-related consequences to victims.

Having defined the industry scope, we performed an analysis of the RTLS targeting the industrial and healthcare sectors available on the market, that took into consideration aspects such as product features, availability time, or cost of purchase. Ultimately, we identified and purchased the following RTLS solutions:

- Sewio Indoor Tracking RTLS UWB Wi-Fi Ki[12] (Figure 3)
- Avalue Renity Artemis Enterprise Kit[13] (Figure 4)

Figure 3 - Sewio Indoor Tracking RTLS UWB Wi-Fi Kit.

Figure 4 - Avalue Renity Artemis Enterprise Kit

Both of these UWB RTLS kits come equipped with a set of tags, anchors, and a server software that can be accessed to view the location of tags, enable functionalities such as the safety features described above, perform maintenance operations, etc.

[12] **"Indoor Tracking RTLS UWB Wi-Fi Kit,"** Sewio,
[13] **"Artemis Enterprise Kit,"** Avalue Renity.

2.1.2 Technical Scope

Figure 5 portrays the architecture of a generic UWB RTLS, outlining the components involved as well as the protocols that are used in its communications.

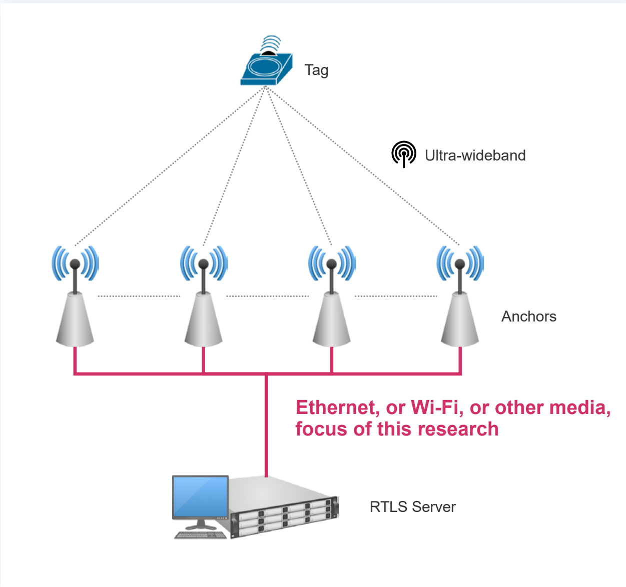


Figure 5 - Architecture of a generic UWB RTLS.

In an average RTLS infrastructure, a tag communicates with a set of anchors deployed in strategic positions of a room by means of UWB signals. These anchors then not only communicate with each other via UWB, but also interact with the RTLS server via common network media, such as Ethernet or Wi-Fi.

The purpose of each of these communications is different, and is summarized below:

- A tag sends UWB signals to the anchors, which receive them and keep track of the arrival times of each UWB message. This information will be used later by the RTLS Server to compute the position of the tag.
- One reference anchor sends UWB signals to the other anchors, which receive them and keep track of the arrival times of each UWB message. This information is then used by the RTLS Server to perform the synchronization of the anchors.
- Finally, the anchors send all arrival times of the transmitted and received UWB messages to an RTLS Server via Ethernet, Wi-Fi, or other media. The RTLS Server uses all data to complete the anchor synchronization process and reconstruct the position of the tag.

Given the architecture illustrated above, to obtain an overall secure positioning system, it is crucial that both the UWB signals and the communications via Ethernet, Wi-Fi, or other media are secured. A flaw in any of these communication steps may compromise the security of the entire infrastructure.

Up to now, security research has exclusively focused on the analysis of UWB signals, leading to the publication of multiple security studies that appeared in numerous conferences, such as ACM WiSec 2021 Architecture of a generic UWB RTLS[14], NDSS 2019[15], or Usenix 2019.[16]

[14] **"Security Analysis of IEEE 802.15.4z/HRP UWB Time-of-Flight Distance Measurement,"** Mridula Singh, Marc Roeschlin, Ezzat Zalzala, Patrick Leu, and Srdjan Čapkun, in Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '21), 2021.

[15] **"UWB with Pulse Reordering: Securing Ranging against Relay and Physical-Layer Attacks,"** Mridula Singh, Patrick Leu, and Srdjan Čapkun, in Proceedings of Network and Distributed Systems Security (NDSS) Symposium 2019, 2019.

[16] **"UWB-ED: Distance Enlargement Attack Detection in Ultra-Wideband,"** Mridula Singh, Patrick Leu, AbdelRahman Abdou, and Srdjan Čapkun, in Proceedings of the 28th USENIX Security Symposium, 2019.

2.2 TDoA Background and Theory

Literature presents many algorithms that leverage TDoA to locate assets in any kind of environment.[17,18,19] To better clarify the reversing procedure adopted for this work and understand the preconditions necessary for an attack, the fundamentals behind TDoA are worth a brief analysis.

2.2.1 Packet Taxonomy

In a TDoA RTLS, there are normally two kinds of packets that are exchanged between the anchors and the server:

1. Synchronization packets, also known as "sync" packets, or "CCP" packets;
2. Positioning packets, also known as "blink" packets, or "TDoA" packets.

Synchronization packets are used for anchor synchronization purposes. Periodically, a reference anchor (sometimes called "master" in off-the-shelf RTLS) transmits an UWB signal that is received by all other non-reference anchors (sometimes called "slaves" in off-the-shelf RTLS). The reference anchor sends a synchronization packet on the network containing the instant at which it has sent the UWB signal, and the non-reference anchors a synchronization packet containing the instant at which they received it. It is important note that the anchors' clocks are usually not in sync with each other (e.g., at the same exact time, anchor 1 might have its clock at 8.4322348 s, anchor 2 at 2.4524391 s, anchor 3 at 15.1147349 s, etc.), due to different boot times, clock drifts, or other reasons.

This synchronization schema is a form of wireless synchronization, because it involves the transmission of a wireless UWB signal. Alternatively, some RTLS may replace the transmission of UWB signals with a wired clock signal generated by a single clock source and distributed to all anchors. This solution, however, requires additional wiring and appliances and, as such, is less common in off-the-shelf solutions.

Positioning packets are used for tag localization purposes. A tag emits an UWB signal, which is received by all anchors. All anchors send the instant at which they received the UWB signal from the tag inside positioning packets to the central positioning server. This information, together with the synchronization packets, is used to compute the tag position. Again, these instants generally differ greatly, not only because they depend on the distance travelled by the UWB signal from the tag to reach the anchor, but also on the current anchor's clock that is not in sync with that of the other anchors (e.g., for the same UWB signal emitted by a tag, anchor 1 might report it received at 8.6215658 s, anchor 2 at 2.6490112 s, anchor 3 at 15.3001173 s, etc.).

[17] **"New three-dimensional positioning algorithm through integrating TDOA and Newton's method,"** Junsuo Qu, Haonan Shi, Ning Qiao, Chen Wu, Chang Su, and Abolfazl Razi, in J Wireless Com Network, 2020.
[18] **"Time Difference of Arrival (TDoA) Localization Combining Weighted Least Squares and Firefly Algorithm,"** Peng Wu, Shaojing Su, Zhen Zuo, Xiaojun Guo, Bei Sun, and Xudong Wen, in Sensors, 2019.
[19] **"An Efficient TDOA-Based Localization Algorithm Without Synchronization Between Base Stations,"** Sangdeok Kim, and Jong-Wha Chong, in Location-Related Challenges and Strategies in Wireless Sensor Networks, 2015.

2.2.2 Algorithm Details

The routine implemented in UWB RTLS can usually be organized in two different steps:

1. clock synchronization;
2. position estimation.

Clock Synchronization: As mentioned before, each anchor has a different time domain. To compare the received timestamps from different anchors, the server needs a clock model able to translate the *local* anchor timestamp domain to a *global* timestamp domain. To do this, the reference anchor periodically sends a synchronization UWB signal, which is received by the other anchors. As the anchors receive this signal, they send a packet to the server indicating the timestamp when the signal was received. At this point, the server is able to compute the clock offsets for each anchor *i* at each algorithm iteration instant *t*, based on the reference anchor.

There are many wireless synchronization algorithms that have been proposed in literature. In this white paper, we describe the *Linear Interpolation* algorithm, a simple yet effective way to achieve wireless synchronization among anchors with different time domains. This is also the same algorithm that we applied later while posing as an attacker, listening to the packets exchanged on the wire and trying to reconstruct the position of the tags.

In this algorithm, to achieve synchronization, a new parameter called *Clock Skew (CS)* is computed for each anchor.

$$refAnchorSyncPeriod(t) = sTs(reference, t) - sTs(reference, t\text{-}1$$

Eq. 1

$$nonRefAnchorSyncPeriod(i, t) = sTs(i, t) - sTs(i, t\text{-}1)$$

Eq. 2

$$CS(i, t) = refAnchorSyncPeriod(t)/nonRefAnchorSyncPeriod(i, t)$$

Eq. 3

The computation of the CS derives from the synchronization packets transmission period: for the reference anchor, the parameter `refAnchorSyncPeriod` is computed by subtracting the timestamp of the last-but-one synchronization packet sTs (reference, t-1) to the timestamp of the last synchronization packet sTs(reference, t) sent by the reference anchor (Eq. 1). The same procedure is adopted to compute the `nonRefAnchorSyncPeriod` for each non-reference anchor (Eq. 2).

For each anchor, the *Clock Skew* is computed as the ratio between the `refAnchorSyncPeriod` and its `nonRefAnchorSyncPeriod` (Eq. 3). It is important to notice that the *Clock Skew* for the reference anchor is equal to 1, as it is the reference for all the other anchors.

Finally, to determine the location of a tag *j*, the server needs the positioning timestamps for, at least, *N+1* anchors, where *N* indicates the number of dimensions *(X, Y, Z)* of the tag that the system wants to compute.

To this extent, the concept of *Global Time (GT)* is introduced: the GT represents the conversion of the positioning timestamp of an anchor to a common clock domain, so that these new timestamps can be compared and used together to estimate the tag position.

Given an anchor *i*, a tag *j*, and an iteration instant *t*, the equation follows.

$$GT(i, j, t) = CS(i, t) * (pTs(i, j, t) - sTS(i, t)) + ToF(i)$$

Eq. 4

Eq 4 formally describes what has been mentioned before: *sTs(i, t)* indicates the timestamp of the synchronization packet during the iteration instant *t* sent by anchor *i*, *pTs(i, j, t)* represents the positioning packet sent by tag *j* to anchor *i* during the iteration instant *t*, while *ToF(i)* represents the time of flight from each anchor to the reference, i.e. the time that it takes for a signal to be transmitted and received among the reference anchor and the non-reference ones. Please note that the *GT(reference, j, t)* is simply *pTs(reference, j, t) - sTS(reference, t)*.

Position Estimation: the so obtained GTs can be directly compared to find the difference of the distances between each anchor i and the tag j at a certain iteration instant t:

$$Delta(i, j, t) = (GT(reference, j, t) - GT(i, j, t)) * c$$

Eq. 5

Where *Delta(i, j, t)* is the difference of the respective distances between the tag *j* and the reference anchor at instant *t*, and the tag *j* and the non-reference anchor *i* at instant *t*, while *c* is the speed of light constant. In fact:

$$Delta(i, j, t) = (GT(reference, j, t) - GT(i, j, t)) * c = GT(reference, j, t) * c - GT(i, j, t) * c = d(reference, j, t) - d(i, j, t)$$

Eq. 6

Where *d(reference, j, t)* is the distance between the tag *j* and the reference anchor at instant *t*, and *d(i, j, t)* the distance between the tag *j* and the non-reference anchor *i* at instant *t*.

Once the server computes the distance differences between the tag and each anchor, the last missing step is the computation of the spatial coordinates. This is simply done by using the formula of the distance between two points:

$$d(i, j, t) = sqrt((Xj,t – Xi)^2 + (Yj,t – Yi)^2 + (Zj,t – Zi)^2)$$

Eq. 7

Where *Xj,t* is the *X* coordinate of tag *j* at instant *t*, *Xi* is the *X* coordinate of anchor *i* (constant across time), and *Yj,t, Yi, Zj,t, Zi* are the analogous versions for the Y and Z coordinates.

Finally, by considering Eq. 5, 6, and 7, a non-linear system of equations can be set up to solve for *Xj,t*, *Yj,t*, and *Zj,t*, which is the position of tag *j* at the instant *t*.

$$Delta(1, j, t) = sqrt((Xj,t – Xreference)^2 + (Yj,t – Yreference)^2 + (Zj,t – Zreference)^2) - sqrt((Xj,t – X1)^2 + (Yj,t – Y1)^2 + (Zj,t – Z1)^2)$$

$$Delta(2, j, t) = sqrt((Xj,t – Xreference)^2 + (Yj,t – Yreference)^2 + (Zj,t – Zreference)^2) - sqrt((Xj,t – X2)^2 + (Yj,t – Y2)^2 + (Zj,t – Z2)^2)$$

$$...$$

$$Delta(N, j, t) = sqrt((Xj,t – Xreference)^2 + (Yj,t – Yreference)^2 + (Zj,t – Zreference)^2) - sqrt((Xj,t – XN)^2 + (Yj,t – YN)^2 + (Zj,t – ZN)^2)$$

Eq. 8

By looking at this system, the reader may now understand the requirement of *N+1* anchors, where *N* indicates the number of dimensions *(X, Y, Z)* of the tag that the system wants to compute.

This is a quadratic N-equations-three-unknowns system, that, if solved, leads to the computation of *Xj,t, Yj,t,* and *Zj,t*. For three coordinates, at least three equations are needed, thus 4 anchors. If only two coordinates are necessary, at least two equations are needed, thus 3 anchors.

If more anchors than coordinates are available, it is possible to use the additional available information to increase the precision of the computed tag position, which may be influenced by external factors such as temporary noise, interferences, etc.

From the equations above, it is also possible to conclude that, to obtain the position of a tag, the following data need to be known:

- All coordinates of the anchors involved
- Synchronization timestamps
- Positioning timestamps

2.3 Reverse Engineering of Device Network Traffic

In order to identify the TDoA routines executed by both Sewio and Avalue UWB RTLS, understand how the network traffic is processed by the two solutions, and assess the security of the network communications, a reverse engineering activity was done. The following two sections describe this process for both solutions.

2.3.1 Sewio RTLS

The Sewio RTLS can be configured to employ either Ethernet or Wi-Fi as a backhaul for the communications among the anchors and server. Multiple Wireshark captures in a variety of situations have been performed, to collect as many packet samples as possible. Some of these samples are reported in Figure 6 and Figure 7.

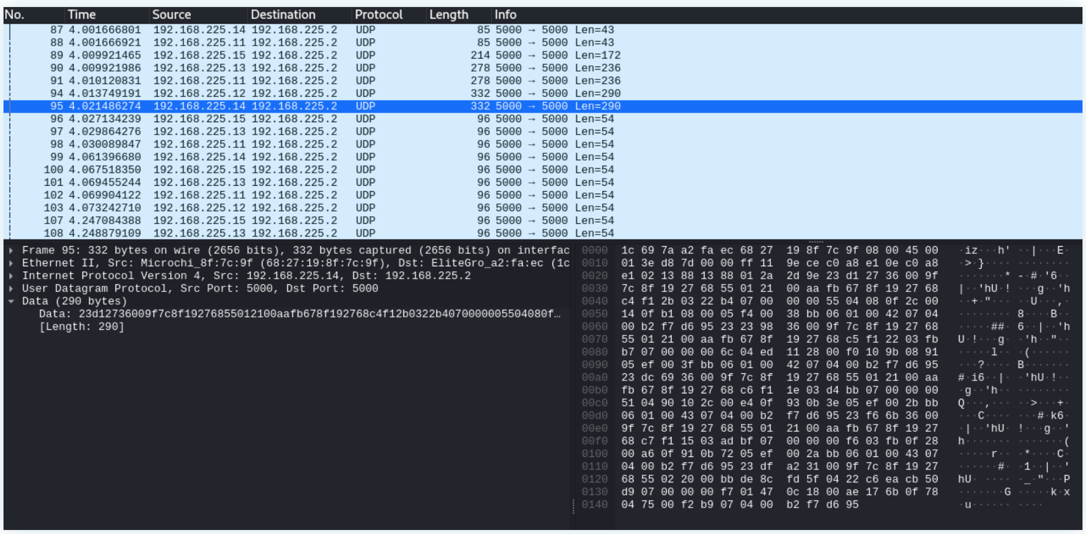


Figure 6 - Sewio RTLS network packet sample.

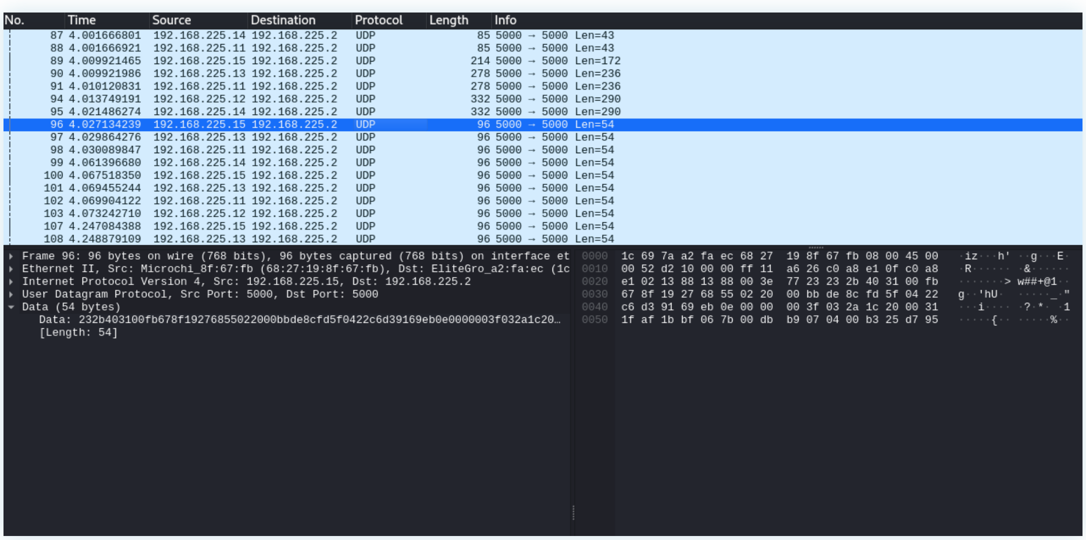


Figure 7 - Sewio RTLS network packet sample (2).

The first step of the reverse engineering process consists of analyzing the traffic generated by the Sewio anchors, to understand which protocols and which ports they use to transmit the information to the server. As can be seen, the Sewio RTLS uses a custom, unknown binary network protocol for the communications among anchors and server. No standard data structures are immediately recognizable. Consequently, an analysis of the server software is required, in order to understand how packets are processed and complete their dissection.

In reference to the previous figures, Sewio anchors (IPs: 192.168.225.{11,12,13,14,15}) communicate with the server over UDP on port 5000. By looking at the output of netstat (Figure 8), the traffic is processed by a NodeJS server instance.

```
root@lab-iot-sewio-wks:~# netstat -apn
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      779/systemd-resolve
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1656/sshd
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      730/cupsd
tcp        0      0 127.0.0.1:6010          0.0.0.0:*               LISTEN      5116/sshd: sewiortl
tcp        0      0 0.0.0.0:5000            0.0.0.0:*               LISTEN      3404/node
tcp        0      0 0.0.0.0:5001            0.0.0.0:*               LISTEN      3404/node
```

Figure 8 - Sewio RTLS listening ports.

