BLE GATT Fuzzing

Baptiste Boyer
bboyer@quarkslab.com

Quarkslab

Abstract. Bluetooth Low Energy (BLE) is a widely adopted wireless communication technology used by billions of devices in various applications. These applications range from IoT domain to more sensitive devices such as medical ones.

BLE has been subject to a lot of research so far, but only a few of them targeted specification corner cases which require high-level manipulation of the GATT layer. This paper proposes to explore this dense and sometimes unclear specification, and to show how we have designed attack scenarios on the ATT and GATT layers. We propose a fuzzing approach to reach our goals since it provides an easy and efficient way to identify potential vulnerabilities and weaknesses regarding the various BLE stacks implementations.

The adopted methodology is presented first, outlining the steps followed during this work. Then, a comprehensive description of the ATT and GATT layers is made. After that, our attack scenarios are detailed, covering their elaboration from the specification and their implementation. Our test bench setup is then introduced, including the tools and services used, along with an overview of the various BLE stacks fuzzed. The final section identifies the various non-conformities, bugs, and vulnerabilities found in relation to the specification across the different stacks, which were reported to the appropriate entities.

1 Introduction

1.1 Context

Bluetooth Low Energy (BLE) is a widely adopted wireless communication protocol in embedded systems, known for its energy-efficient design. Its applications range from the Internet of Things domain to more sensitive devices, including medical equipment. However, the proliferation of BLE-enabled devices and the diverse implementations of the BLE stack have given rise to numerous security issues. The lack of standardization in the BLE stack implementation contributes to the existence of these vulnerabilities across various devices.

Indeed, the BLE specification lacks certain details, leaving room for interpretation by developers. This absence of explicit guidance results in unspecified behaviors, corner cases, and, consequently, potential non-conformities, bugs, and vulnerabilities.

This project served as a six-month internship at Quarkslab with two main objectives. The first goal was to assess and test an internal framework designed to facilitate the interaction between a computer and various wireless devices by developing a fuzzer. The second was to dig into a less-explored aspect of BLE security: the Attribute Protocol (ATT) and Generic Attribute Protocol (GATT) layers.

1.2 Why fuzz the GATT layer?

The relevance of the GATT layer in terms of security arises from its proximity to the application layer. A poorly implemented GATT layer directly impacts the application, making it crucial to ensure robustness at this level. Additionally, the GATT layer has not received sufficient attention in security research compared to the dedicated Security Manager Protocol (SMP). The majority of BLE security research has focused on SMP, neglecting potential vulnerabilities and issues within the GATT layer. The lack of specificity in the BLE specification leaves room for developers to overlook certain possibilities, potentially leading to inadequate stack implementations.

1.3 State of the Art

Numerous security studies have explored the vulnerabilities of BLE, with notable examples such as KNOB [16], BlueMirror [18], and SweynTooth [19]. What stands out is a shared characteristic among these renowned research endeavors—they generally do not delve into the specifics of the GATT layer. Efforts to implement specific GATT fuzzers can be found on GitHub, such as projects like "*blefuzz*" [20] and "*ble-fuzzer*" [15]. However, these implementations are limited to the basic ATT Write Request method and rely on the *gatttool* [1] command provided by *BlueZ* [2], the open-source Bluetooth stack implementation for Linux. It's important to note that *gatttool* offers only fundamental methods related to GATT operations and lacks support for utilizing the full range of ATT methods.

In SweynTooth [19], Matheus Garbelini proposed a BLE fuzzer based on the modelization of the protocol state machine. Unlike our focus on the ATT and GATT layers, his fuzzer targeted multiple layers, including the Security Manager Protocol, L2CAP, Link Layer, and ATT. Despite SweynTooth discovering 17 vulnerabilities, only one pertained to the ATT layer, **underscoring the need for dedicated ATT and GATT layer assessments**.

1.4 Contribution

This paper introduces our own approach to ATT and GATT fuzzing, based on attack scenarios constructed through an in-depth analysis of the specification, with a specific emphasis on the ATT and GATT layers. It provides an overview of the methodology used, the implementation details of a custom fuzzer, the chosen target devices and corresponding BLE stacks, and the results derived from the experimentation and analysis.

2 Methodology

Due to the constraints imposed by the internship timeline, a clear and efficient methodology was crucial. As shown by Figure 1, the process started with an in-depth analysis of the specification, with a focus on the ATT and GATT layers. This analysis uncovered corner cases and unspecified behaviors, forming the basis for our attack scenarios. The scenarios were implemented using the *WHAD* [17] framework. Determining the BLE stacks for testing was another critical step. Following this, various GATT servers were developed, and the fuzzer was launched against these servers, producing valuable results including non-conformities, bugs, and vulnerabilities, which were duly reported.

Here is the adopted methodology:

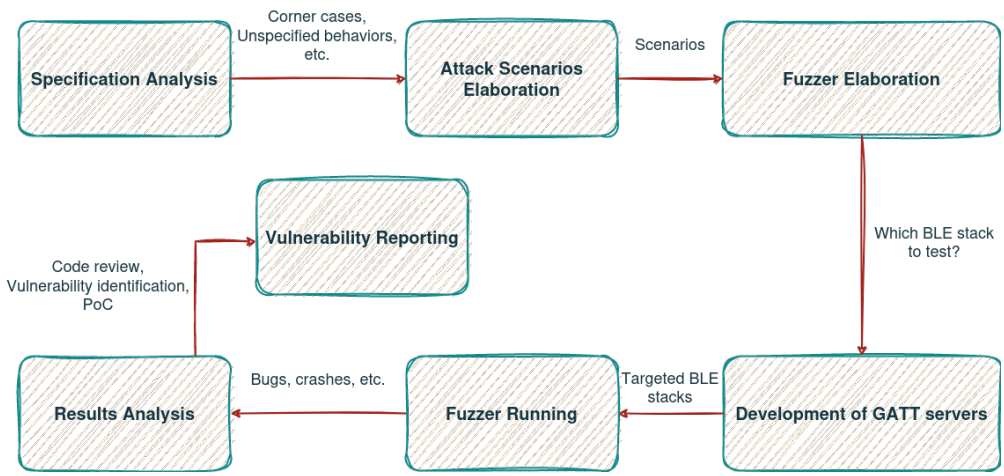


Fig. 1. Adopted methodology

3 BLE Analysis

Bluetooth Low Energy (BLE) is a wireless communication protocol designed for short-range communication between devices, emphasizing low power consumption and optimized data transfer for various applications. It operates within the 2.4 GHz band and is particularly suitable for battery-powered devices, such as smartphones, wearables, and Internet of Things devices. It employs a central-peripheral architecture, where a central device can communicate with multiple peripheral devices. The communication involves small packets of data, known as Protocol Data Units (PDUs), exchanged at regular intervals.

Key features of BLE include its quick connection setup, low latency, and support for periodic data transmissions. It uses a GATT (Generic Attribute Profile) structure to organize data into attributes, making it flexible for a wide range of applications. BLE also incorporates security measures, including encryption and authentication, to ensure secure data transmission.

3.1 BLE stack

The BLE protocol is built upon multiple layers, extending from the radio layer to the application layer.

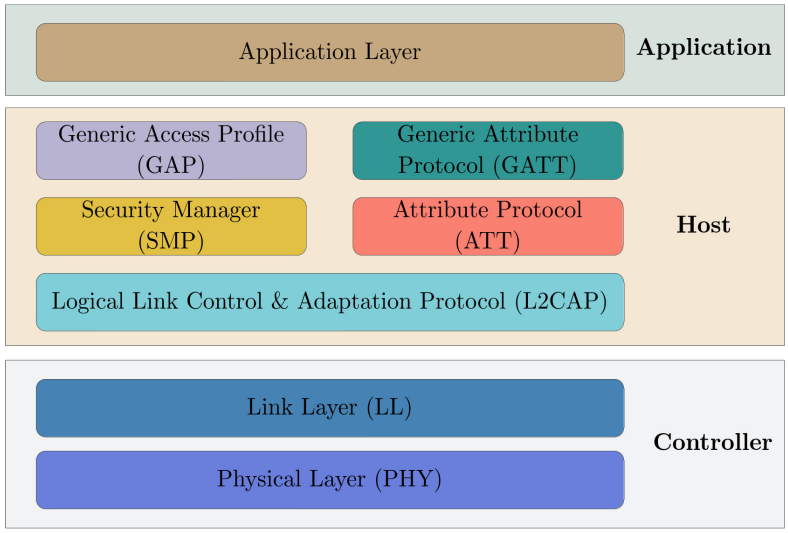


Fig. 2. BLE stack

Before we begin with a brief description of the various layers, it's essential to introduce the key components: the Host and the Controller, which are the two fundamental entities in the BLE framework.