A quick look at the output of ps also confirmed that NodeJS is running RTLSmanager.js (Figure 9).

```
4 S root      3404  3397  0  80   0 - 302713 -      2021 ?        04:24:48 node ./RTLSmanager.js
0 S root      3429  3404  0  80   0 - 169569 -      2021 ?        01:52:36 /usr/local/bin/node /home/rtlsmanager/lib/report/UdpSocketReceiver.js 5000 UDP
0 S root      3435  3404  0  80   0 - 169752 -      2021 ?        01:59:21 /usr/local/bin/node /home/rtlsmanager/lib/report/UdpSocketReceiver.js 5000 UDP
0 S root      3442  3404  0  80   0 - 169499 -      2021 ?        01:33:58 /usr/local/bin/node /home/rtlsmanager/lib/report/UdpSocketReceiver.js 5000 UDP
0 S root      3449  3404  0  80   0 - 185946 -      2021 ?        04:17:05 /usr/local/bin/node /home/rtlsmanager/lib/report/UdpSocketReceiver.js 5000 UDP
0 S root      3456  3404  0  80   0 - 166685 -      2021 ?        00:00:00 /usr/local/bin/node /home/rtlsmanager/lib/report/TcpSocketReceiver.js 5000 TCP
0 S root      3463  3404  0  80   0 - 166721 -      2021 ?        00:00:00 /usr/local/bin/node /home/rtlsmanager/lib/report/TcpSocketReceiver.js 5000 TCP
0 S root      3470  3404  0  80   0 - 166872 -      2021 ?        00:00:00 /usr/local/bin/node /home/rtlsmanager/lib/report/TcpSocketReceiver.js 5000 TCP
0 S root      3472  3404  0  80   0 - 166809 -      2021 ?        00:00:00 /usr/local/bin/node /home/rtlsmanager/lib/report/TcpSocketReceiver.js 5000 TCP
```

Figure 9 - Sewio RTLS running processes.

The dissection starts inside the `handleIncomingData()` method of *SocketReceiver.js* (Figure 10).

```
processRawData(msg, ip, distribConfig) {
    const unpacked = report.unpack(msg, distribConfig);
    const filteredReports = [];
    for (const unpackedReport of unpacked.reports) {
        if (this._distribConfig.blackboxFilter !== undefined) {
            if (this.bbFilter(unpackedReport) === true) {
                continue;
            }
        }
        if (unpackedReport.reportParts) {
            if (unpackedReport.reportParts.blinkPayload) {
                const blinkPayload = this.handleSpecialStatusHeader(unpackedReport.reportParts.blinkPayload);
                unpackedReport.reportParts.blinkPayload = blinkPayload.payload;
            }
        }
        if (unpackedReport.reportParts && this.filterReport(unpackedReport) && unpackedReport.reportParts.type !== "I") {
            const message = this.serializeReport(unpackedReport.reportParts);
            this.sendReportToRTLSServerStatic(unpackedReport.reportParts.fcode, message);
        }
        filteredReports.push(unpackedReport);
    }
    process.send({ unpackedReports: filteredReports, ip: ip });
    return unpacked.buffer;
}
handleIncomingData(chunk, ip) {
    const unprocessedBuffer = this.processRawData(chunk, ip, this._distribConfig);
    return unprocessedBuffer;
}
```

Figure 10 - handleIncomingData() and processRawData() methods of SocketReceiver.js.

That method immediately calls the `processRawData()` method, that, in turn, calls the `unpack()` function of *unpack.js*, which is 567 lines long (Figure 11).

```
350 function unpack(buf, distribConfig) {
351     const parsedMsgs = [];
352     let generation = "";
353     let origBuffer = Buffer.alloc(buf.length);
354     buf.copy(origBuffer);
355     const delimiter = buf[0];
356     if (delimiter !== DefaultSettings_1.DefaultSettings.separators.OLD_GEN) {
357         generation = "NEW_GEN";
358     }
359     else {
360         generation = "OLD_GEN";
361     }
362     switch (generation) {
```

Figure 11 - Lines 350-362 of `unpack()` function of *unpack.js.*

The first lines of the function revealed that the first byte of a Sewio UWB packet acts as a delimiter. If the separator is `0x23` (the enum is defined in *DefaultSettings.js*), then the packet is a NEW_GEN packet; if `0x7c`, an OLD_GEN packet. The parsing changes on the basis of this value. In this research, we only analyzed a NEW_GEN packet, as only packets of this type were found in the network traffic generated by the purchased solution (Figure 12).

```
363         case "NEW_GEN": {
364             while (buf.length !== 0) {
365                 const delimIndex = indexOfByte(buf, DefaultSettings_1.DefaultSettings.separators.NEW_GEN, undefined);
366                 if (delimIndex === -1) {
367                     console.log("Could not find start of report in this buffer", DefaultSettings_1.DefaultSettings.separators.NEW_GEN, buf);
368                     const emptyBuffer = Buffer.from([]);
369                     return {
370                         reports: parsedMsgs,
371                         buffer: emptyBuffer
372                     };
373                 }
374                 if (delimIndex !== 0) {
375                     buf = buf.slice(delimIndex);
376                 }
377                 if (buf.length < DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.SEPARATOR.len +
378                     DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.CRC.len +
379                     DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.LEN.len) {
380                     logger.general.log("info", "Buffer was so short, that it cannot be report");
381                     return {
382                         reports: parsedMsgs,
383                         buffer: origBuffer
384                     };
385                 }
386                 const anchorCrc = buf.readUInt16LE(DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.CRC.offset);
387                 const reportLen = buf.readUInt16LE(DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.LEN.offset);
388                 let readedLength = DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.SEPARATOR.len +
389                     DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.CRC.len;
390                 let reportParts;
391                 let reportUniversal;
392                 if (buf.length < DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.SEPARATOR.len +
393                     DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.CRC.len +
394                     DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.LEN.len +
395                     reportLen) {
396                     return {
397                         reports: parsedMsgs,
398                         buffer: origBuffer
399                     };
400                 }
```

Figure 12 - Lines 363-400 of `unpack()` function of *unpack.js*.

The parsing of a NEW_GEN packet proceeds by extracting the second and third bytes from the packet, representing its CRC, and the fourth and fifth bytes, the report length. After doing so and performing some length checks, the lines of code responsible for the packet integrity are executed (Figure 13).

```
401                 const singleReport = buf.slice(readedLength, readedLength + reportLen + DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.LEN.len);
402                 const mycrc = crc.crc16ccitt(singleReport);
403                 if (anchorCrc === mycrc) {
404                     buf = buf.slice(readedLength + reportLen + DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.LEN.len);
405                     origBuffer = buf;
406                 }
407                 else {
408                     logger.general.log("warn", "CRC NOT! MATCH");
409                     const nextDelimIndex = indexOfByte(buf, DefaultSettings_1.DefaultSettings.separators.NEW_GEN, delimIndex + DefaultSettings_1.DefaultSettings.
                            binaryReportGeneralSyntax.SEPARATOR.len);
410                     if (nextDelimIndex === -1) {
411                         logger.general.log("warn", "Corrupted Report");
412                         const empty = Buffer.from([]);
413                         return {
414                             reports: parsedMsgs,
415                             buffer: empty
416                         };
417                     }
418                     else {
419                         buf = buf.slice(nextDelimIndex);
420                         continue;
421                     }
422                 }
```

Figure 13 - Lines 401-422 of `unpack()` function of *unpack.js*.

The verification of the packet integrity is crucial from a security perspective, because it affects the ability of an attacker to forge valid synchronization and positioning of packets. As can be noticed on line 402 in Figure 13, and specifically on line 402, Sewio RTLS makes use of the `crc16ccitt()` function to verify the integrity of the packet. This implies that the solution only limits to verify that no corrupted packets are processed by inner code—no security checks are done for preventing an unauthorized actor from creating and injecting packets in the network traffic.

The dissection continues on the basis of its type (called report type) and the included function code, which are extracted in advance from the packet (Figure 14).

```
423                 const reportType = singleReport.readUInt8(DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.TYPE.offset - readedLength);
424                 const fCode = singleReport.readUInt8(DefaultSettings_1.DefaultSettings.binaryReportGeneralSyntax.FCODE.offset - readedLength);
425                 switch (String.fromCharCode(reportType)) {
426                     case "E": {
427                         const datavars = binary.parse(singleReport)
428                             .word16lu("report_len")
429                             .word8("anchor_id6")
430                             .word8("anchor_id5")
```

Figure 14 - Lines 423-430 of `unpack()` function of *unpack.js*.

During our analysis, only traffic of report type “U” was seen, thus we only analyzed this type of packet in our research.

The dissection of this specific type is handled by the `parseReportUniversal()` function, long 276 lines (Figure 15).

```
770                     case "U": {
771                         reportUniversal = parseReportUniversal(singleReport);
772                         if (reportUniversal === undefined) {
773                             logger.general.log("error", "Failed to parse universal report");
774                             break;
775                         }
```

Figure 15 - Lines 770-775 of `unpack()` function of *unpack.js*.

The `parseReportUniversal()` function starts extracting the report length, the anchor MAC address, and the report type from the packet (Figure 16).

```
66  function parseReportUniversal(reportBuffer) {
67      const reportUniversalObj = {};
68      const reportHeader = binary.parse(reportBuffer)
69          .word16lu("report_len")
70          .word8("anchor_id6")
71          .word8("anchor_id5")
72          .word8("anchor_id4")
73          .word8("anchor_id3")
74          .word8("anchor_id2")
75          .word8("anchor_id1")
76          .word8("report_type").vars;
```

Figure 16 - Lines 66-76 of `parseReportUniversal()` function of *unpack.js*.

Finally, it dissects the inner body of the packet, on the basis of its type. A packet can contain multiple submessages (called "options"), that may carry different types of information. For the sake of brevity, we only report the dissection of the most relevant messages (Figure 17):

- The *"syncEmission"* message is sent by the reference anchor and contains the synchronization timestamp when it generated the sync UWB signal;
- The *"syncArrival"* message is sent by the non-reference anchors and contains the synchronization timestamps when they received the UWB signal generated by the reference one;
- The *"blink"* message is sent by all anchors and contains the positioning timestamps.

In the parsing code, it is possible to spot the lines of code that extract the `first_path_amp1, first_path_amp2, first_path_amp3, max_growth_cir, and rx_pream_count` values, which will be mentioned again in section 2.4.

```
100         if (optionsOk === true) {
101             for (const oneOptionBuffer of options) {
102                 const optionNum = oneOptionBuffer.readUInt8(0);
103                 switch (optionNum) {
104                     case 0:
105                         reportUniversalObj.syncEmission = binary.parse(oneOptionBuffer)
106                             .word8("option")
107                             .word16lu("option_len")
108                             .word8("fcode")
109                             .word8("device_id6")
110                             .word8("device_id5")
111                             .word8("device_id4")
112                             .word8("device_id3")
113                             .word8("device_id2")
114                             .word8("device_id1")
115                             .word8("seq_num")
116                             .word8("sync_group_seq_num")
117                             .word64lu("uwb_timestamp")
118                             .vars;
119                         break;
120                     case 1:
121                         reportUniversalObj.syncArrival = binary.parse(oneOptionBuffer)
122                             .word8("option")
123                             .word16lu("option_len")
124                             .word8("fcode")
125                             .word8("device_id6")
126                             .word8("device_id5")
127                             .word8("device_id4")
128                             .word8("device_id3")
129                             .word8("device_id2")
130                             .word8("device_id1")
131                             .word8("seq_num")
132                             .word8("sync_group_seq_num")
133                             .word64lu("uwb_timestamp")
134                             .word16lu("max_noise")
135                             .word16lu("first_path_amp1")
136                             .word16lu("std_noise")
137                             .word16lu("first_path_amp2")
138                             .word16lu("first_path_amp3")
139                             .word16lu("max_growth_cir")
140                             .word16lu("rx_pream_count")
141                             .word16lu("firstpath_index")
142                             .vars;
143                         break;
144                     case 2:
145                         reportUniversalObj.blink = binary.parse(oneOptionBuffer)
146                             .word8("option")
147                             .word16lu("option_len")
148                             .word8("fcode")
149                             .word8("device_id6")
150                             .word8("device_id5")
151                             .word8("device_id4")
152                             .word8("device_id3")
153                             .word8("device_id2")
154                             .word8("device_id1")
155                             .word8("seq_num")
156                             .word64lu("uwb_timestamp")
157                             .word16lu("max_noise")
158                             .word16lu("first_path_amp1")
159                             .word16lu("std_noise")
160                             .word16lu("first_path_amp2")
161                             .word16lu("first_path_amp3")
162                             .word16lu("max_growth_cir")
163                             .word16lu("rx_pream_count")
164                             .word16lu("firstpath_index")
165                             .vars;
166                         break;
```

Figure 17 - Lines 100-166 of `parseReportUniversal()` function of *unpack.js*.

An analysis on the usage of the extracted data by the subsequently executed code was done, to determine if any of those fields were processed inside decryption routines. The analysis confirmed that all data extracted from the network packets are directly used *"as-is"* (an example can be found in Figure 18), including the synchronization and positioning timestamps necessary for reconstructing the positioning data, and no decryption routines were called. Therefore, it is possible to conclude that there is no confidentiality in the network communications exchanged among anchors and server.

```
function convertRawTimestampToString(byte1, byte2, byte3, byte4, byte5) {
    const ticksString = (("0" + byte1.toString(16)).slice(-2)) +
        (("0" + byte2.toString(16)).slice(-2)) +
        (("0" + byte3.toString(16)).slice(-2)) +
        (("0" + byte4.toString(16)).slice(-2)) +
        (("0" + byte5.toString(16)).slice(-2));
    return parseInt(ticksString, 16) * (1 / (128 * 499.2E6));
}
function convertTimestampToString(timestamp) {
    return timestamp * (1 / (128 * 499.2E6));
}
```

Figure 18 - `covertRawTimestampToString()` and `convertTimestampToString()` functions of *unpack.js*.

A Wireshark dissector has been written and is being released to the public in conjunction with this white paper, together with a sample PCAP. Figures 19 and 20 represent the same packets shown at the beginning of this chapter, dissected.

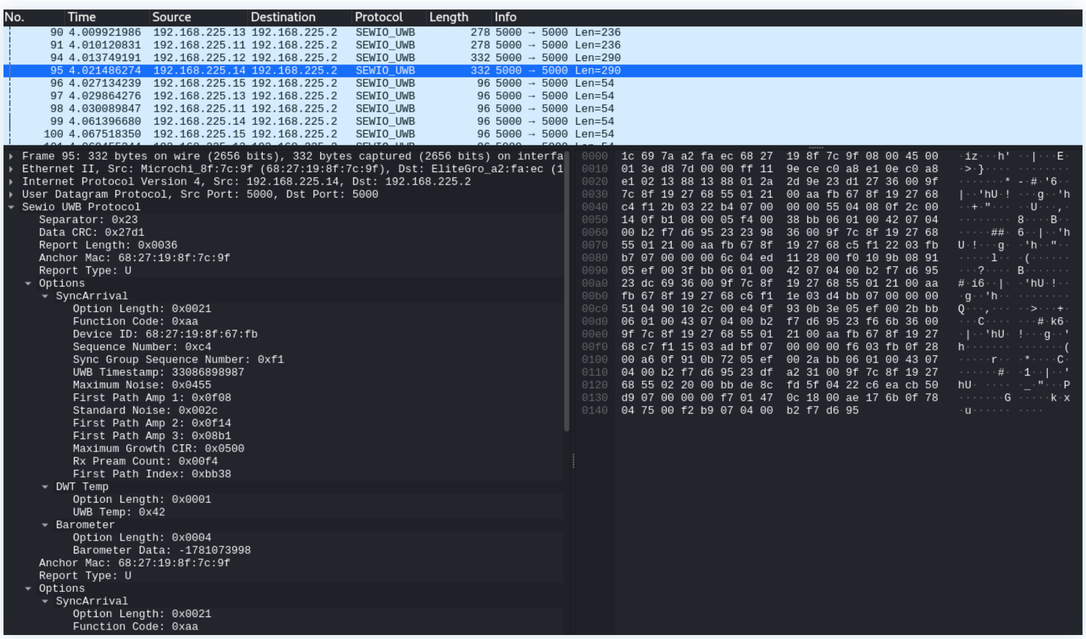


Figure 19 - Sewio RTLS dissected network packet sample.

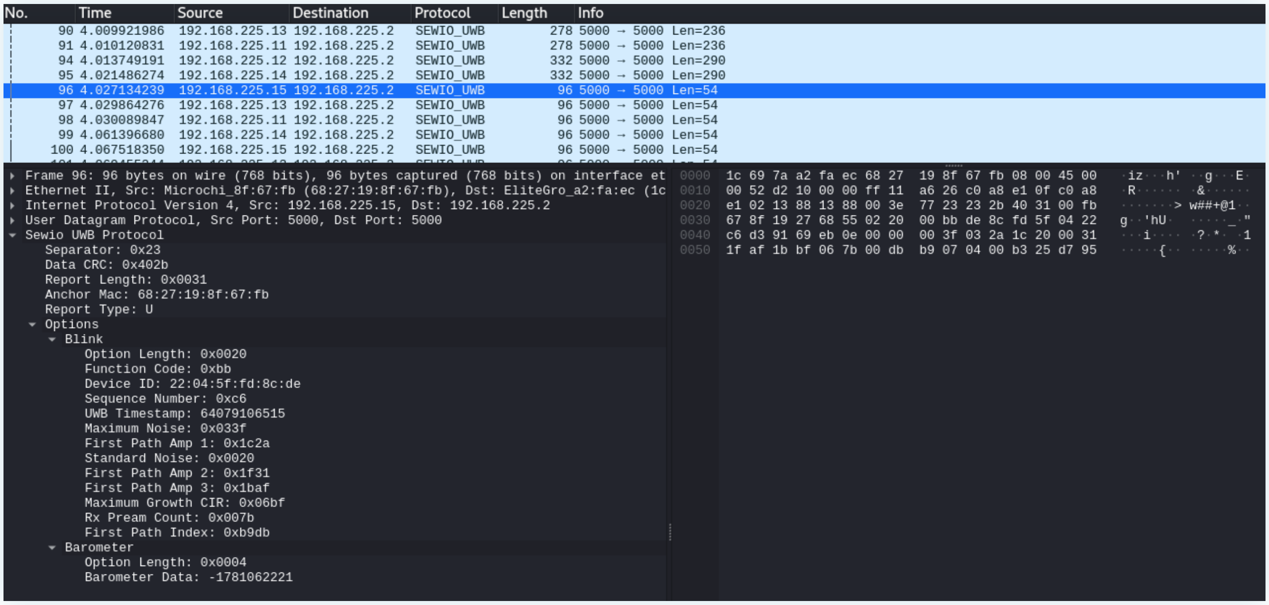


Figure 20 - Sewio RTLS dissected network packet sample (2).

2.3.2 Avalue RTLS

Similar to Sewio, the Avalue RTLS can be configured to use either Ethernet or Wi-Fi as a backhaul for the communications among anchors and server. A Wireshark capture of the network traffic has been done in various conditions, in order to have as many packet samples as possible. Some of these samples are reported in Figure 21 and Figure 22.