The Host and Controller represent distinct entities with specific roles in the communication process. The Host is typically associated with the device's primary processing unit, such as a computer or smartphone. It oversees higher layers of the BLE protocol stack, managing connections, configuring parameters, and handling application-specific communication needs. In contrast, the Controller operates at a lower layer, responsible for radio frequency communication aspects. Implemented in specialized hardware, like a Bluetooth chip, the

Controller executes tasks such as frequency hopping, modulation, and power control, ensuring reliable and efficient radio communication. Together, the Host and Controller collaborate to enable seamless and efficient communication in BLE-enabled devices. To facilitate their collaboration, they utilize the Host Controller Interface (HCI), an interface layer that enables communication between the software-based Host and the hardware-based Controller. This interface allows the Host to send commands and receive events related to the lower layers, ensuring a cohesive and interoperable Bluetooth communication system.

Starting from the bottom layer of the stack, we will progressively ascend through the various layers toward the uppermost layer.

- The *Physical Layer* (PHY) is the lowest layer and deals with the transmission and reception of raw binary data over the radio link. It manages aspects such as modulation, frequency hopping, and power control for reliable communication.
- Located above the Physical Layer, the *Link Layer* (LL) oversees the establishment, maintenance, and termination of connections. It handles packet transmission, error detection, and all the link control procedures, playing a crucial role in ensuring the integrity of the communication link.
- The *Logical Link Control & Adaptation Protocol* (L2CAP) layer resides above the HCI and is responsible for segmentation and reassembly of data packets. It supports multiple protocols and adapts data for transmission, enabling efficient communication between the Host and Controller.
- The *Security Manager Protocol* (SMP) layer focuses on establishing and managing security between devices. It handles tasks such as pairing, key distribution, and encryption, ensuring the confidentiality and integrity of data exchanged between connected devices. The SMP layer is crucial for securing BLE communications, especially in scenarios where sensitive information is involved.
- The *Generic Access Profile* (GAP) layer handles general aspects of device discovery, connection establishment, and security. It defines roles such as "peripheral" and "central" and manages procedures for devices to interact seamlessly.
- At the top of the stack, the *Application Layer* (App) deals with user-specific functionalities. It includes application-specific profiles and services that enable the implementation of diverse applications, such as health monitors, smart home devices, and more.

We have gained a comprehensive understanding of the BLE stack, yet it's evident that we have not addressed the Attribute Protocol (ATT) and the GATT – crucial components that come into play after the connection process.

3.2 ATT

Data Represention

"*The Attribute Protocol defines two roles; a server role and a client role. It allows a server to expose a set of attributes to a client that are accessible using the Attribute Protocol.*" [Spec v5.4 Vol.3 Part.F.2]

Based on this definition given in the protocol overview, we know that the BLE protocol has a client-server architecture and that the server exposes a collection of attributes to the client. But what is an attribute?

An attribute is a data representation format composed of four fields. The Attribute data structure has the following format:

Attribute field	**Size (bytes)**	**Description**
Handle	2	A unique identifier assigned to each attribute, allowing devices to reference and interact with them.
UUID	2 or 16	An identifier that uniquely identifies a specific attribute type, defining its purpose and characteristics.
Value	Variable length	The actual data associated with an attribute, representing the information communicated between devices.
Permissions	Implementation specific	Access rights and restrictions governing how an attribute can be read, written, or otherwise interacted with by devices, ensuring security and control over the data exchange.

Table 1. Attribute Format

The attribute handle, a non-zero value, serves as a reference for the attribute. In a BLE server, all attributes are organized in its database based on an ascending order of attribute handle values. While it is not obligatory for consecutive attributes to follow sequential integer handle values, it is allowed for gaps to exist between attribute handle values. However, it is essential that the handle values maintain an ascending order.

The universally unique identifier (UUID) serves to identify each attribute type, ensuring its uniqueness across all space and time. Given that many devices share common functionalities, a specific range of UUID values has been reserved for predefined values. These values are specified in the Bluetooth Assigned

Numbers [3] document. Each of these values exposes a set of actions and data for common use cases.

The attribute values can have either a fixed length or a variable length. In the case of variable length attribute values, a singular attribute value is permitted to be transmitted in a PDU. If the value exceeds a certain length, it has the option to be divided across multiple PDUs.

The attribute permissions define whether a resource is eligible for reading and/or writing, along with the necessary security level. Various security combinations are permissible. For instance, an attribute might demand no permissions for reading, yet the client may need to authenticate in order to modify the resource.

Exchange Methods

The Attribute Protocol outlines methods for both reading and writing attributes, totaling six distinct methods, each associated with a specific PDU. In the context of the ATT protocol, a PDU represents the packet that is exchanged with the lower layer – specifically, the L2CAP layer. This packet is then encapsulated for transmission over the physical link or sent to the upper layers.

The six methods and their corresponding PDU types are categorized as follows: *command, request, response, notification, indication,* and *confirmation.* Additionally, certain Attribute Protocol PDUs may include an Authentication Signature, providing a means of authenticating the PDU's originator without necessitating encryption. The combination of the method and the signed bit is referred to as the *opcode.*

Type	**Purpose**	**Suffix**
Commands	Sent to a server by a client that does not invoke a response.	CMD
Requests	Sent to a server by a client that invokes a response.	REQ
Responses	Sent to a client by a server in response to a request.	RSQ
Notifications	Sent to a client by a server that does not invoke a confirmation. Unsollicited PDUs.	NTF
Indications	Sent to a client by a server that invokes a confirmation. Unsollicited PDUs.	IND
Confirmations	Sent to a server by a client to confirm receipt of an indication.	CFM

Table 2. Attribute PDUs

Attribute PDUs have the following format:

Name	**Size (bytes)**	**Description**
Attribute Opcode	1	The attribute PDU operation code bit 7: Authentication Signature Flag bit 6: Command Flag bits 5-0: Method
Attribute parameters	0 to (ATT_MTU - X)	The attribute PDU parameters X = 1 if Authentication Signature Flag of the Attribute Opcode is 0 X = 13 if Authentication Signature Flag of the Attribute Opcode is 1
Autentication Signature	0 or 12	Optional authentication signature for the Attribute Opcode and Attribute Parameters

Table 3. Format of attribute PDU

The Attribute Opcode is composed of three fields, the Authentication Signature Flag, the Command Flag, and the Method. The Method is a 6-bit value that determines the format and meaning of the Attribute Parameters.

If the Authentication Signature Flag of the Attribute Opcode is set to one, the Authentication Signature value shall be appended to the end of the attribute PDU, and X is 13. If the Authentication Signature Flag of the Attribute Opcode is set to zero, the Authentication Signature value shall not be appended, and X is 1.

If the packet is received as anticipated without issues, the protocol specifies the corresponding actions that the receiving end should take upon success. In the event of an error, the protocol includes provisions for an error response, indicating the source of the error. The exchange flows for these scenarios are succinctly summarized in the table 4 below.

Attribute PDU Method	**Successful Response PDU**	**Error Response Allowed**
Exchange MTU Request	Exchange MTU Response	Yes
Find Information Request	Find Information Response	Yes
Find By Type Value Request	Find By Type Value Response	Yes
Read By Type Request	Read By Type Response	Yes
Read Request	Read Response	Yes
Read Blob Request	Read Blob Response	Yes
Read Multiple Request	Read Multiple Response	Yes
Read by Group Type Request	Read By Group Type Response	Yes
Read Multiple Variable Request	Read Multiple Variable Response	Yes
Write Request	Write Response	Yes
Write Command	*none*	No
Signed Write Command	*none*	No
Prepare Write Request	Prepare Write Response	Yes
Execute Write Request	Execute Write Response	Yes
Handle Value Notification	*none*	No
Handle Value Indication	Handle Value Confirmation	No
Multiple Handle Value Notification	*none*	No

Table 4. Attribute request and response summary

Let's elaborate on the behavior of the Prepare Write Request, as most of our findings are associated with this PDU.