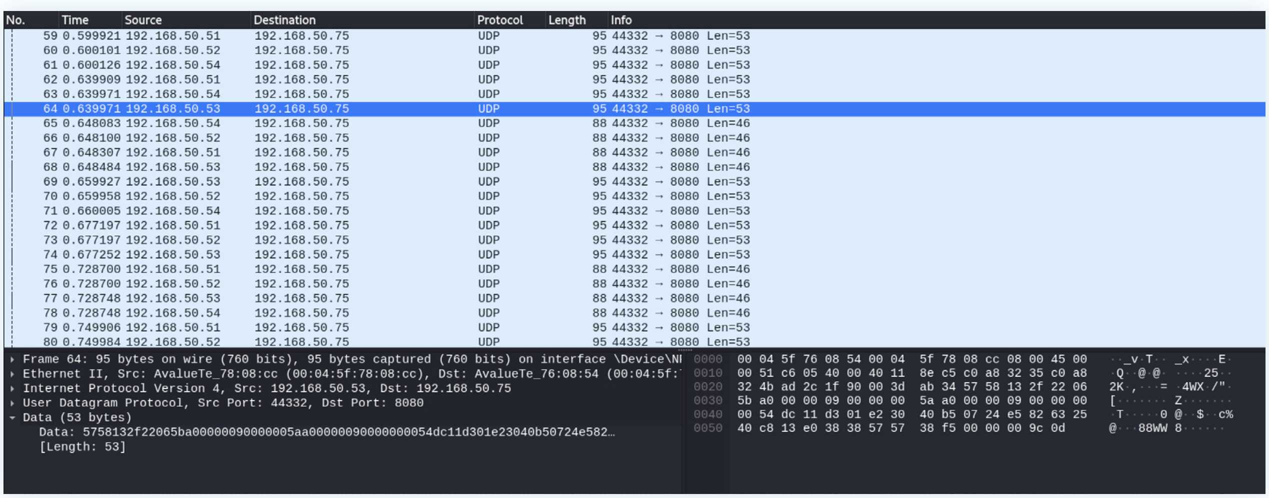


Figure 21 - Avalue RTLS network packet sample.

No.	Time	Source	Destination	Protocol	Length	Info
59	0.599921	192.168.50.51	192.168.50.75	UDP	95	44332 → 8080 Len=53
60	0.600101	192.168.50.52	192.168.50.75	UDP	95	44332 → 8080 Len=53
61	0.600126	192.168.50.54	192.168.50.75	UDP	95	44332 → 8080 Len=53
62	0.639909	192.168.50.51	192.168.50.75	UDP	95	44332 → 8080 Len=53
63	0.639971	192.168.50.54	192.168.50.75	UDP	95	44332 → 8080 Len=53
64	0.639971	192.168.50.53	192.168.50.75	UDP	95	44332 → 8080 Len=53
65	0.648083	192.168.50.54	192.168.50.75	UDP	88	44332 → 8080 Len=46
66	0.648100	192.168.50.52	192.168.50.75	UDP	88	44332 → 8080 Len=46
67	0.648307	192.168.50.51	192.168.50.75	UDP	88	44332 → 8080 Len=46
68	0.648484	192.168.50.53	192.168.50.75	UDP	88	44332 → 8080 Len=46
69	0.659927	192.168.50.53	192.168.50.75	UDP	95	44332 → 8080 Len=53
70	0.659958	192.168.50.52	192.168.50.75	UDP	95	44332 → 8080 Len=53
71	0.660005	192.168.50.54	192.168.50.75	UDP	95	44332 → 8080 Len=53
72	0.677197	192.168.50.51	192.168.50.75	UDP	95	44332 → 8080 Len=53
73	0.677197	192.168.50.52	192.168.50.75	UDP	95	44332 → 8080 Len=53
74	0.677252	192.168.50.53	192.168.50.75	UDP	95	44332 → 8080 Len=53
75	0.728700	192.168.50.51	192.168.50.75	UDP	88	44332 → 8080 Len=46
76	0.728700	192.168.50.52	192.168.50.75	UDP	88	44332 → 8080 Len=46
77	0.728748	192.168.50.53	192.168.50.75	UDP	88	44332 → 8080 Len=46
78	0.728748	192.168.50.54	192.168.50.75	UDP	88	44332 → 8080 Len=46
79	0.749906	192.168.50.51	192.168.50.75	UDP	95	44332 → 8080 Len=53
80	0.749984	192.168.50.52	192.168.50.75	UDP	95	44332 → 8080 Len=53

```
› Frame 65: 88 bytes on wire (704 bits), 88 bytes captured (704 bits) on interface \Device\NI
› Ethernet II, Src: AvalueTe_78:08:d2 (00:04:5f:78:08:d2), Dst: AvalueTe_76:08:54 (00:04:5f:
› Internet Protocol Version 4, Src: 192.168.50.54, Dst: 192.168.50.75
› User Datagram Protocol, Src Port: 44332, Dst Port: 8080
⌄ Data (46 bytes)
    Data: 57581128f500200000000130000005ea0000009000000d41790f346dbfe3f0044980c1122…
    [Length: 46]

0000  00 04 5f 76 08 54 00 04  5f 78 08 d2 08 00 45 00
0010  00 4a 21 3b 40 00 40 11  33 96 c0 a8 32 36 c0 a8
0020  32 4b ad 2c 1f 90 00 36  01 c5 57 58 11 28 f5 00
0030  20 00 00 00 13 00 00 00  5e a0 00 00 09 00 00 00
0040  d4 17 90 f3 46 db fe 3f  00 44 98 0c 11 22 22 3e
0050  3e 29 ed 00 00 00 b2 0a
```

Figure 22 - Avalue RTLS network packet sample (2).

As can be noticed again with Sewio, the Avalue RTLS uses a custom, unknown binary network protocol for this specific purpose, with no immediately recognizable standard data structures. It is thus necessary to reverse engineer the server software again.

A quick look at the server revealed that a Tomcat instance is listening on port 8080/udp, the destination to which Avalue anchors (Ips: 192.168.50.{51,52,53,54}) were noticed sending the network traffic (Figure 23).

Listening Ports

Image	PID	Address	Port	Protocol	Firewall Status
tomcat8.exe	2648	IPv6 unspecified	6000	UDP	Allowed, not restricted
tomcat8.exe	2648	IPv4 unspecified	6000	UDP	Allowed, not restricted
tomcat8.exe	2648	IPv4 loopback	8065	TCP	Allowed, not restricted
tomcat8.exe	2624	IPv6 unspecified	8080	TCP	Allowed, not restricted
tomcat8.exe	2624	IPv4 unspecified	8080	TCP	Allowed, not restricted
tomcat8.exe	2624	IPv6 unspecified	8080	UDP	Allowed, not restricted
tomcat8.exe	2624	IPv4 unspecified	8080	UDP	Allowed, not restricted
tomcat8.exe	2624	IPv4 loopback	8085	TCP	Allowed, not restricted
tomcat8.exe	2648	IPv6 unspecified	8686	TCP	Allowed, not restricted
tomcat8.exe	2648	IPv4 unspecified	8686	TCP	Allowed, not restricted

Figure 23 - Avalue RTLS listening ports.

By accessing the Tomcat manager installed on the server, it is possible to determine that the only custom application running on the system is *"uwb-lib"* (Figure 24).

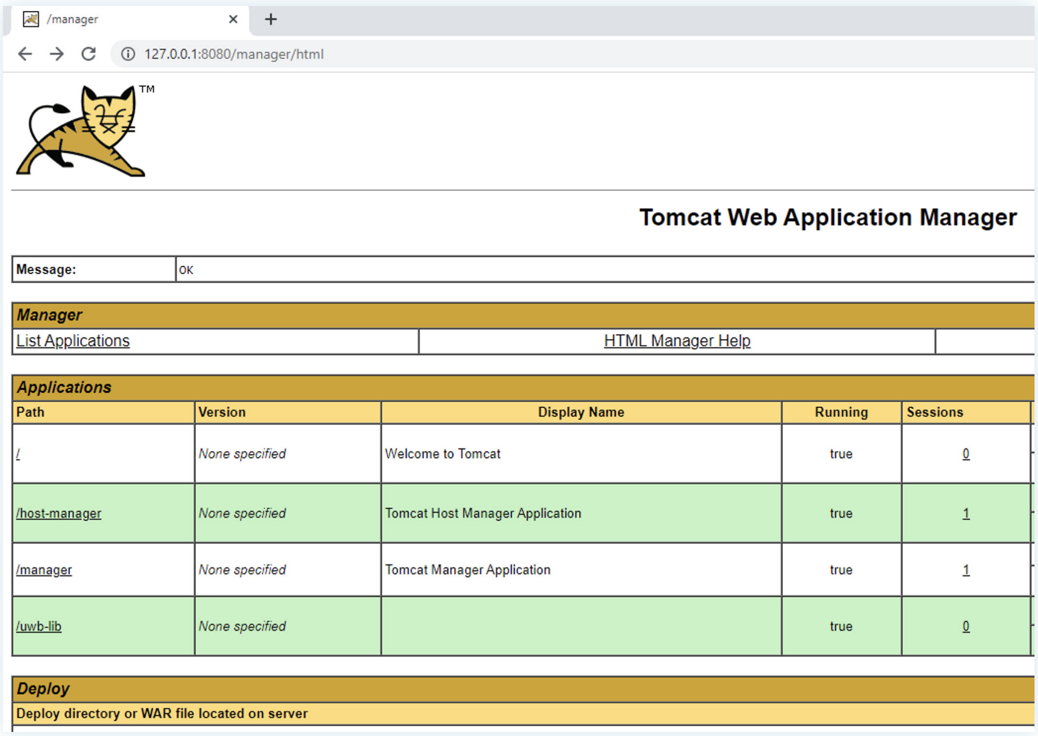


Figure 24 - Applications running on Tomcat server.

In order to decompile the Java code of the application, we decided to use the "Enhanced Class Decompiler" plugin inside a local Eclipse installation, which outputs decompiled Java code straight into Eclipse and embeds multiple decompilation tools (JD, Jad, FernFlower, CFR, Procyon). Notably, during this analysis, FernFlower was used, which experimentally proved able to produce quality decompiled code.

The dissection starts inside the `handlePacket()` method of `UwbParserManager` (Figure 25).

```
    public AbstractUwbPacket handlePacket(byte[] bytes) {
        AbstractUwbPacket packet = null;
        if (ArrayUtils.isNotEmpty(bytes) && UwbLibUtils.isUwbPacket(bytes)) {
            BaseUwbPacketParser parser = (BaseUwbPacketParser) this.packetTypeToParserMap
                    .get(UwbLibUtils.getPacketType(bytes));
            if (parser != null) {
                packet = parser.parse(bytes);
            }
        }

        return packet;
    }
}
```

Figure 25 - `handlePacket()` method of *UwbParserManager.*

The `isUwbPacket()` method immediately unveiled that the first two bytes of an Avalue UWB packet are fixed to the values `0x57` and `0x58`. Additionally, a brief analysis of the `getPacketType()` method revealed that the third byte identifies the type (Figure 26).

```java
    public static boolean isUwbPacket(byte[] bytes) {
        if (ArrayUtils.isEmpty(bytes)) {
            return false;
        } else {
            boolean pass = false;
            if (bytes[0] == 87 && bytes[1] == 88) {
                pass = true;
            }

            return pass;
        }
    }

    public static PacketType getPacketType(byte[] bytes) {
        if (isUwbPacket(bytes)) {
            return PacketType.infer(bytes[2]);
        } else {
            throw new RuntimeException("The input is not valid uwb packet data.");
        }
    }

```

Figure 26 - `isUwbPacket()` and `getPacketType()` methods of *UwbLibUtils*.

A look at the PacketType class revealed the enum with all possible packet types (Figure 27). Notably, although multiple types are defined, during normal operations only two types of packets can be seen in the network traffic:

- "*CCP*" packets, the synchronization packets described in the previous chapter;
- "*TDoA*" packets, the positioning packets.

```java
 PacketType {
'te) 0), ANCHOR_FUNCTION_ACK((byte) 0), ToF((byte) 16), TDoA((byte) 17), SOS((byte) 18), CCP(
 (byte) 19), ANCHOR_COMMAND((byte) 20), RANGING_LISTENING((byte) 21), ANCHOR_REGISTRATION(
        (byte) 32), TAG_REGISTRATION((byte) 33), HEARTBEAT((byte) -96), ANCHOR_EXPLORATION(
            (byte) -95), ANCHOR_EXPLORATION_RESPONSE((byte) -95), REBOOT((byte) -80), UPGRADE(
                (byte) -79), RESTORE_CONFIGURATION((byte) -78), ANCHOR_FUNCTION_CONFIGURATION(
                    (byte) 48), MUTUAL_RANGING((byte) 54), ANCHOR_UWB_MODULE_CONFIGURATION(
                        (byte) 55), SLAVE_ANCHOR_ID_CONFIG((byte) 56), ANCHOR_GROUP_CONFIG(
                            (byte) 57), STATIC_TIME_SLICING_CONFIG(
                                (byte) 58), DYNAMIC_TIME_SLICING_CONFIG(
                                    (byte) 60), SYSTEM_SYNC_CONFIG(
                                        (byte) 61), RESPONSE(
                                            (byte) 102), UWB_DEBUG(
                                                (byte) 103), WIFI_ENABLE(
                                                    (byte) 112), WIFI_CONFIGURATION(
                                                        (byte) 113), ANCHOR_IP_CONFIGURATION(
                                                            (byte) 114), UWB_RF_GATEWAY_CONFIGURATION(
                                                                (byte) 115), UWB_MODULE_QUERY(
                                                                    (byte) 80), UWB_MODULE_QUERY_RESPONSE(
                                                                        (byte) 80), NETWORK_CONFIGURATION_QUERY(
                                                                            (byte) 116), NETWORK_CONFIGURATION_QUERY_RESPONSE(
                                                                                (byte) 116);
```

Figure 27 - All available packet types in Avalue UWB protocol.

An implementation of the `parser()` method was found in the `UwbPacketParserTemplate` class (Figure 28).

```java
    public AbstractUwbPacket parse(byte[] bytes) {
        if (!this.packetCheckerExists()) {
            throw new RuntimeException("Packet checker must be specified.");
        } else {
            AbstractUwbPacket packet = null;
            if (ArrayUtils.isNotEmpty(bytes)) {
                ByteBuffer byteBuffer = ByteBuffer.wrap(bytes).order(ByteOrder.LITTLE_ENDIAN);
                boolean valid = this.processHeader(byteBuffer);
                if (valid) {
                    this.processCheckSum(byteBuffer);
                    packet = this.processBody(byteBuffer);
                }
            }

            return packet;
        }
    }

    protected abstract boolean processHeader(ByteBuffer var1);

    protected abstract AbstractUwbPacket processBody(ByteBuffer var1);

    protected void processCheckSum(ByteBuffer byteBuffer) {
        if (!this.packetChecker.check(byteBuffer.array())) {
            throw new RuntimeException(String.format("Failed to pass packet checker (input: %s).",
                    ArrayUtils.toString(byteBuffer.array())));
        }
    }
```

Figure 28 - `parse()` and `processCheckSum()` methods of *UwbPacketParserTemplate*.

The `processHeader()` method was implemented in `BaseUwbPacketParser`. This allowed us to discover that the fourth byte is the length of the body (Figure 29).

```java
    protected boolean processHeader(ByteBuffer byteBuffer) {
        RuntimeException ex = null;
        if (!UwbLibUtils.isUwbPacket(byteBuffer.array())) {
            ex = new RuntimeException("The input data is not valid UWB packet.");
        }

        if (this.supportedPacketType != UwbLibUtils.getPacketType(byteBuffer.array())) {
            ex = new RuntimeException(
                    String.format("Not matching packet type (input: %s).", ArrayUtils.toString(byteBuffer.array())));
        }

        boolean valid = true;
        if (ex != null) {
            throw ex;
        } else {
            byteBuffer.get();
            byteBuffer.get();
            byteBuffer.get();
            int bodyLength = Byte.toUnsignedInt(byteBuffer.get());
            byte[] bytes = byteBuffer.array();
            int actualBodyLength = bytes.length - this.headerLength - this.checkSumLength;
            if (bodyLength != actualBodyLength) {
                valid = false;
            }

            byteBuffer.position(4);
            return valid;
        }
    }
```

Figure 29 - `processHeader()` method of *BaseUwbPacketParser*.

Indeed, the `processCheckSum()` method is interesting from a security perspective, as its implementation directly impacts the ability to forge accepted UWB packets in subsequent injection attacks. A look at the `check()` method in `SimplePacketChecker` not only revealed that the checksum is just 2-byte long (the last two bytes of a packet), but also that it is simply the sum of all previous bytes (Figure 30). Although this might be enough to distinguish and discard accidentally corrupted packets from valid ones, it is evident that this mechanism does not add any protection against deliberate attacks.

```java
    public boolean check(byte[] byteArray) {
        boolean result = false;
        if (ArrayUtils.isNotEmpty(byteArray) && byteArray.length >= this.checkSumLength) {
            int checkSumAnswer = UwbLibUtils.toUnsignedShort(UwbLibUtils
                    .getShort(Arrays.copyOfRange(byteArray, byteArray.length - this.checkSumLength, byteArray.length)));
            int sum = 0;

            for (int idx = 0; idx < byteArray.length - this.checkSumLength; ++idx) {
                sum += UwbLibUtils.toUnsignedByte(byteArray[idx]);
            }

            result = checkSumAnswer == sum;
        }

        return result;
    }
```

Figure 30 - `check()` method of *SimplePacketChecker.*

If the `processCheckSum()` method is passed, the parsing continues with the `processBody()` method, which depends on the actual packet type. Following, the `processBody()` methods of *CCP* and *TDoA* packets are reported, that are the UWB synchronization and positioning packets (Figure 31 and Figure 32). It is important to notice that, in these methods, it is possible to spot the exact location of the synchronization timestamps in the *CCP* packets, and of the positioning timestamps in the *TDoA* packets.

```java
    protected AbstractUwbPacket processBody(ByteBuffer byteBuffer) {
        return (new Builder()).setOrder(UwbLibUtils.toUnsignedShort(byteBuffer.getShort()))
                .setSlaveId(byteBuffer.getLong()).setMasterId(byteBuffer.getLong()).setStatus(byteBuffer.get())
                .setTxsTimestamp(byteBuffer.getDouble()).setRxsTimestamp(byteBuffer.getDouble())
                .setSignalData(this.generalDataPacketParser == null
                        ? null
                        : this.generalDataPacketParser.parseSignal(byteBuffer))
                .setExtData(this.generalDataPacketParser == null
                        ? null
                        : this.generalDataPacketParser.parseExtData(byteBuffer))
                .build();
    }
}
```

Figure 31 - `processBody()` method of *CCPPacketParser.*

```java
    protected AbstractUwbPacket processBody(ByteBuffer byteBuffer) {
        return (new Builder()).setOrder(UwbLibUtils.toUnsignedShort(byteBuffer.getShort()))
                .setTagId(byteBuffer.getLong()).setAnchorId(byteBuffer.getLong())
                .setSyncTimestamp(byteBuffer.getDouble())
                .setEventData(this.generalDataPacketParser == null
                        ? null
                        : this.generalDataPacketParser.parseEvent(byteBuffer))
                .setBatData(
                        this.generalDataPacketParser == null ? null : this.generalDataPacketParser.parseBat(byteBuffer))
                .setSignalData(this.generalDataPacketParser == null
                        ? null
                        : this.generalDataPacketParser.parseSignal(byteBuffer))
                .setExtData(this.generalDataPacketParser == null
                        ? null
                        : this.generalDataPacketParser.parseExtData(byteBuffer))
                .build();
    }
}
```