The Prepare Write Request PDU is vital when dealing with the need to write a long Attribute value, i.e., an attribute value where the size of ATT exceeds (ATT_MTU - 1). The default Maximum Transmission Unit (MTU) in BLE communications is 23 bytes. This PDU is utilized to request the server to prepare the write of a value for a specified attribute. The server responds with a Prepare Write Response to confirm the correct reception of the value. Multiple Prepare Write Request PDUs may be sent to transmit the complete Attribute value. Once all the PDUs have been transmitted, the client sends an Execute Write Request to instruct the server to write the value.

The format of the Prepare Write Request is as follows:

Parameter	**Size (bytes)**	**Description**
Attribute Opcode	1	0x16 = ATT_PREPARE_WRITE_REQ PDU
Attribute Handle	2	The handle of the attribute to be written
Attribute Offset	2	The offset of the first octet to be written
Part Attribute Value	0 to (ATT_MTU-5)	The value of the attribute to be written

Table 5. Format of ATT_PREPARE_WRITE_REQ PDU

A practical use of the Prepare Write and Execute Write requests could be illustrated as follows:

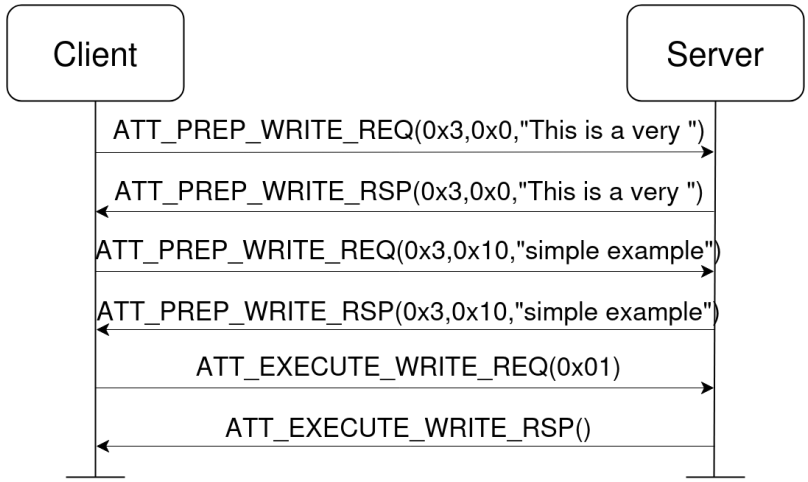


Fig. 3. Write long Attribute Values example

In brief, the ATT protocol is vital for managing data representation (attributes) within a BLE server database and dictating transaction activities, whether successful or not. This forms the foundation for packet fragmentation and encapsulation in lower stack protocols. Simultaneously, these principles become the building blocks utilized by the GATT protocol to establish a higher level of abstraction for streamlined data access in BLE communication.

3.3 GATT

The Generic Attribute Profile (GATT) establishes a service framework built upon the Attribute Protocol. This framework defines procedures and formats for services and their characteristics are defined. These procedures encompass various operations such as discovering, reading, writing, notifying, and indicating characteristics. Additionally, GATT outlines procedures for configuring the broadcast of characteristics. It also defines some standard profiles with associated services and characteristics [3].

The GATT Profile establishes the format for exchanging profile data, defining fundamental elements such as services and characteristics within the structure. These elements are encapsulated by Attributes, which serve as containers for carrying profile data in the Attribute Protocol.

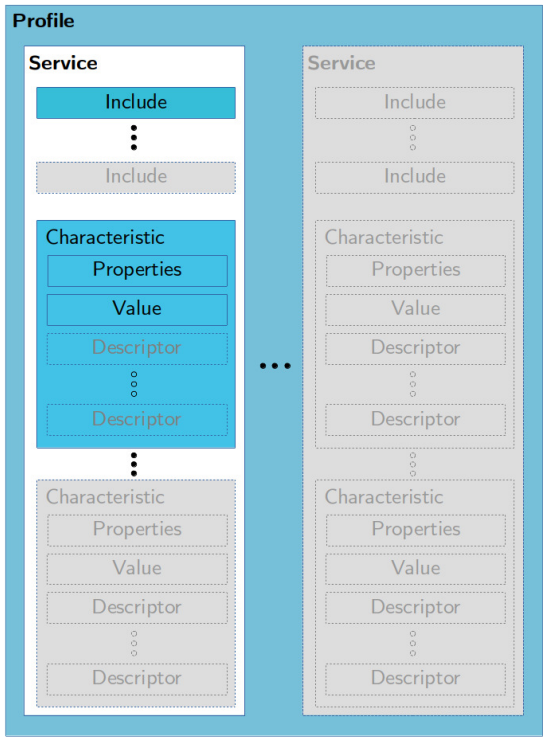


Fig. 4. GATT Profile hierarchy

At the highest level of the hierarchy is a profile, which is formed by one or more services. A service is comprised of characteristics or may include other services. Each characteristic holds a value and may include optional information related to that value. The profile data, encompassing services, characteristics, and their components, such as values and descriptors, is stored in Attributes on the server. This hierarchical arrangement provides a structured and organized approach for managing and representing data within the BLE framework.

GATT services define specific functions supported by the server and act as a mechanism for organized grouping. They can be referenced and included by other services, promoting modularity and reusability across different profiles. There are two main types of services: *Primary Services*, exposing the device's primary functionality and discoverable through the *Primary Service Discovery* procedure, and Secondary Services, designed for additional auxiliary functionality. Each service may have one or more characteristics, representing specific features. Officially adopted BLE services have a 16-bit UUID length, while custom services can have a 128-bit UUID length.

A characteristic is an attribute associated with a service, encompassing properties and configuration details on how the value is accessed, as well as information on its display or representation. The characteristic is precisely defined by its characteristic definition, which includes a characteristic declaration, characteristic properties, and an associated value. Additionally, the definition may incorporate descriptors that provide details about the value or allow for server configuration in relation to the characteristic.

Attribute Handle	**Attribute Types**	**Attribute Value**			**Attribute Permissions**
0xNNNN	0x2803 - UUID for "Characteristic"	Characteristic Properties	Characteristic Value Attribute Handle	Characteristc UUID	Read Only, No Authentication, No Authorization

Table 6. Characteristic declaration

Attribute Value	Size	Description
Characteristic Properties	1 bytes	Bit field of characteristic properties
Characteristic Value Handle	2 bytes	Handle of the Attribute containing the value of this characteristic
Characteristic UUID	2 or 16 bytes	16-bit Bluetooth UUID or 128-bit UUID for Characteristic Value

Table 7. Attribute Value field

The GATT profile encompasses 11 features, each linked to procedures and sub-procedures as described in table 8.

Feature	Sub-Procedure
Server Configuration	Exchange MTU
Primary Service Discovery	Discover All Primary Services Discover Primary Services By Service UUID
Relationship Discovery	Find Included Services
Characteristic Discovery	Discover All Characteristic of a Service Discover Characteristic By UUID
Characteristic Descriptor Discovery	Discover All Characteristic Descriptors
Characteristic Value Read	Read Characteristic Value Read Using Characteristic UUID Read Long Characteristic Values Read Multiple Characteristic Values Read Multiple Variable Length Characteristic Values
Characteristic Value Write	Write Without Response Signed Write Without Response Write Characteristic Value Write Long Characteristic Values Characteristic Value Reliable Writes
Characteristic Value Notification	Single Notifications Multiple Variable Length Notifications
Characteristic Value Indication	Indications
Characteristic Descriptor Value Read	Read Characteristic Descriptors Read Long Characteristic Descriptors
Characteristic Descriptor Value Write	Write Characteristic Descriptors Write Long Characteristic Descriptors

Table 8. GATT features mapping to procedures

These defined processes outline the utilization of the Attribute Protocol to achieve the respective feature within the GATT profile. The figure 3 serves as an example for the Write Long Characteristic Values sub-procedure.

In brief, GATT provides a standardized structure and set of operations for efficient and consistent communication between devices utilizing the Attribute Protocol within the BLE framework.

4 Attack scenarios

The attack scenarios took shape following an intensive two-week immersion into the intricacies of the BLE specification [4]. The exhaustive study encompassed the detailed specifications of the ATT and GATT layers, comprising approximately 140 pages. The overarching goal was to pinpoint corner cases and inaccuracies within the specification, providing a foundation for the development of precise attack scenarios. In total, 9 scenarios were defined.