Figure 32 - `processBody()` method of *TDoAPacketParser.*

The `parseEvent()`, `parseBat()`, `parseSignal()`, and `parseExtData()` methods in `BasicGeneralDataPacketParser` conclude the parsing procedure. Of these methods, it is worth mentioning that the `parseSignal()` method performs the extraction of the F`irstPathAmp1`, `FirstPathAmp2`, `FirstPathAmp3`, `MaxGrowthCIR`, and `RxPreamCount` values, mentioned again in section 2.4 (Figure 33).

```java
    public BaseEventData parseEvent(ByteBuffer byteBuffer) {
        return new BasicEventData(byteBuffer.get());
    }

    public BaseBatData parseBat(ByteBuffer byteBuffer) {
        return new BasicBatData(byteBuffer.get());
    }

    public BaseSignalData parseSignal(ByteBuffer byteBuffer) {
        return (new Builder()).setFirstPathAmp1(byteBuffer.getShort()).setFirstPathAmp2(byteBuffer.getShort())
                .setFirstPathAmp3(byteBuffer.getShort()).setMaxGrowthCIR(byteBuffer.getShort())
                .setRxPreamCount(byteBuffer.getShort()).build();
    }

    public BaseExtData parseExtData(ByteBuffer byteBuffer) {
        BaseExtData result = null;
        if (byteBuffer.position() < byteBuffer.array().length - 2) {
            byte extType = byteBuffer.get();
            byte extLen = byteBuffer.get();
            byte[] extData = null;
            if (extLen != 0 && byteBuffer.position() + extLen <= byteBuffer.array().length - 2) {
                extData = new byte[extLen];

                for (int idx = 0; idx < extData.length; ++idx) {
                    extData[idx] = byteBuffer.get();
                }
            }

            result = (new tw.com.gips.uwb.commons.packet.v2.v2_0_xx.data.ext.BasicExtData.Builder())
                    .setExtType(ExtDataType.infer(extType)).setExtData(extData).build();
        }

        return result;
    }
```

Figure 33 - `parseEvent()`, `parseBat()`, `parseSignal()`, and `parseExtData()` methods of *BasicGeneralDataPacketParser.*

After tracking how all these data are used in the following steps of the implemented TDoA algorithm, it was possible to conclude that there is no confidentiality in the network communications exchanged among anchors and server. All data are extracted from the network packets and directly used "as-is" into the functions, including the synchronization and positioning timestamps necessary for reconstructing the positioning data (an example can be found in Figure 34). In fact, in the aforementioned evidence, a scrupulous reader might have noticed that, from the beginning, those data were parsed using specific functions such as `getDouble()`, a strong indication that no cryptography was in place.

```java
    public static <E extends BaseTDoAPacket> boolean isValidTDoATime(E packet) {
        boolean result = false;
        if (packet != null) {
            double time = packet.getSyncTimestamp();
            result = time > 1.0E-10D && time < 17.207401025641026D;
        }

        return result;
    }

```

Figure 34 - `isValidTDoATime()` method of *UwbLibUtils.*

A Wireshark dissector, specifically for the parsing of *CCP* and *TDoA* packets, has been written and is being released to the public in conjunction with this white paper, together with a sample PCAP. Figures 35 and 36 represent the same packets shown at the beginning of this chapter, dissected.

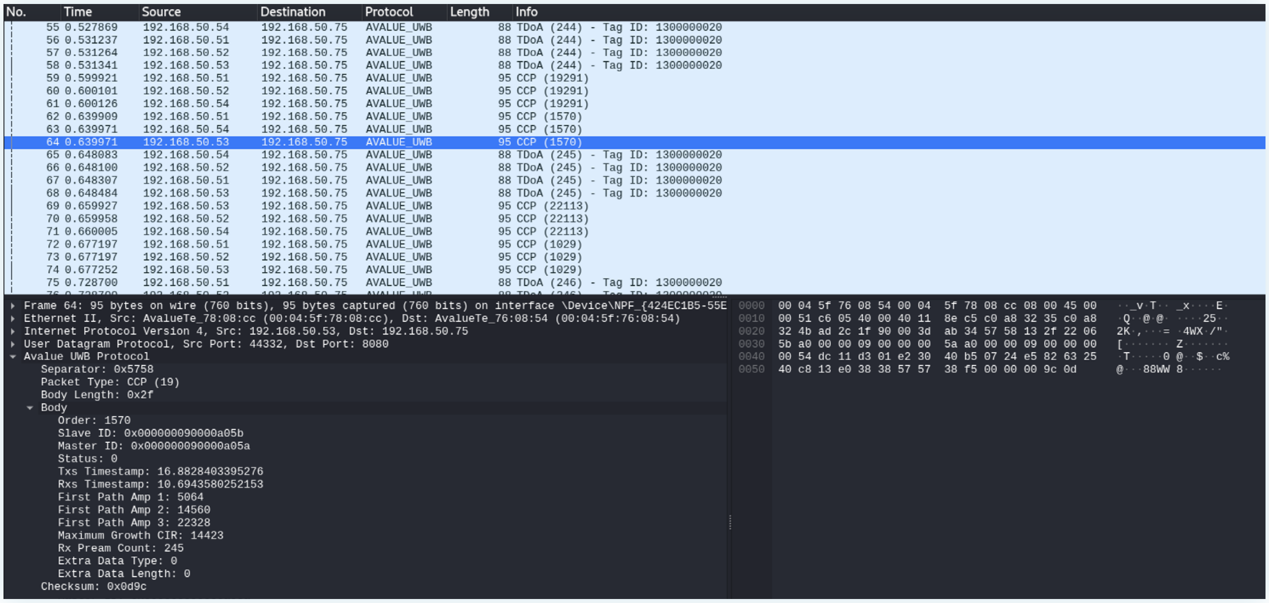


Figure 35 - Avalue RTLS dissected network packet sample.

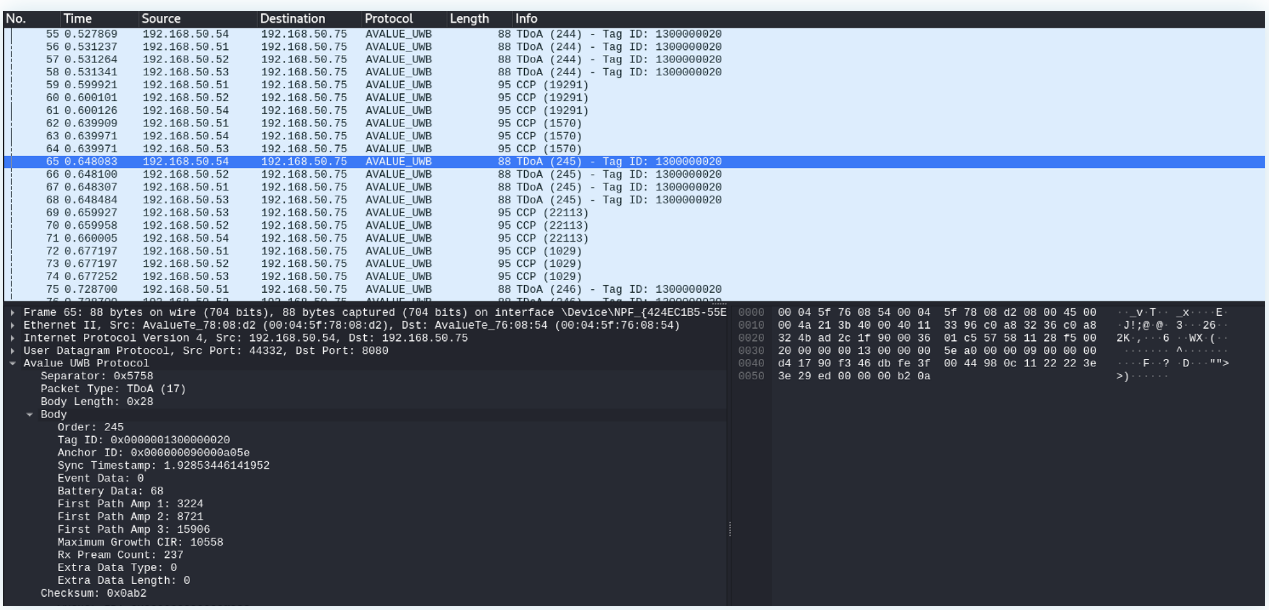


Figure 36 - Avalue RTLS dissected network packet sample (2).

2.4 Anchor Coordinates Prerequisite

In the previous section, we concluded that there is neither confidentiality nor secure integrity mechanisms protecting the communications performed by the analyzed UWB RTLS. However, as stated at the end of section 2.2, to compute the position of a tag, all coordinates of the involved anchors need to be known. This is the most challenging requirement for an attacker, and could make a difference in the ultimate ability to estimate the position of a tag or not. This section is entirely devoted to this specific problem. Notably, we present a technique that completely remote adversaries (the most limiting situation) can exploit to estimate the anchors' coordinates with enough accuracy to mount practical attacks.

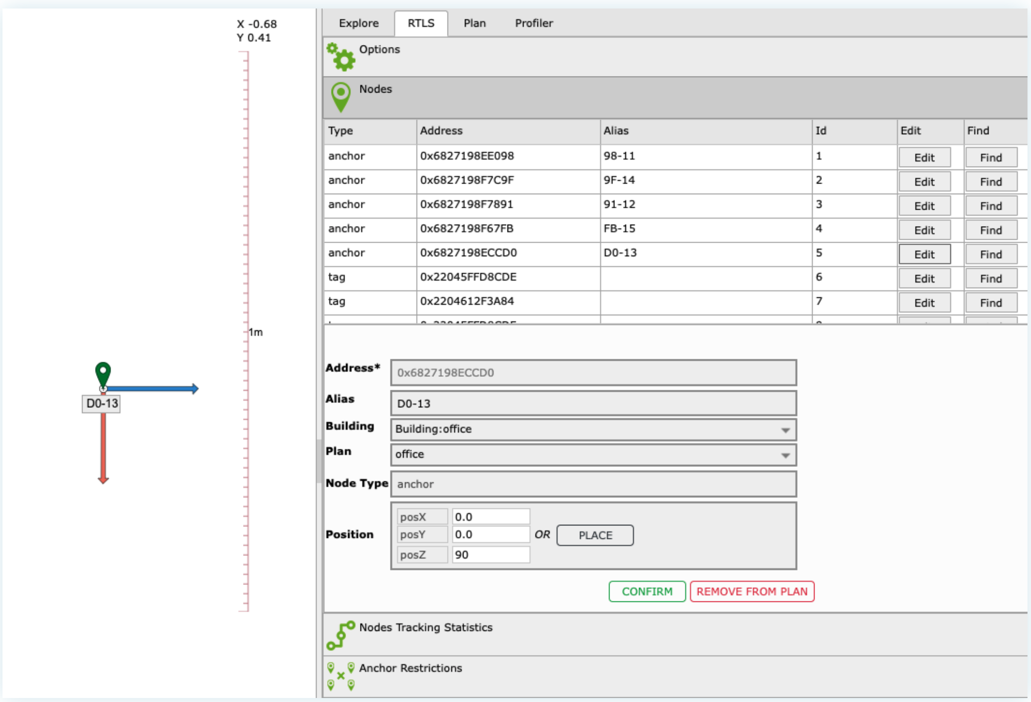


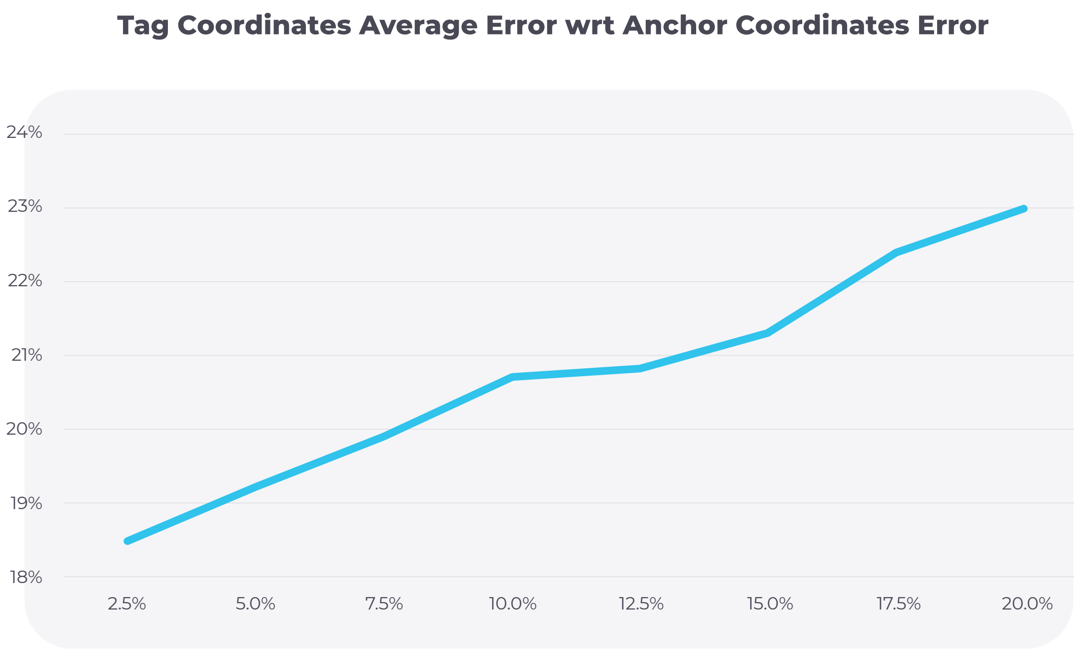
Figure 37 - Anchor coordinates setup in Sewio RTLS.

Normally, the coordinates of the anchors used in an RTLS are manually input as parameters inside the server software at the first installation (Figure 37). Afterwards, in the solutions we analyzed, this information was never found transmitted in the network traffic.

Physical access: If an attacker has physical access to the monitored area, this problem can be solved in a variety of ways:

- If the anchors are mounted in visible positions, obtaining their coordinates is a simple task;
- If the anchors are not mounted in visible positions, an attacker may still be able to estimate their coordinates by measuring the power levels of their transmitted wireless signals (UWB and/or any other wireless technology used by the anchors, such as Wi-Fi). The position where the peak power level is detected is roughly the anchor location.

In fact, according to our tests, the anchor coordinates do not need to be precise to obtain a good estimation of the tag positions. As shown in the following chart, if the anchor coordinates are estimated with an error of less than 10% with respect to the real value, the tag coordinates are computed with an average error of less than 20%; about 50 cm in a 6m x 5m room.



Remote access: If the attacker has no access to the monitored area, they must derive the anchor coordinates only by looking at the traffic the anchors are sending. This is the most challenging condition for an attacker because, the anchor coordinates are never transmitted through the network traffic.

Although no explicit data are transmitted, to this extent, there is important information coming from the anchors that can be leveraged to estimate the distance between each anchor and the reference one.

Together with the locating data, anchors transmit the power level information of the received UWB signal on the wire, to allow the locating server to filter out poorly received wireless packets (Figure 38). In particular, these data are:

- First Path Amplitude point 1 (FP1)
- First Path Amplitude point 2 (FP2)
- First Path Amplitude point 3 (FP3)
- Preamble Accumulation Count (PAC)
- Maximum Growth CIR (MGC)

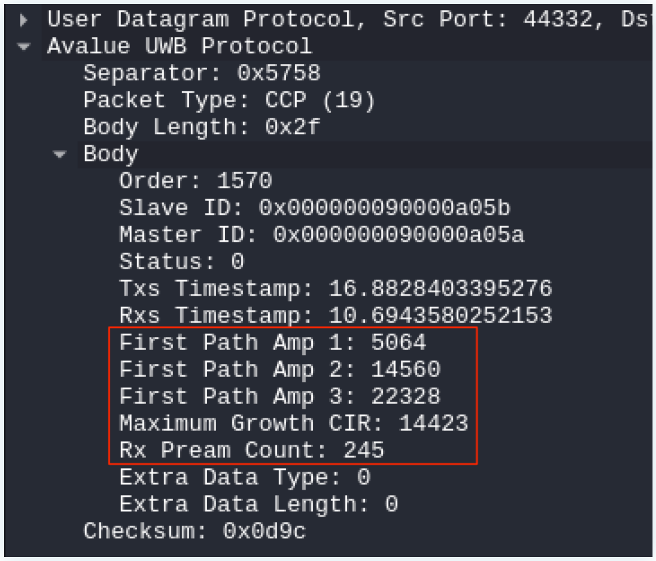


Figure 38 - Power levels transmitted in network traffic.

With these data, two different metrics can be computed, related to the power level of the tag transmission: the First Path Power Level (FPPL) and the Receive Power Level (RPL). According to the documentation of the Decawave DW1000,[20] the UWB chip on which these (and many other) RTLS are based:

$$FPPL = 10 * \log10((FP1^2 + FP2^2 + FP3^2)/PAC^2) - A$$

Eq. 9

$$RPL = 10 * \log10((MGC \times 2^{17}) / PAC^2) - A$$

Eq. 10

Where A is a constant for a Pulse Recurrence Frequency. When working at 16 MHz, it is 115.72; when working at 64MHz, it is 121.74 dB.

It is not possible to directly estimate the absolute distance given a certain power level. Tests were completed, and this estimation seems too influenced by the environmental conditions that exist at the instant of the measurement.

However, what can be done with decent accuracy is to assume that, if the power level information (either first path or total received) is identical (or inside a certain level of acceptance) in all positioning packets generated in a given moment *t0*, the tag *j0* that caused the generation of the aforementioned positioning packets is located exactly (or about exactly) at the same distance from all anchors.

In other words, given a pair of anchors, the difference of the distance between a tag *j0* and anchor *i0* and the tag *j0* and anchor *i1* is 0, thus implying that *GT(i0, j0, t0) = GT(i1, j0, t0)*. This is also true for the reference anchor, thus *GT(reference, j0, t0) = GT(i0, j0, t0)*.

This equation is very important, because, for the reference anchor, the *Clock Skew* is 1 and the time of flight from itself is 0 by definition. Consequently, it is possible to use this equation for each of the other non-reference anchors to estimate their times of flight. From those, the distance of each anchor with respect to the reference anchor can be estimated with enough accuracy.

[20] **"Decawave DW1000,"** Qorvo.

As a matter of fact, this is the equation 5 that was present in section 2.2.2.

$$Delta(i, j, t) = (GT(reference, j, t) - GT(i, j, t)) * c$$

Eq. 5

If the distance from all anchors is identical, this means that:

$$0 = (GT(reference, j0, t0) - GT(i0, j0, t0)) * c$$

Eq. 11

And that:

$$GT(reference, j0, t0) = GT(i0, j0, t0)$$

Eq. 12

However, considering equation 4:

$$GT(i, j, t) = CS(i, t) * (pTs(i, j, t) - sTS(i, t)) + ToF(i)$$

Eq. 4

This means that we can derive:

$$CS(reference, t0) * (pTs(reference, j0, t0) - sTS(reference, t0)) - ToF(reference) = CS(i0, t0) * (pTs(i0, j0, t0) - sTS(i0, t0)) - ToF(i0)$$

Eq. 13

However, considering that CS(reference, t0) = 1 and ToF(reference) = 0 by definition:

$$pTs(reference, j0, t0) - sTS(reference, t0) = CS(i0, t0) * (pTs(i0, j0, t0) - sTS(i0, t0)) - ToF(i0)$$

Eq. 14

And we can conclude that:

$$ToF(i0) = CS(i0, t0) * (pTs(i0, j0, t0) - sTS(i0, t0)) - pTs(reference, j0, t0) + sTS(reference, t0)$$

Eq. 15

This equation is used to obtain an accurate estimation of the distances of all anchors with respect to the reference anchor. However, having the distances is not enough: to compute the position of a tag, the coordinates of the anchors are required.