Developing scenarios allowed us to simulate real-world use cases, considering the sequence of interactions and potential states the BLE stack could be in. Indeed, the BLE stack operates as a state machine, where the system transitions between various states based on events and actions. This approach provided a more comprehensive evaluation of the system's behavior, helping uncover issues that might not be apparent when focusing solely on individual PDUs.

The choice to develop a new fuzzer rather than using an existing one was deliberate and aligned with the specific objectives of the internship. The primary motivation was to thoroughly test and evaluate the *WHAD* framework. Existing fuzzers might not have been tailored to the unique features and requirements of *WHAD*, and developing a new fuzzer allowed for a more targeted assessment of the framework's capabilities. *WHAD*'s ability to handle and process received PDUs during fuzzing demonstrates its dynamic testing capabilities.

4.1 Scenarios

Scenario #1

Observation: "*Many Attribute Protocol PDUs use a sequential request-response protocol. Once a client sends a request to a server, that client shall send no other request to the same server on the same ATT bearer until a response PDU has been received.*" [Spec Vol.3 Part.F 3.3.2]

Scenario: With this observation in mind, our approach involved injecting an ATT PDU into a REQUEST/RESPONSE sequence. The objective was to observe how the stack responds when tasked with processing an additional ATT PDU while already engaged in the handling of another. This scenario aimed to provide insights into the stack's behavior under concurrent ATT PDU processing, contributing to a comprehensive understanding of its robustness and potential vulnerabilities.

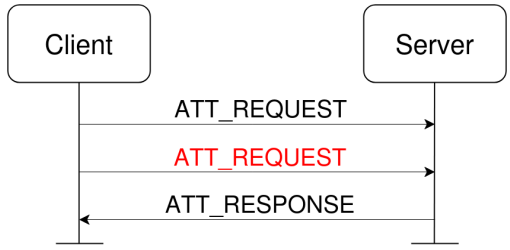


Fig. 5. Scenario 1

Scenario #2

Observation: "*Many Attribute Protocol PDUs use a sequential request-response protocol.*" [Spec Vol.3 Part.F 3.3.2]

Scenario: Analyzing the provided table 4 it became evident that the majority of sent REQUESTs awaited a corresponding RESPONSE. This led us to explore the scenario of sending an unsolicited PDU, specifically a RESPONSE without a prior REQUEST, to the server. The objective was to investigate how the server would handle such unexpected and unconventional PDU interactions, providing valuable insights into the stack's robustness and its ability to manage unexpected protocol deviations.

Fig. 6. Scenario 2

Scenario #3

Observation: "*A client may send more than one ATT_PREPARE_WRITE_REQ PDU to a server, which will queue and send a response for each handle value pair. A server may limit the number of prepared writes that it can queue. A higher layer specification should define this limit.*" [Spec Vol.3 Part.F 3.4.6.1]

Scenario: Adhering to the observation that it is the developer's responsibility to check the number of prepared writes that the server can queue, we

conducted an experiment by sending numerous prepare write requests to observe the stack's behavior. The objective was to assess how the stack manages a potentially high number of prepared writes, considering that an excessive volume may consume resources and potentially result in undefined behavior.

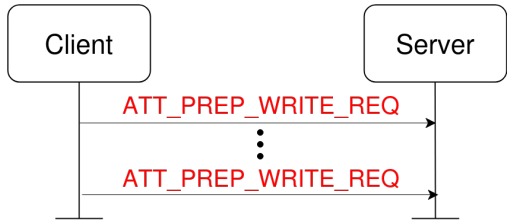


Fig. 7. Scenario 3

Scenario #4

Observation: Various ATT primitives used for reading or finding information include parameters for a starting handle and an ending handle. According to the specification, the server is expected to respond with data in a specific order, stating that "*pairs shall be returned in ascending order of attribute handles.*" [Spec Vol.3 Part.F 3.4.3.1]

Scenario: In line with this assumption, we deliberately modify the order in which the handle-data pairs are returned. This intentional alteration allows us to observe and analyze the stack's behavior when presented with a non-standard sequence of attribute handles and corresponding data during read or find information operations.