For this purpose, an adversary can leverage an installation constraint that is common in RTLS: due to dilution of precision problems, RTLS vendors require that anchors are positioned in a shape that is as regular as possible. Ideally, it must be a square whenever possible, at most a rectangle[21] (Figure 39).

An attacker that can listen to the traffic on the wire can also adapt the expected shape on the basis of the number of anchors detected in the communications. For instance, if they detect 4 anchors, it is likely a rectangle; if 6 anchors, it could be a hexagon or a rectangle with two anchors positioned in the middle of the longest side.

[21] "Sewio The Dilution of Precision – Anchor Geometry"

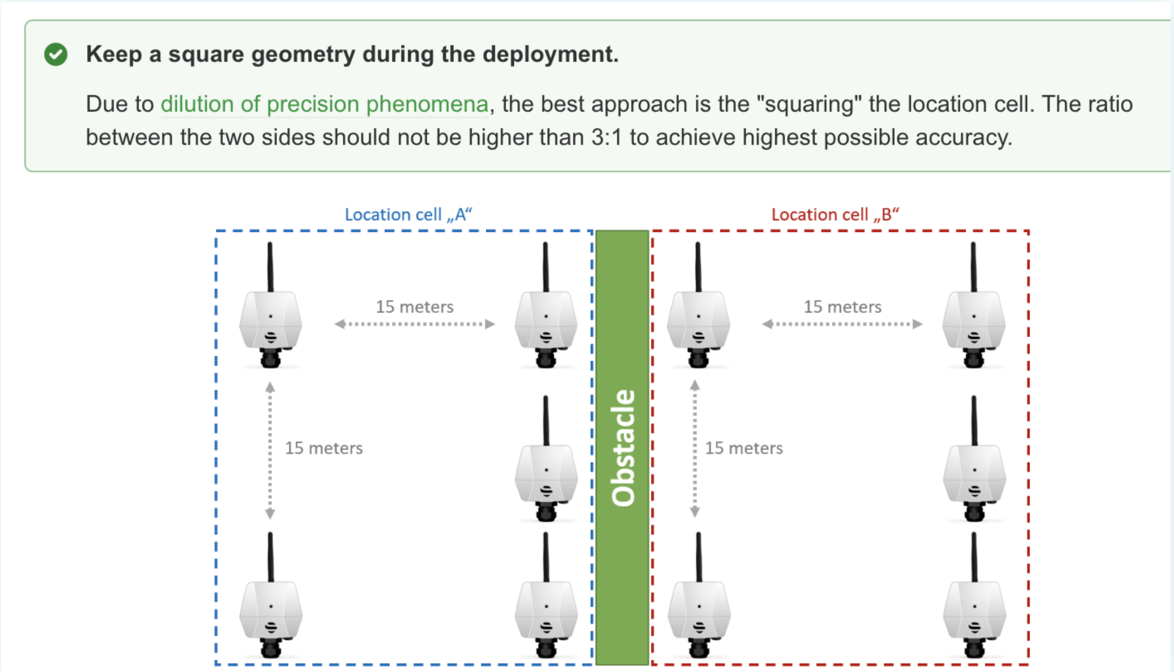


Figure 39 - Sewio anchor deployment guidelines.

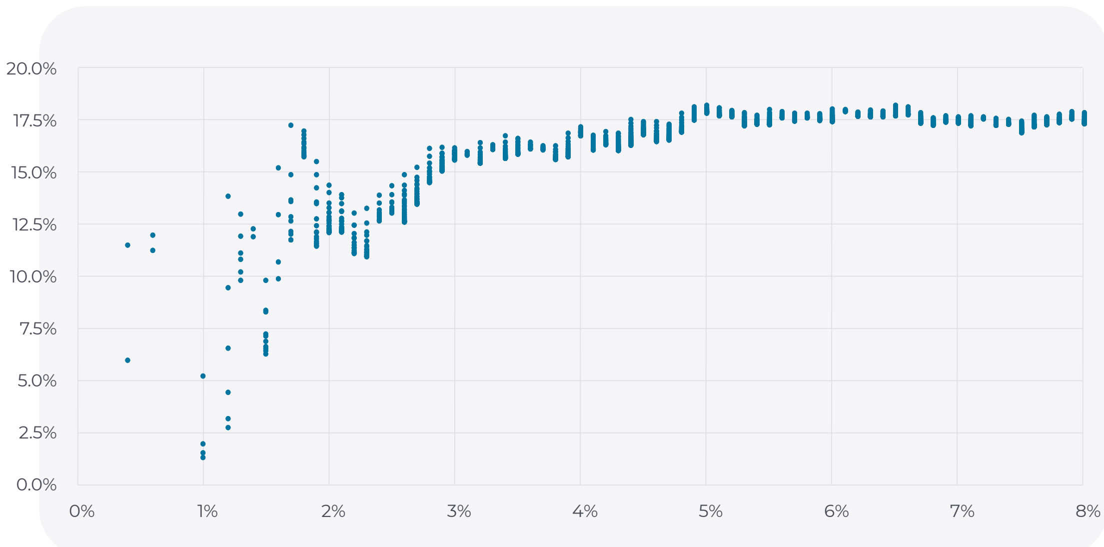
Given that the distance of each anchor with respect to the reference anchor is known, and given that it can now be safely assumed that the anchor map is as regular as possible and usually a rectangle, by arbitrarily setting the reference anchor in position (0;0), the coordinates of all other anchors can be easily estimated, because they will be given by the two shortest distances obtained from the estimation of the times of flight.

For instance, let's say that we determined that the distances of anchors from the reference anchor are 5m, 7m, and 8.5m. It can safely be estimated that the anchor coordinates are (0;0), (5;0), (0;7), and (5;7), with 8.5m being the diagonal of the rectangle. There is also the possibility of the specular result (0;0), (0;5), (7;0), and (7;5), but this is not a problem—it is just a matter of defining a coordinate system and sticking to it.

This was actually tested in the Avalue RTLS, using both FPPL and the RPL. According to the tests done, the best results are obtained using the FPPL with a threshold of 1% between the lowest power level and the highest power level read in a given positioning communication. However, this situation is rare: an attacker may want to use the RPL or raise the threshold in case no suitable communications appear on the wire.

As shown in the chart below, using the FPPL with threshold set to 1%, it was possible to estimate the anchor distances with an error of less than 10% with respect to the real value. Remembering that this translates into an average error of less than 20% during the computation of the tag positions, this can be accurate enough for attack scenarios where cm-level precision is not required.

Anchor Coordinates Average Error wrt First Path Power Level (FPPL) Acceptance Threshold



2.5 Adversary Tactics, Techniques and Procedures (TTPs)

In the previous sections, we defined the scope of our research, described the necessary data and steps to compute the position of a tag, detailed the reverse engineering work that allows timestamps to be located inside the network packets, and explained how an attacker can fulfil the last requirement, that is, estimating the anchors coordinates.

In this chapter, we describe the adversary Tactics, Techniques, and Procedures (TTPs), which is the behavior of an attacker wanting to practically abuse these systems. After discussing how a threat actor can obtain access to the target information, we present the two types of attacks that can be enacted: the passive eavesdropping attack, which allows the position of all tags in the network to be reconstructed, and the active traffic manipulation attack, which allows the position of tags detected by the RTLS to be modified.

2.5.1 Traffic Interception

To perform any meaningful attacks against these RTLS systems, it is first necessary to:

1. Gain a foothold inside the backhaul network used by the anchors and server for their communications;
2. Execute a Man in the Middle (MitM) attack, to intercept all network packets exchanged among anchors and server, and, notably, the synchronization and positioning timestamps.

Network Access: Both Sewio and Avalue RTLS allow either Ethernet or Wi-Fi to be used for the network backhaul.

Gaining access to an Ethernet network requires that an attacker either compromise a computer connected to that network, or surreptitiously add a rogue device to the network. Besides of course depending on the computer security practices adopted by the asset owner, the complexity of these actions also varies on the basis of the chosen deployment configuration. As a matter of fact, some UWB RTLS allow anchors and a server to be placed in heterogeneous subnetworks, with the only requirement being that those networks are routed[22] (Figure 40). In such cases, there is an increased likelihood of successfully compromising or surreptitiously adding one system in any of the networks traversed by the network communications, if those networks and devices are not adequately designed and protected.

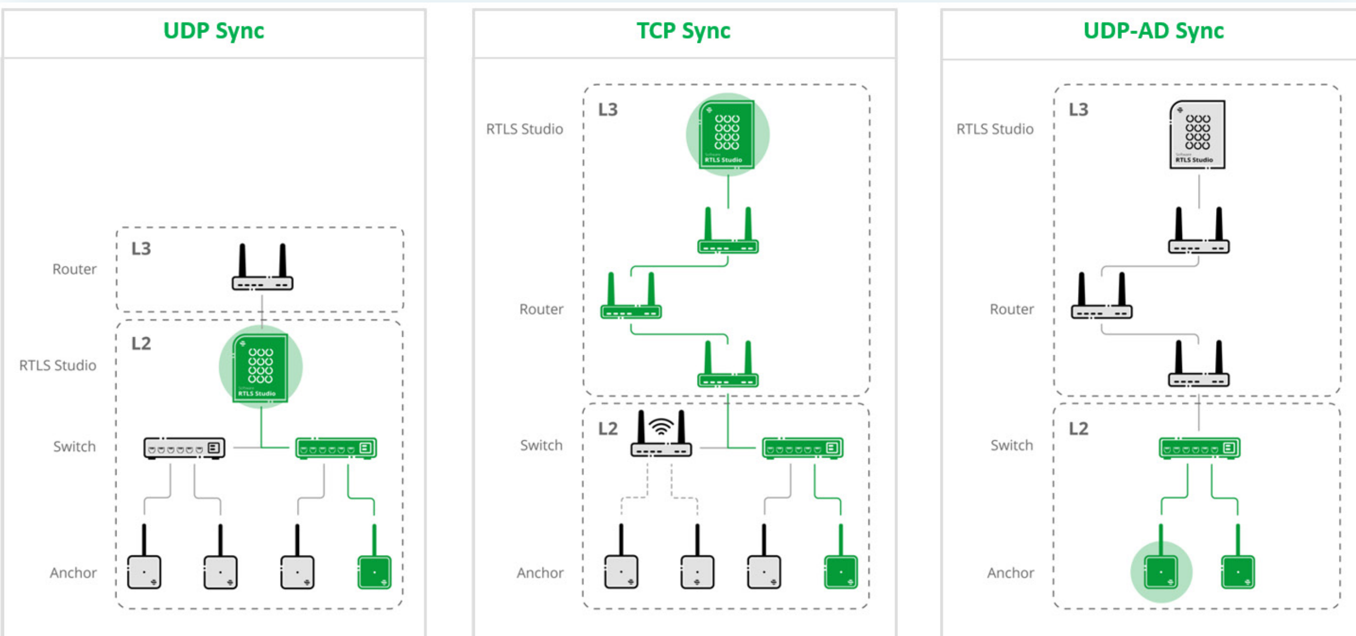


Figure 40 - Deployment configurations available on Sewio RTLS.

As for Wi-Fi, both solutions support WPA2-PSK as the security protocol for protecting the wireless communications. Thus, gaining access to the network usually requires either knowledge of the WPA2 password, or the exploitation (if any) of vulnerabilities in the wireless network appliances.

[22] **"TDMA Synchronization,"** Sewio.

As for the first point, it must be stated that both solutions, out of the box, feature a static password that can be found in the public documentation[23,24] (Figure 41). In case an asset owner does not change it, obtaining access to the backhaul network is simple.

Avalue have completed all the pre-settings before shipping, please refer below information for login.

http://tplinkwifi.net

Password: 7[illegible]

Wi-Fi credentials for admin access are:

SSID: Artemis

Password: 7[illegible]

Avalue also provide a configuration file that can be simply re-loaded into the router.

To load the file into the router, follow these steps:

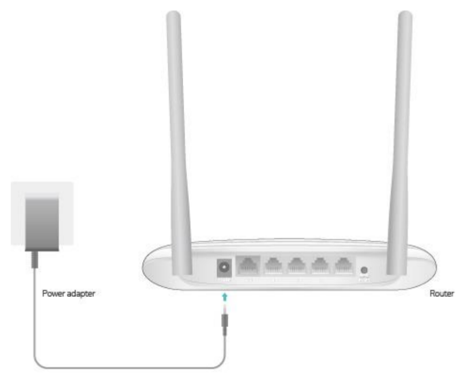


Figure 41 - Default WPA2-PSK password on Avalue RTLS.

Man in the Middle (MitM): Depending on the position gained, just obtaining access to the network may not be enough. Since in both RTLS the anchors do not send the information via broadcast packets, a MitM attack might still be required to intercept the communications. However, in the tests executed, it was possible to conduct a MitM on both solutions via standard ARP spoofing attacks just by having one foothold on the backhaul network, and without the RTLSs showing any warnings or abnormal behavior that may alert an operator.

The following code command launched from a workstation connected to a generic port of the backhaul network switch allowed all anchors-to-server communications to be intercepted, as well as all server-to-anchors ones:

```
arpspoof -i attacker_eth -t server_ip anchor1_ip &
arpspoof -i attacker_eth -t anchor1_ip server_ip
```

[23] **"Network – Wi-Fi,"** Sewio.

[24] "Avalue Renity Artemis Enterprise Kit Quick Reference Guide," (publicly available to customers).

By repeating this command for all the anchors in the system, it is quickly possible to intercept all the generated traffic.

Figure 42 and Figure 43 report the log of the code commands and the network traffic captured via Wireshark of a successful MitM attack executed against both solutions.

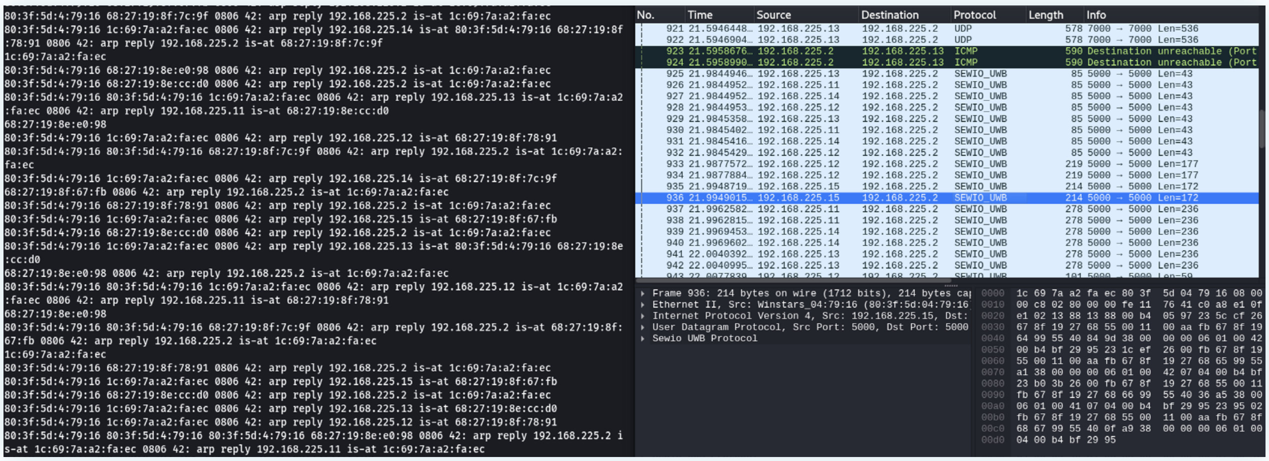


Figure 42 - MitM attack against Sewio RTLS.

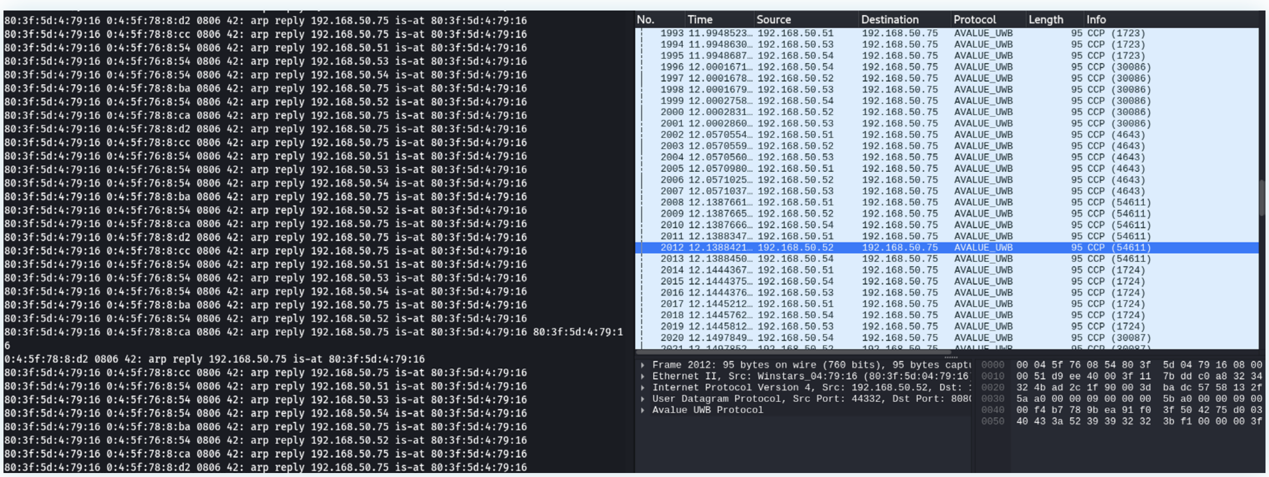


Figure 43 - MitM attack against Avalue RTLS.

2.5.2 Passive Eavesdropping Attacks

If an attacker has managed to obtain access to an RTLS network and has successfully launched a MitM attack against the anchors and the server, they can now reconstruct the position of tags in the network by simply following one of the standard TDoA algorithms available in literature, such as the one explained in section 2.2.2.

In this section, as an example, an execution trace of the previously mentioned algorithm is reported. The aim is to locate an Avalue RTLS tag when positioned roughly in the center of a monitored room. Figure 44 shows the position of the tag as depicted by the RTLS web application.

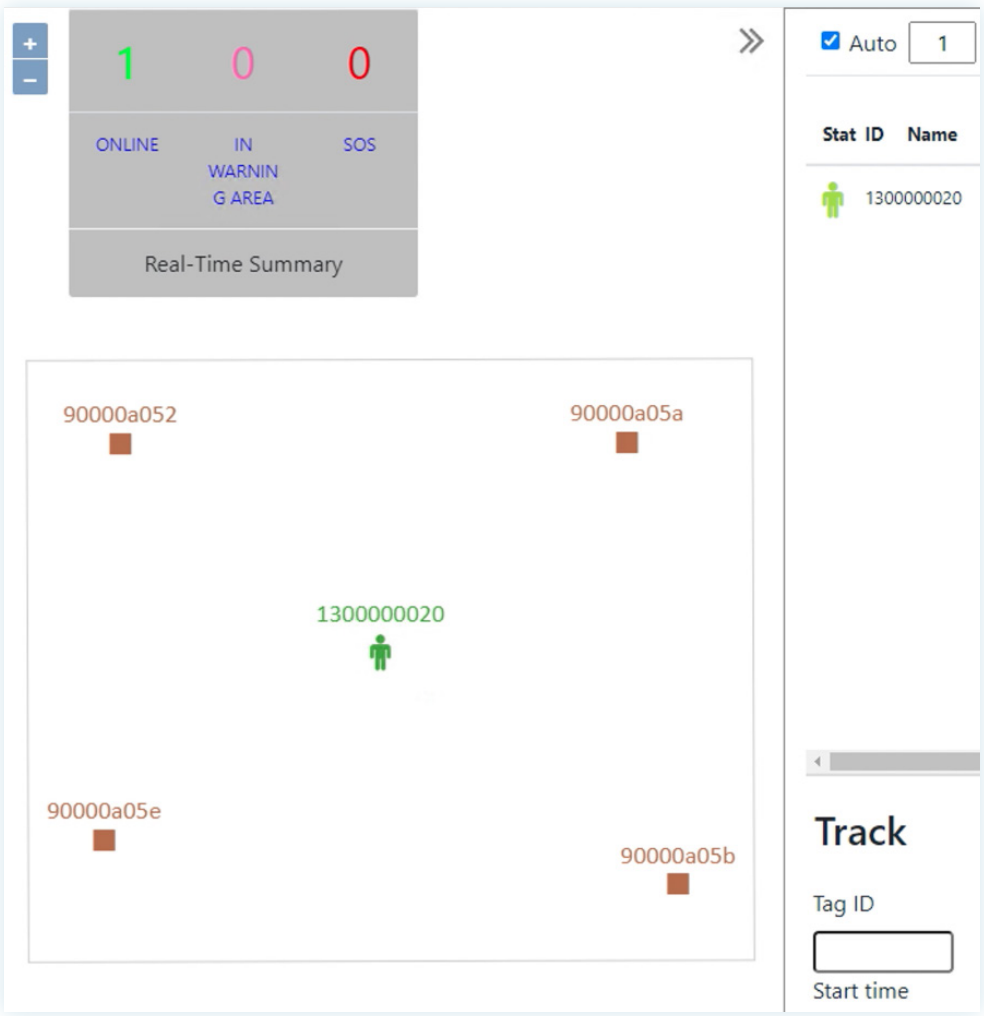


Figure 44 - Target tag position as shown by the Avalue RTLS.

90000a052, 90000a05a, 90000a05b, and 90000a05e are the four anchors in use by the RTLS. 1300000020 is the tag. The four anchors are located at the following 2D coordinates:

- Coordinates of 90000a052 = (0, 0)
- Coordinates of 90000a05a = (6.3308356, 0)
- Coordinates of 90000a05b = (6.989999001, -5.28999995)
- Coordinates of 90000a05e = (-0.2500003, -4.91999999)

To start with, it is necessary to compute the global times of the positioning packets, so that they can be compared together. As indicated by equation 4, besides the collection of all positioning timestamps, this requires capturing the synchronization timestamps of the same iteration, as well as the synchronization timestamps of the previous iteration.

By looking at the Wireshark traffic (Figure 45), the timestamps to capture are the ones included in the packets highlighted in pink. Notice that, in this capture, the reference anchor was 90000a052, and that, in the Avalue RTLS, the synchronization timestamp of the reference anchor is duplicated in all synchronization packets sent by the non-reference anchors.

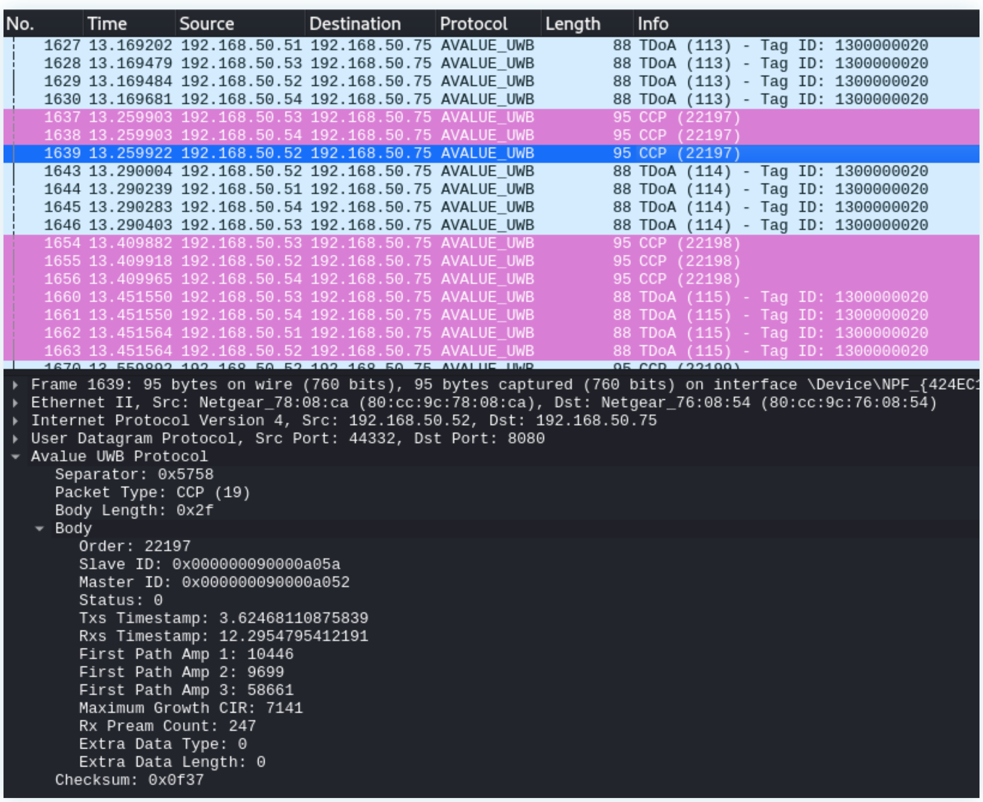


Figure 45 - Network traffic generated by the Avalue RTLS.

The following information was thus extracted:

sTs(reference, t0) = 3.624681109
sTs(90000a05a, t0) = 12.29547954
sTs(90000a05b, t0) = 6.106995629
sTs(90000a05e, t0) = 14.38947882

sTs(reference, t1) = 3.774681365
sTs(90000a05a, t1) = 12.44547979
sTs(90000a05b, t1) = 6.256995869
sTs(90000a05e, t1) = 14.5394791

pTs(reference, 1300000020, t1) = 3.967019137
pTs(90000a05a, 1300000020, t1) = 12.63781749
pTs(90000a05b, 1300000020, t1) = 6.449333591
pTs(90000a05e, 1300000020, t1) = 14.7318168

By using equation 3, it is possible to compute the *Clock Skews* of all anchors for that iteration.

CS(reference, t1) = (3.774681365 - 3.624681109)/(3.774681365 - 3.624681109) = 1

CS(90000a05a, t1) = (3.774681365 - 3.624681109)/(12.44547979 - 12.29547954) = 1.0000000399999

CS(90000a05b, t1) = (3.774681365 - 3.624681109)/(6.256995869 - 6.106995629) = 1.0000001066665

CS(90000a05e, t1) = (3.774681365 - 3.624681109)/(14.5394791 - 14.38947882) = 0.9999998400003

With the coordinates reported above, it is possible to derive the times of flight for all non-reference anchors with respect to the reference one, by simply computing their distance and dividing it by the speed of light, i.e., the approximated speed of a signal travelling through air.

ToF(reference) = sqrt((0 - 0)^2 + (0 - 0)^2)/c = 0 s

ToF(90000a05a) = sqrt((6.3308356 - 0)^2 + (0 - 0)^2)/c = 2.11174E-08 s

ToF(90000a05b) = sqrt((6.989999001 – 0)^2 + (-5.28999995 – 0)^2)/c = 2.92405E-08 s

ToF(90000a05e) = sqrt((-0.2500003 – 0)^2 + (-4.91999999 – 0)^2)/c = 1.64325E-08 s

With these data, it is possible to derive the global times of the positioning packets.

*GT(reference, 1300000020, t1) = 1 * (3.967019137 - 3.774681365) - 0 = 0.192337772 s*

*GT(90000a05a, 1300000020, t1) = 1.0000000399999 * (12.63781749 - 12.44547979) + 2.11174E-08 = 0.192337773 s*

*GT(90000a05b, 1300000020, t1) = 1.0000001066665 * (6.449333591 - 6.256995869) + 2.92405E-08 = 0.192337772 s*

*GT(90000a05e, 1300000020, t1) = 0.9999998400003 * (14.73181684 - 14.5394791) + 1.64325E-08 = 0.192337773 s*

Having found the global times of the positioning packets, it is now necessary to compute the distance differences for each non-reference anchor with respect to the reference one.

*Delta(reference, 1300000020, t1) = (0.192337772 - 0.192337772) * 299792458 = 0 m*

*Delta(90000a05a, 1300000020, t1) = (0.192337772 - 0.192337773) * 299792458 = -0.299792458 m*

*Delta(90000a05b, 1300000020, t1) = (0.192337772 - 0.192337772) * 299792458 = 0 m*

*Delta(90000a05e, 1300000020, t1) = (0.192337772 - 0.192337773) * 299792458 = -0.299792458 m*

Finally, it is possible to generate all available equations of the non-linear system of equations.

-0.299792458 = sqrt(X1300000020,t1^2 + Y1300000020,t1^2) - sqrt((X1300000020,t1-6.3308356)^2 + Y1300000020,t1^2)

0 = sqrt(X1300000020,t1^2 + Y1300000020,t1^2) - sqrt((X1300000020,t1-6.989999001)^2 + (Y1300000020,t1+5.28999995)^2)

-0.299792458 = sqrt(X1300000020,t1^2 + Y1300000020,t1^2) - sqrt((X1300000020,t1+0.250000299)^2 + (Y1300000020,t1+4.91999999)^2)

As we are interested in two coordinates, only two equations are needed to compute a solution. For instance, in case we solve a system using the second and third equations, the result is shown in Figure 46. It is also possible to use the additional available information to increase the precision of the computed tag position, which may be influenced by external factors.

Input interpretation

solve

$$0 = \sqrt{X^2 + Y^2} - \sqrt{(-6.99 + X)^2 + (5.29 + Y)^2}$$

$$-0.299792 = \sqrt{X^2 + Y^2} - \sqrt{(0.25 + X)^2 + (4.92 + Y)^2}$$

Result

$X = 3.69763$ and $Y = -2.37725$

Figure 46 - Solution of a generated non-linear system of equations.

Figure 47 reports the plot of the computed position on a chart.

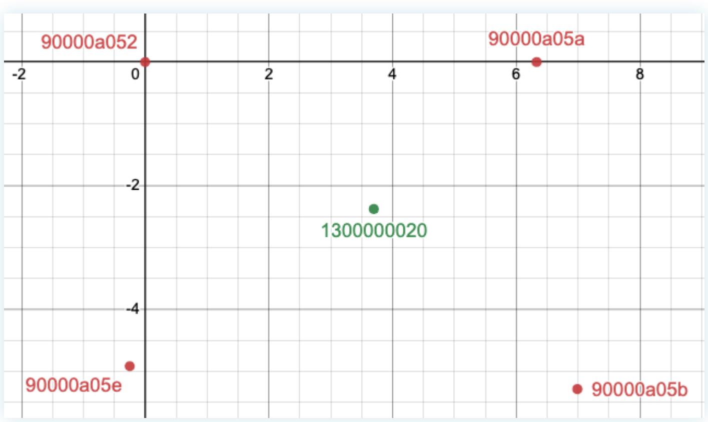


Figure 47 - Plot of the computed position of the target tag.

Figure 48 summarizes the entire procedure that is necessary for a passive eavesdropping attack.

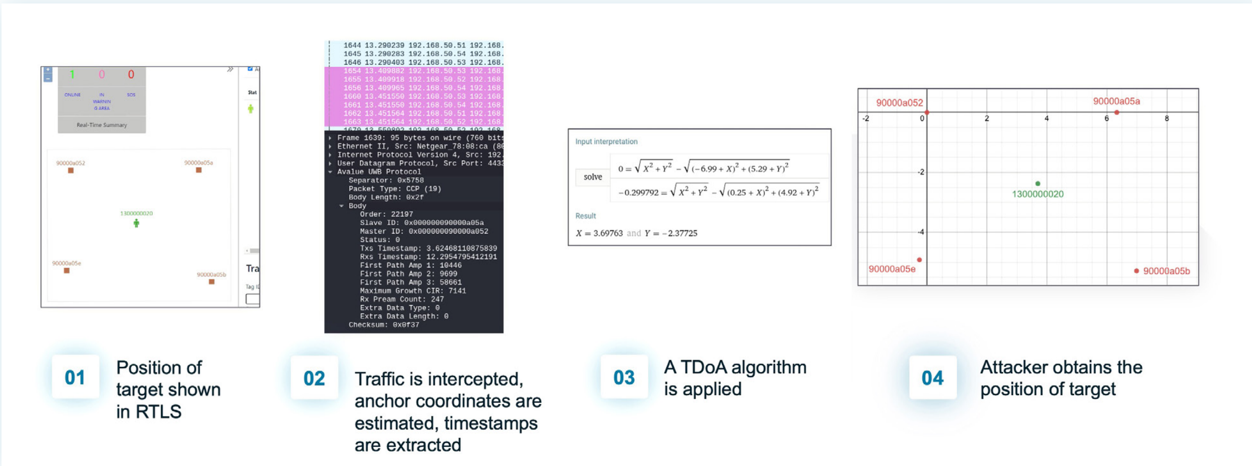


Figure 48 - Passive eavesdropping attack summary.

2.5.3 Active Traffic Manipulation Attacks

With the ARP spoofing attack detailed in section 2.5.1, and using the same algorithm described in the previous section, an attacker is able to see all the traffic among the server and the anchors and reconstruct the position of arbitrary tags, the only constraint being that they must have a foothold on the same subnet.

The logical question that arises at this point is: can an attacker leverage the acquired position to also perform active traffic manipulation attacks? There are many use cases and reasons that may induce an attacker to investigate this possibility. An example may be the desire to tamper with geofencing rules. In RTLS, geofencing rules can be configured, among other things, for access control purposes (an alert is triggered if a certain tag enters a restricted area), or anti-theft purposes (an alert is triggered if a certain tag leaves a defined area)[25] (Figure 49). If an attacker is able to alter the position of a tag by modifying the positioning packet related to that tag, it may become possible to enter restricted zones or steal valuable items without the operators being able to detect that a malicious activity is ongoing. Other examples will be described in the next sections.

[25] **"Geofencing Technology and Applications",** Sewio.

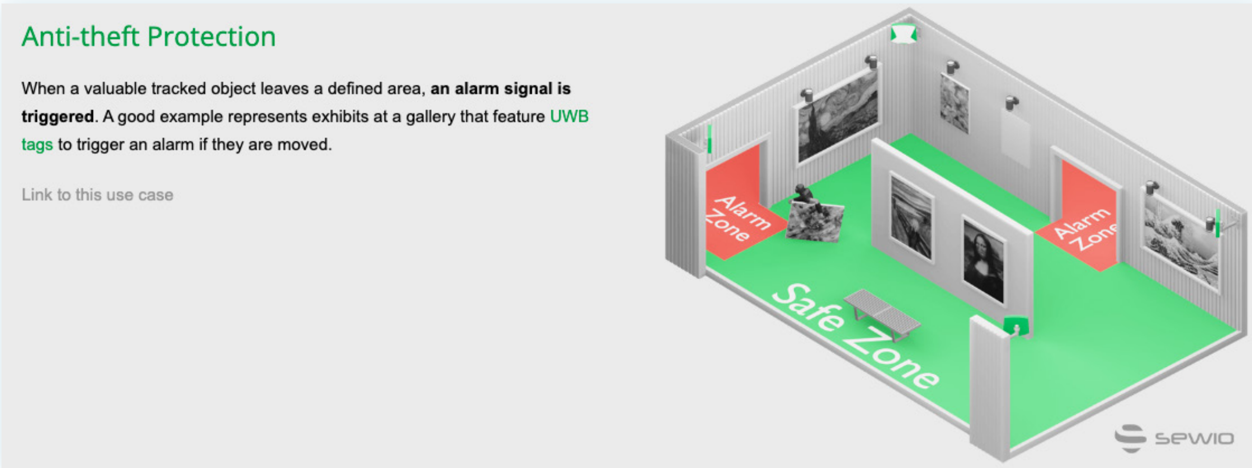


Figure 49 - Anti-theft protection in RTLS.

To accomplish such an attack, three subtasks need to be accomplished:

- Target Reconnaissance
- Active Traffic Filtering
- Packet Information Manipulation

Target Reconnaissance: In an active traffic manipulation attack, the goal of an adversary is to alter the position of a certain tag to make it appear at a desired coordinate instead of the real one.

To successfully deceive an operator into believing a certain tag is positioned at a given coordinate, it is important that the tag movements on the screen appear as natural as possible for the target. Therefore, it is crucial for an attacker to monitor the target position over a relevant time frame (e.g., one week, one month, or whatever is appropriate for the chosen target) and study their normal routine, to make the attack as believable as possible. For instance, if the target of an attack is a tag tracking the motion of a human being, faking its position in a stuttered way with harsh, sudden movements would warn an operator and make them think that, at the very least, a malfunction of some extent is occurring.

This reconnaissance phase can be accomplished by simply re-applying the same algorithm described in the previous chapters. Target profiling statistics (e.g., normal routine paths, average speed, minimum/maximum accelerations, or other relevant data) can be automatically generated, in order to finely tune the attack parameters and increase the chances of a successful attack.

Active Traffic Filtering: Another major difference with respect to a passive eavesdropping attack is that, in an active traffic manipulation attack, it is important to keep the network traffic "as-is" aside from the set of packets that are related with the target position. Notably, the resulting behavior that needs to be achieved is the following:

- if the packet is a synchronization packet, it must be automatically forwarded to the destination (as the alteration of synchronization packets would cause the modification of the positions of all tags monitored by the RTLS, not only the target ones);
- if the packet is a positioning packet, verification must be completed to determine if it is related to the target tag. If so, its timestamp must be modified (and the checksum updated). If not, it must be forwarded unaltered to the destination.

To this extent, many techniques are available. This work leveraged NFQUEUE, a flexible userspace packet handler provided by Iptables. The key idea behind it is to save the spoofed packets into a temporary queue, parse them one by one, determine if they need to be altered or not, then process them accordingly. To do so, a firewall rule was set, to forward the incoming packets to the queue:

```
Iptables –D FORWARD –p {UDP/TCP} -sport {port} -j
NFQUEUE –queue-num n
```

With this command, the firewall is configured to redirect all incoming packets from a specified port to the NFQUEUE number *n*. Then, from the attacking script, it is possible to bind the receiving of each packet to a function that parses them and properly invokes the manipulation routines.

Packet Information Manipulation: The final step of the attack is the manipulation of the information included in the packet. In most scenarios, this translates to altering the timestamps and updating the packet integrity fields (RTLS may transmit additional information for specific use cases, e.g., tag battery level, the press of a button on the tag, etc.).

Altering the timestamps is simply a matter of inverting all the equations described in section 2.2.2. If in a passive eavesdropping attack the positioning timestamps are known and the tag coordinates are unknown, then in an active traffic manipulation attack the tag coordinates are known (those will be the target coordinates that an attacker wants to fake for a target tag) and the positioning timestamps are unknown. Starting from equation 8, an attacker can simply execute all algorithm steps in reverse order and, eventually, obtain the positioning timestamps to include in the modified packets for a certain algorithm iteration.

Having finalized the packet content, all an attacker needs to do is run the integrity check algorithm used by the target RTLS to generate the packet checksum, then send the modified packets to the target RTLS server. Figure 50 summarizes the procedure necessary for this process.

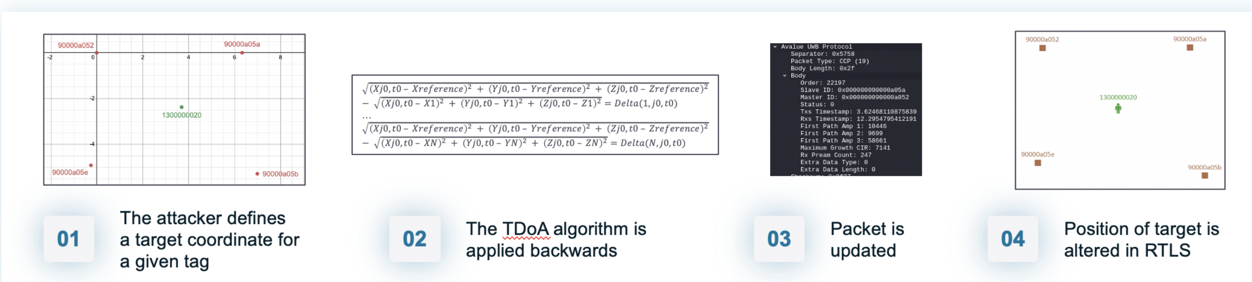


Figure 50 - Packet information manipulation summary.

2.6 Attacks Against Real-world Use Cases

In this section, we show how an adversary can practically leverage the primitives that we described in the previous chapter to perform attacks against common real-world use cases for RTLS.

2.6.1 Locating and Targeting People/Assets

As described in section 2.1.1, UWB RTLS can be used in real-world facilities to keep track of the position of people or assets: in factories, UWB RTLS help the management system to locate and rescue any employees in case of emergency; in hospitals, they are used to track patients' positions and quickly provide medical assistance in case of sudden, serious medical symptoms; in generic buildings, they can monitor the position of valuable items; etc.

One of the first attacks that an adversary may attempt against a real-world RTLS is to passively eavesdrop on the network traffic with the aim of reconstructing all tag positions and, thus, the position of the related people or assets.

There may be a variety of reasons behind this desire:

- An attacker wants to gain knowledge on the habits of a target person, with the aim of stalking them and/or causing them harm;
- An attacker wants to locate the position of a valuable item, with the aim of stealing it.

To programmatically perform such an attack against a real-world RTLS, all an attacker needs to do is develop an attack script that first performs a MitM attack against the RTLS backhaul network as described in section 2.5.1, then runs one of the TDoA algorithms available in literature, such as the one presented in section 2.2.

If the attacker does not have prior knowledge of the anchor positions, they will need to account for the preliminary anchor coordinate estimation phase (as described in section 2.4) to obtain the anchor positions. The attack script needs to keep track of all synchronization and positioning timestamps seen on the network, then continuously update the tag positions shown on the map.

In our tests, it was possible to develop a Python script capable of enacting the steps described above against both Sewio and Avalue RTLS. Figure 51 depicts an execution of the script against the Avalue RTLS. As can be noticed, the script managed to compute the same position of the target tag as shown by the RTLS web interface. Additionally, no warnings or abnormal behavior that could have alerted an operator were noticeable.

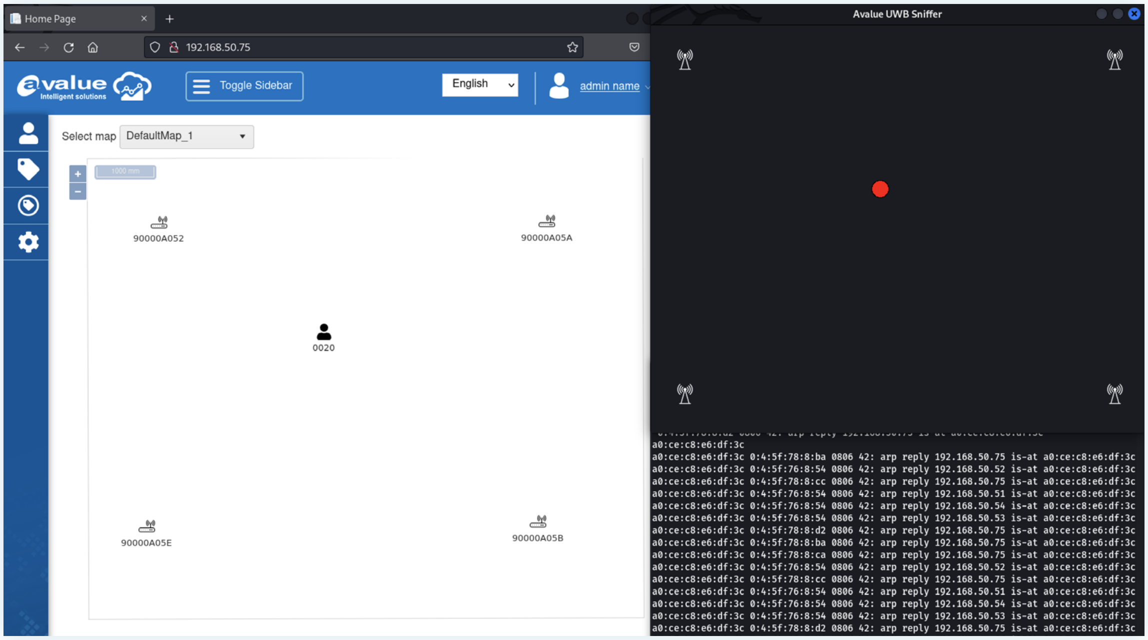


Figure 51 - Passive eavesdropping attack against Avalue RTLS.

To prevent malicious actors from immediately reusing our results and performing attacks against real-world RTLS, the code of the script has not been released with this white paper.

2.6.2 Geofencing

One of the most crucial functionalities of RTLS from a safety perspective is represented by geofencing. RTLS that offer geofencing functionalities allow the configuration of spatial-aware rules that are triggered whenever a tag enters or exits a specific area. For instance, in factories and hospitals, geofencing rules can be set up to trigger the stoppage of hazardous machines in case a human being walks near them.

Geofencing rules can also be employed for non-safety related purposes. As an example, in generic buildings, geofencing rules can act as an anti-theft solution that triggers an alert whenever a valuable item leaves a certain zone.

From the use cases described above, it is clear that geofencing rules represent a critical functionality of an RTLS and, thus, a valuable target for an adversary. Some examples of attacks that can be enacted follow:

- By modifying the position of tags and placing them inside areas monitored by geofencing rules, an attacker can cause the stoppage of entire production lines;
- By placing a tag outside an area monitored by geofencing rules, an attacker can cause machinery to start moving when a person is in proximity, potentially causing harm;
- By making a tag appear in a steady position, an attacker can steal an item tracked by the tag without raising any alerts.

All aforementioned attack scenarios require an attacker to actively manipulate the network traffic, in order to change the position of a tag at will. To programmatically perform this attack against a real-world RTLS, an attacker needs to develop an attack script that first performs a MitM attack against the RTLS backhaul network as described in section 2.5.1, then performs all steps described in chapter 2.5.3, i.e., target reconnaissance, active traffic filtering, and packet information manipulation. Again, if the attacker does not have prior knowledge of the anchors' positions, they will need to account for the preliminary anchor coordinates estimation phase (section 2.4.) The attack script needs to keep track of all synchronization timestamps seen on the network, generate positioning timestamps accordingly to mimic a natural target tag movement, then send the modified packets to the target RTLS server.

To verify the possibility of actually interfering with a realistic safety geofencing rule, we configured a Mitsubishi R08SFCPU controller that was part of one of our lab demos to listen to the geofencing alerts raised by the Sewio RTLS and, on the basis of the alerts received, control an electric motor (Figure 52) according to the following rules:

- If the controller receives an alert of a tag entering the electric motor geofenced zone, it stops the motor, and turns on the safety warning light ON;
- If the controller receives an alert of a tag exiting the electric motor geofenced zone, it restarts the motor, and turns off the safety warning light OFF;

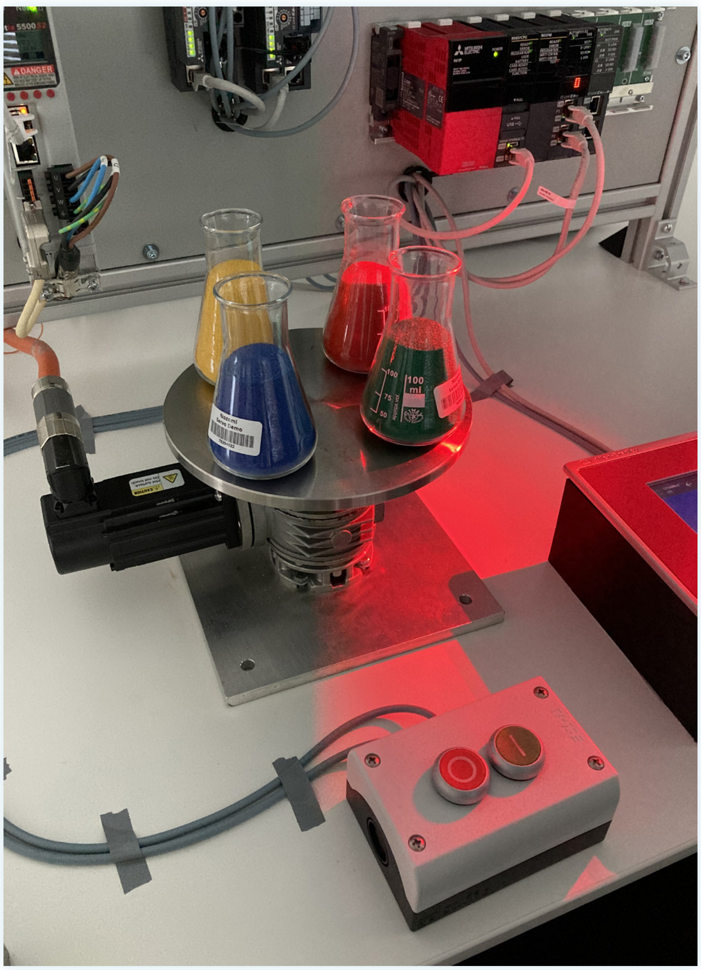


Figure 52 - Geofencing demo setup.

In our tests, it was possible to develop a Python script capable of enacting the steps described above against both the Sewio and Avalue RTLS. Figure 53 and Figure 54 depict an execution of the script against the Sewio RTLS integrated with the previously described electric motor. Notably, the script managed to:

- cause the motor to arbitrarily stop, by modifying the position of tags and placing them inside the geofenced zone;
- cause the motor to restart even when tags (people) were in proximity to it, by modifying the position of tags and placing them outside the geofenced zone.

Again, no warnings or abnormal behavior that could have alerted an operator were noticeable, as the injected positions mimicked the natural movements of the target tag, which were previously studied in the reconnaissance phase.

To prevent malicious actors from immediately reusing our results and performing attacks against real-world RTLS, the code of the script is not being released with this white paper.

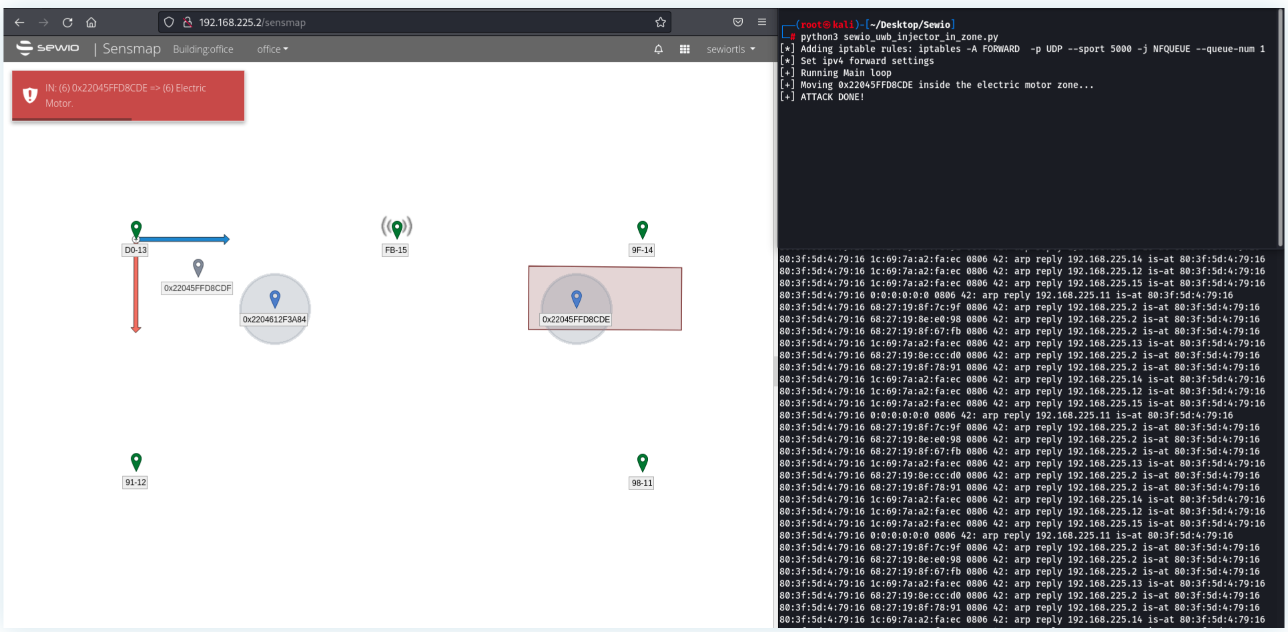


Figure 53 - Active traffic manipulation attack against Sewio RTLS – Placing tag inside the geofenced zone.

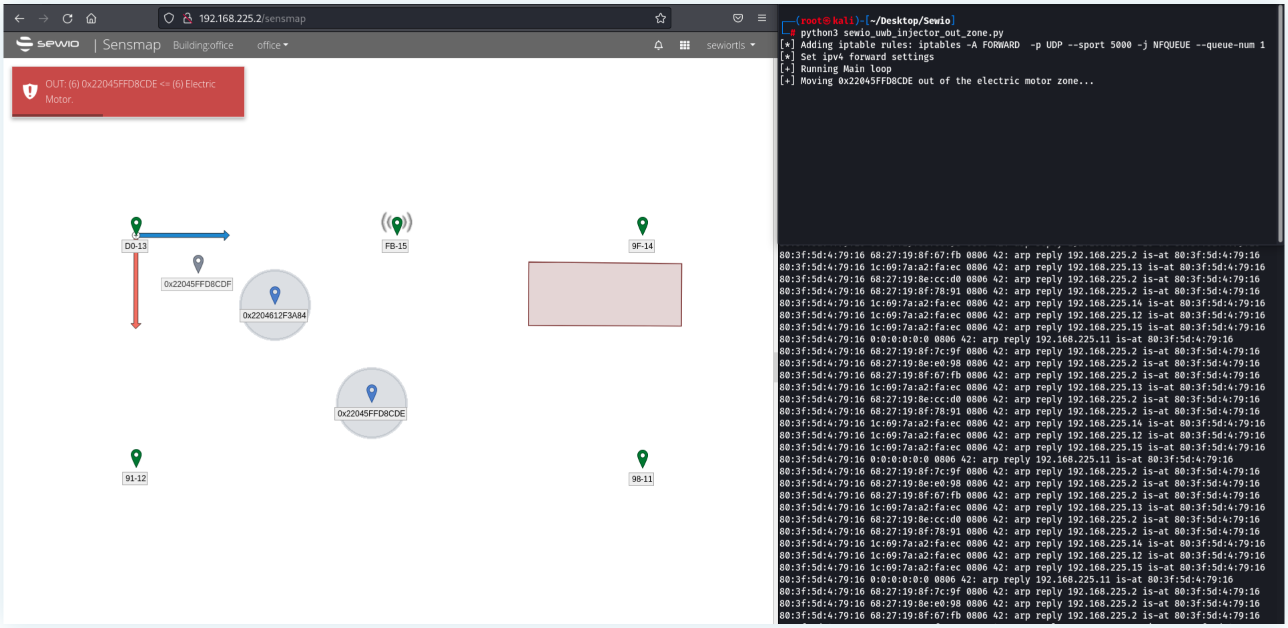


Figure 54 - Active traffic manipulation attack against Sewio RTLS – Placing tag outside the geofenced zone.

2.6.3 Contact Tracing

Given that RTLS offer the possibility of tracking a person's movements inside a wide variety of facilities, and the more and more widespread requirements imposed by the COVID-19 pandemic of having a way to keep track of close contacts among people, vendors have started offering contact tracing functionalities. By considering factors such as contact duration, presence of barriers, or even the usage of shared tools for more complex solutions, an RTLS can estimate the risk that a person has contracted a certain disease given a set of positive individuals.

As with the previously described use cases, such a feature can become a target for malicious actors. Some examples of attacks that can be enacted follow:

- An attacker can induce false contacts among people, aiming to cause a certain group of victims to be erroneously considered at high risk of being positive, thus being forced to preventively quarantine;
- An attacker can prevent the detection of true contacts among people, with the aim of facilitating the spread of COVID-19 or other illnesses throughout a company, resulting in downtime due to mass employee quarantine that could have long-term effects (especially for immune-compromised personnel).

All aforementioned attack scenarios require an attacker to actively manipulate the network traffic, in order to change the position of a tag at will. This can be done exactly as described in the previous chapter. Of the two analyzed solutions, only the Sewio RTLS offered a contact tracing functionality via tag zones: by defining a validity radius for each tag, a contact is recorded if two tag circles experience a contact event.

In our tests, it was possible to develop a Python script capable of interfering with the contact tracing functionality offered by Sewio, as shown in Figures 55 and 56. Notably, the script managed to:

- Generate false contact events among arbitrary tags;
- Prevent true contacts among tags being detected.

Again, no warnings or abnormal behavior that may have alerted an operator were noticeable, as the injected positions mimicked the natural movements of the target tag, which were previously studied in the reconnaissance phase.

To prevent malicious actors from immediately reusing our results and performing attacks against real-world RTLS, the code of the script is not being released with this whitepaper.

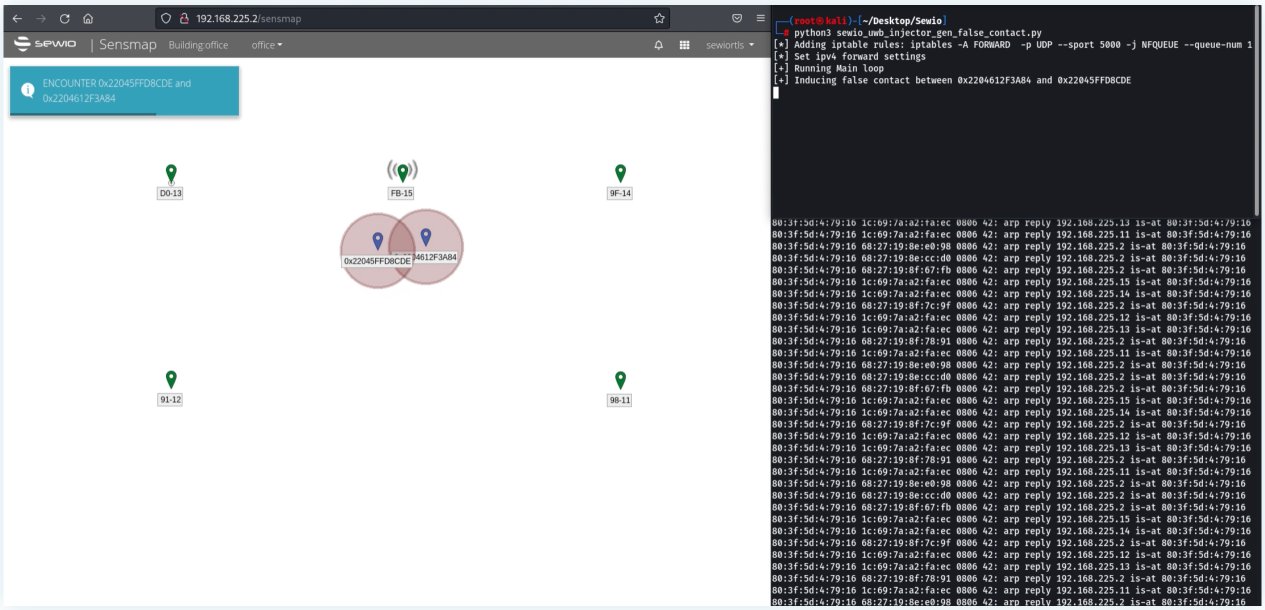


Figure 55 - Active traffic manipulation attack against Sewio RTLS – Generating a false contact.

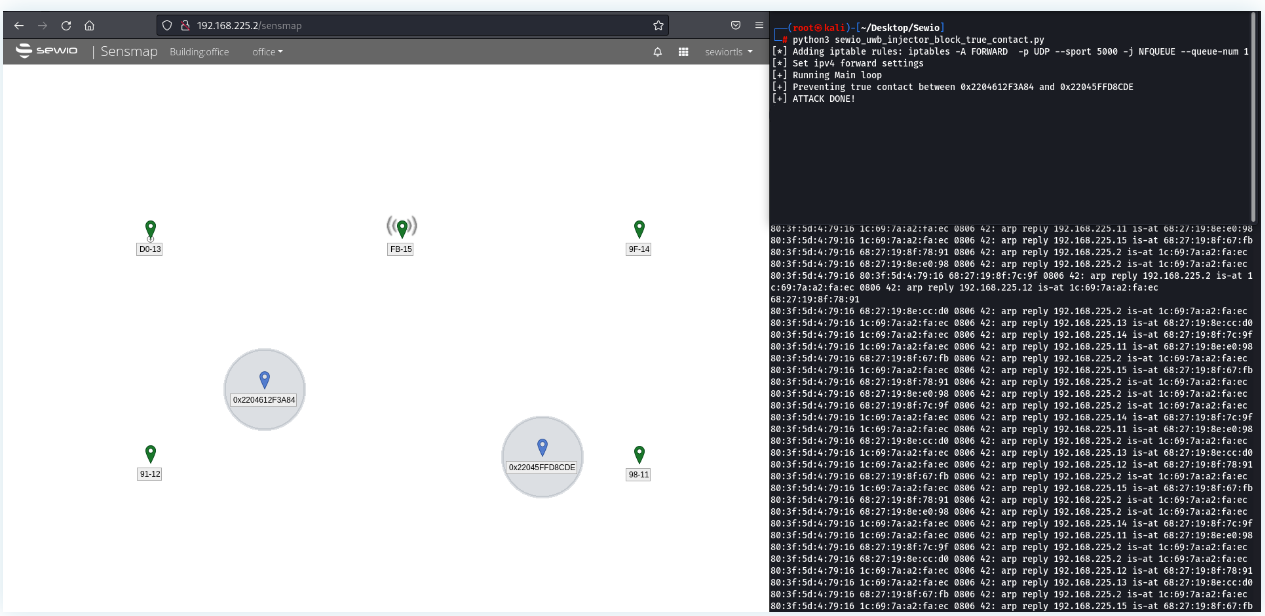


Figure 56 - Active traffic manipulation attack against Sewio RTLS – Preventing a true contact.

3. Remediations

Among the possible remediations that an end user can implement, the most effective ones are segregation and firewall rules, application of intrusion detection systems (IDS), and traffic encryption. This section presents each of these possible remediations and evaluates the advantages and challenges of each option.

3.1 Segregation and Firewall Rules

As mentioned before, one of the most stringent requirements for the success of an attack is that an adversary has a foothold on the same subnet where the UWB RTLS is installed. Thus, a first mitigation to these attacks is to move the entire UWB RTLS backhaul network to a segregated network, and secure the access to the network both physically and logically, with the aim of preventing unauthorized actors from gaining access to it. This is now mandated by some RTLS vendors[26], as shown in Figure 57.

Physical access

- Restrict physical access to the device to qualified personnel.
- Disable unused physical interfaces of the device. Unused interfaces could be used to gain access to the operating site.

Software - Safety functions

- Only use protocols that are required to operate the device.
- Restrict access to the device with a firewall or rules in an ACL (Access Control List).
- Using VLANs gives you good protection against DoS attacks. Check whether this is practicable.
- Activate the access logging function (external). Use the central logging function to record changes and access.
- Configure a SysLog server to save all logs to a central location.

SIMATIC RTLS4030G
Operating Instructions, 04/2021, C79000-G8976-C515-06

Figure 57 - Siemens RTLS4030G operating instructions.

[26] **"Simatic RTLS Localization System Simatic RTLS4030G Operating Instructions,"** Siemens, April 2021.

While traditional solutions taken from the IT domain such as VLANs, IEEE 802.1X, or firewall rules can be greatly effective for this purpose, some challenging aspects need to be considered.

The RTLS server is a critical component to protect in an UWB RTLS backhaul network, as, by design, it needs to listen to all incoming communications from the anchor network and, at the same time, be accessible by the operators monitoring it.

The majority of the time, the RTLS server will be hosted on a bare metal or virtual computer with two network interfaces, one attached to the backhaul network, the other attached to a management network. While designing the firewall rules, it must be kept in mind that some RTLS may be configured to expose core network services on all interfaces by default. For instance, we noticed that both Sewio RTLS and Avalue RTLS, as shown in Figure 58 and Figure 59, exposed the services responsible for the processing of packets from the anchors on all interfaces. Although no meaningful attacks can be done without the synchronization and positioning timestamps, there might be room for Denial-of-Service (DoS) attacks from the management network if these services are not filtered via specific firewall rules. By executing a DoS attack, an adversary may temporarily halt the continuous update of tag positions, potentially impeding geofencing rules or contact tracing features from correctly operating for a short amount of time.

```
root@lab-iot-sewio-wks:~# netstat -apn
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      779/systemd-resolve
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1656/sshd
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      730/cupsd
tcp        0      0 127.0.0.1:6010          0.0.0.0:*               LISTEN      5116/sshd: seviortl
tcp        0      0 0.0.0.0:5000            0.0.0.0:*               LISTEN      3404/node
tcp        0      0 0.0.0.0:5001            0.0.0.0:*               LISTEN      3404/node
```

Figure 58 - Sewio RTLS listening ports.

Listening Ports

Image	PID	Address	Port	Protocol	Firewall Status
tomcat8.exe	2648	IPv6 unspecified	6000	UDP	Allowed, not restricted
tomcat8.exe	2648	IPv4 unspecified	6000	UDP	Allowed, not restricted
tomcat8.exe	2648	IPv4 loopback	8065	TCP	Allowed, not restricted
tomcat8.exe	2624	IPv6 unspecified	8080	TCP	Allowed, not restricted
tomcat8.exe	2624	IPv4 unspecified	8080	TCP	Allowed, not restricted
tomcat8.exe	2624	IPv6 unspecified	8080	UDP	Allowed, not restricted
tomcat8.exe	2624	IPv4 unspecified	8080	UDP	Allowed, not restricted
tomcat8.exe	2624	IPv4 loopback	8085	TCP	Allowed, not restricted
tomcat8.exe	2648	IPv6 unspecified	8686	TCP	Allowed, not restricted
tomcat8.exe	2648	IPv4 unspecified	8686	TCP	Allowed, not restricted

Figure 59 - Avalue RTLS listening ports.

Finally, it must be considered that, even if security measures are adopted to enforce network segregation, the lack of transport protection measures in the protocol design of RTLS remains. If an attacker were able to physically attach to the wired network (for instance, by cutting a wire and connecting a device in the middle) or managed to obtain the wireless password (for instance, by cracking a WPA2-PSK handshake), there would be nothing preventing an attacker from successfully accomplishing the attacks described in this white paper, even in the presence of access control measures. Thus, a continuous monitoring of the physical status of the wired network must be enforced, or periodic wireless password rotation and other wireless security best practices must be strictly followed.

3.2 Intrusion Detection Systems

Another fundamental requirement for the success of an attack is that an adversary must first perform a MitM to obtain all necessary synchronization and positioning timestamps, which are not normally sent via broadcast packets. Consequently, another possible mitigation is to install an IDS in the UWB RTLS backhaul network. By monitoring for signatures such as new ARP frames or new links between nodes (that an attacker is bound to generate), an IDS can quickly detect an ongoing MitM, as shown in Figure 60.

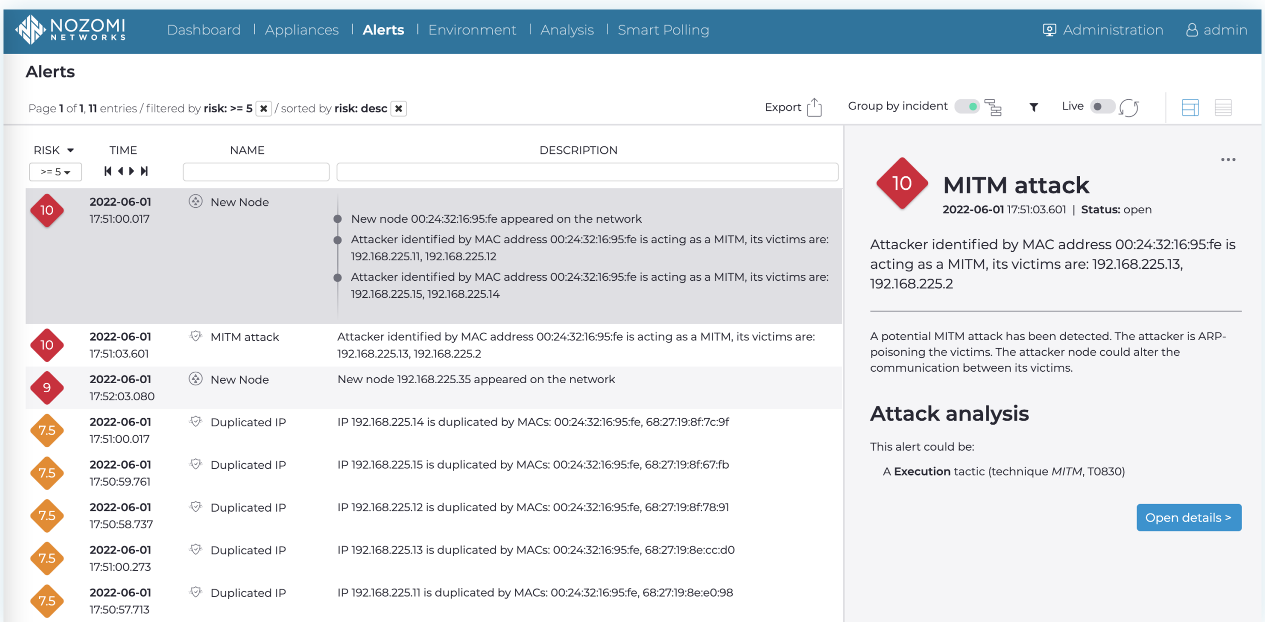


Figure 60 - Nozomi Networks Guardian detecting the MitM attack against Sewio RTLS.

Like the previous mitigation, it is still inherently vulnerable to an attacker that performs a physical MitM: if they were able to physically attach to the wired network, or managed to obtain the wireless password, an intrusion detection system would be unable to discriminate between an attack taking place and legitimate traffic, even when featuring application layer inspection functionalities. As a matter of fact, even in the case of an active traffic manipulation attack, the tampered traffic would be indistinguishable from legitimately generated traffic if crafted by an attacker with prior knowledge of a given target's normal movements.

3.3 Traffic Encryption

The most effective mitigation that an asset owner can apply is to add a traffic encryption layer on top of the existing communications, to prevent even a physical MitM from successfully tampering with the systems. This option was actually tested on the Avalue RTLS, as it was the only solution that allowed administrative access to the RTLS server as well as the anchors.

As a proof of concept, with the goal of using already available tools, we attempted to encrypt all traffic generated by the anchors by encapsulating it through an SSH tunnel. First, a classic SSH tunnel was created, by connecting each anchor to an SSH service exposed by the RTLS server and setting up a local port forwarding service. Then, an instance of code was run on the server and all anchors, to create the UDP to TCP (and vice versa) bridges that are necessary to tunnel the network traffic generated by the anchors (that is UDP) inside SSH (that supports only TCP) and then back to UDP for the server processing. Finally, all anchors were configured to send all traffic to the internal service exposed by the code instances running on them. A result of the experiment is depicted in Figure 61.

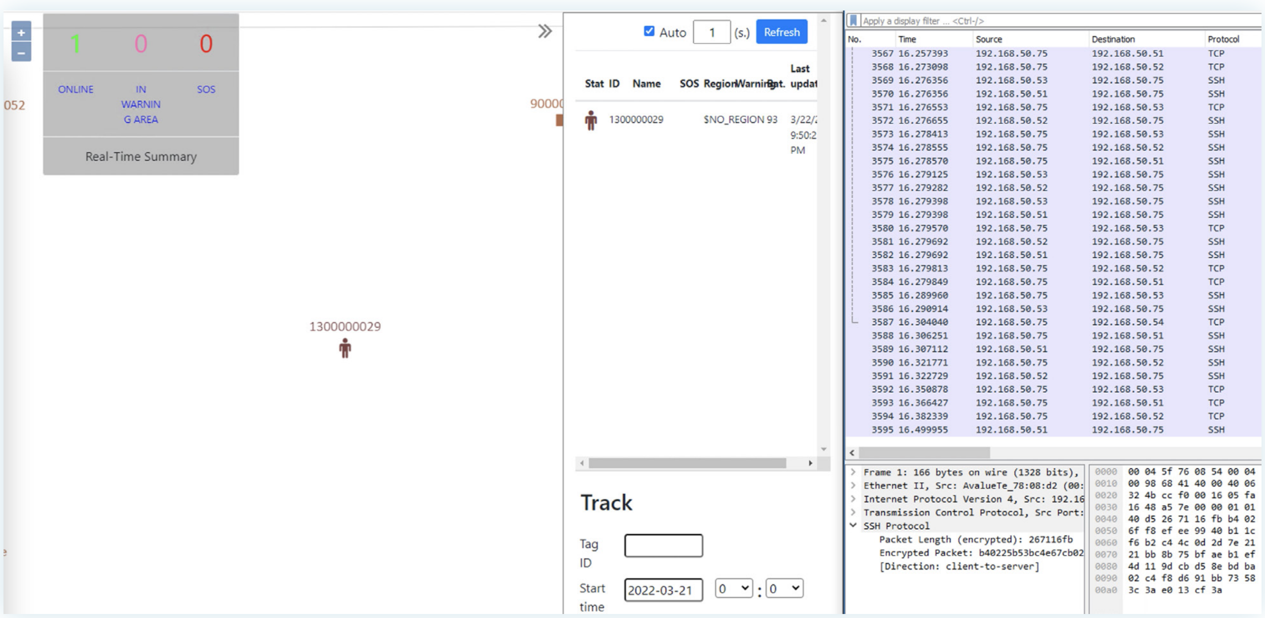


Figure 61 - SSH tunnel PoC on Avalue RTLS.

As can be noticed by the evidence, the PoC was successful: all traffic generated by the anchors to the server was tunneled inside SSH and, consequently, protected from any MitM attacks, while at the same time preserving the basic RTLS functionality. However, even in this case, some challenges need to be solved.

First, the extra effort produced by the SSH tunnelling increased the load of the anchors and, eventually, led to a perceived delay of the RTLS server with respect to the real-time tag positions. To counteract this effect, it was necessary to increase the period of synchronization packets from the default 150 ms to 500 ms to decrease the number of communications generated by the anchors, at the expense of a reduced position accuracy. This might be a problem for some asset owners that need real-time, precise tag positioning.

Finally, the possibility of enabling such encryption layers on top of the already existing technologies depends entirely on the accessibility of the RTLS server and anchors from the vendor. If either the server or the anchors do not allow administrative access (as was the case with the Sewio RTLS, whose anchors do not expose any SSH access), enacting this solution either requires an extensive work of firmware modification to enable it, or is simply not viable.

4. Summary and Key Takeaways

4.1 Summary

UWB RTLS are becoming increasingly common as both businesses and individuals see the benefits to utilizing this technology to increase efficiency, productivity, and location accuracy of people and assets. Although the IEEE 802.15.4z amendment was aimed at increasing the security of UWB, the design of securing critical protocols was “out of scope” and left completely up to vendors who may or may not know how to implement this type of security at the device level.

After conducting research on two popular UWB RTLS on the market, Nozomi Networks Labs discovered zero-day vulnerabilities that threat actors can exploit to disrupt and manipulate various environments. Our assessment of these protocols in two popular UWB RTLS revealed security flaws that could allow an attacker to gain full access to sensitive data exchanged over-the-air and impact people's safety. In this paper we provided mitigations that individuals as well as asset owners can implement right away to help mitigate these risks.

We believe that this work is important because of the potential impact on people's lives if threat actors were able to exploit the vulnerabilities identified in our research. We hope that by releasing this information publicly we will help raise awareness about how important it is for companies who operate critical infrastructure systems such as airports, hospitals, power plants and manufacturing facilities to ensure the security of their networks to reduce the susceptibility of being compromised by a malicious attacker.

4.1 Key Takeaways

Weak security requirements in critical software can lead to **safety issues** that cannot be ignored

There are attack surfaces out there that no one is looking at, but **they have significant consequences if compromised**

Exploiting secondary communications in UWB RTLS can be **challenging, but it is doable**

Authors

Andrea Palanca

Security Researcher, Nozomi Networks

Andrea Palanca is an information security engineer, with a strong background in penetration testing of web applications and network devices. He is the first author of “A Stealth, Selective, Link-Layer Denial-of-Service Attack Against Automotive Networks”, which unveiled a novel way to exploit a design-level vulnerability affecting the CAN bus standard.

Luca Cremona

Security Researcher, Nozomi Networks

Luca Cremona received the PhD title in 2021 from the Computer Science department of Politecnico di Milano. The main research fields of his PhD include RTL design for secure and power-aware SoCs, with a particular emphasis on Side Channel Attack countermeasures. He's currently a security researcher at Nozomi Networks, working on reverse engineering and hardware hacking topics.

Roya Gordon

Security Research Evangelist, Nozomi Networks

Roya Gordon provides insights and solutions for OT and IoT security. Prior to Nozomi Networks, Roya worked as the Cyber Threat Intelligence subject matter expert (SME) for OT and Critical Infrastructure clients at Accenture, a Control Systems Cybersecurity Analyst at Idaho National Laboratory (INL), and as an Intelligence Specialist in the United States Navy. She holds a Masters in Global affairs with a focus on cyberwarfare from Florida International University (FIU).

Nozomi Networks

The Leading Solution for OT and IoT Security and Visibility

Nozomi Networks accelerates digital transformation by protecting the world's critical infrastructure, industrial and government organizations from cyber threats. Our solution delivers exceptional network and asset visibility, threat detection, and insights for OT and IoT environments. Customers rely on us to minimize risk and complexity while maximizing operational resilience.

NN-WP-UWB-8.5x11-001

nozominetworks.com