Fig. 8. Scenario 4

Scenario #5

Observation: Certain ATT PDUs come with specific fields that are expected to take precise values. This is particularly relevant in the context of Find Information Response, where specific values are anticipated *"The Format parameter can contain one of two possible values."* [Spec Vol.3 Part.F 3.4.3.2]

Name	Format	Description
Handle(s) and 16-bit Bluetooth UUID(s)	0x01	A list of 1 more handles with their 16-bit Bluetooth UUIDs
Handle(s) and 128-bit UUID(s)	0x02	A list of 1 more handles with their 128-bit Bluetooth UUIDs

Table 9. Format field values

Scenario: Observing that the format field is designed to accept only values within a specified range, we deliberately send a Find Information Response with a non-standard format field. Our hypothesis revolves around the assumption that the stack developers might implicitly trust the information sent by the server without thoroughly validating the content of the received PDU.

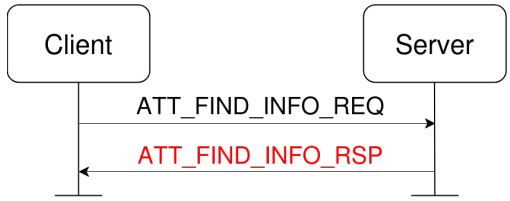


Fig. 9. Scenario 5

Scenario #6

Observation: "*For notifications, which do not have a response PDU, there is no flow control and a notification can be sent at any time. For commands, which do not have a response PDU, there is no flow control and a*

command can be sent at any time. Note: A server can be flooded with commands, and a higher layer specification can define how to prevent this from occurring." [Spec Vol.3 Part.F 3.3.2]

Scenario: Given that it is the developer's responsibility to determine how to prevent flooding, we intentionally send numerous PDUs that don't require a response, attempting to flood the server.

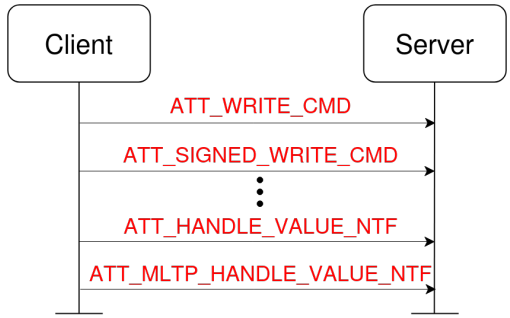


Fig. 10. Scenario 6

Scenario #7

Observation: The MTU is set to a default value, but we can modify this with an Exchange MTU Request.

Scenario: We want to see the behavior of the stack when receiving a PDU with its length greater than the set MTU.

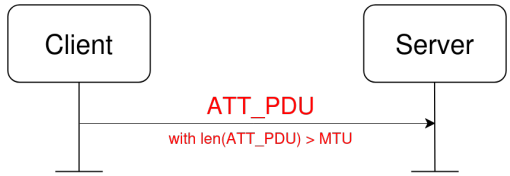


Fig. 11. Scenario 7

Scenario #8

Observation: Upon establishing a connection to the server, the client initiates the process of discovering all services, characteristics, and descriptors through the relevant procedures listed in 8.

Scenario: Based on this observation, we aimed to introduce inconsistencies in the GATT profile by having the server return incorrect data to the client during this discovery phase.

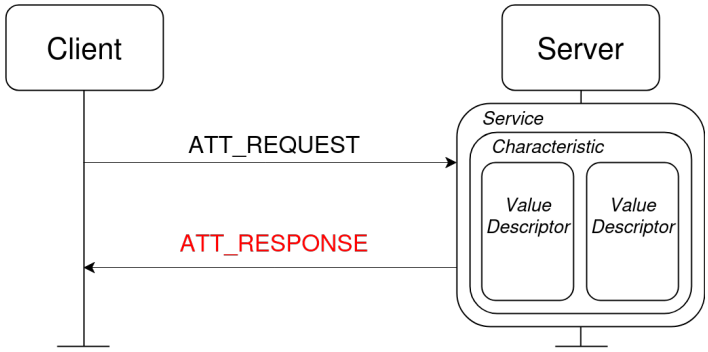


Fig. 12. Scenario 8

Scenario #9

Observation: The Secondary and Included services, being less commonly used in real-world devices, may have implementations that were less tested. Plus, "*If the client detects a circular reference or detects nested include declarations to a greater level than it expects, it should terminate or stop using the ATT bearer.*" [Spec Vol.3 Part.G 3.2]

Scenario: Implement Secondary and Included services to our GATT server and investigate their theoretical capabilities.

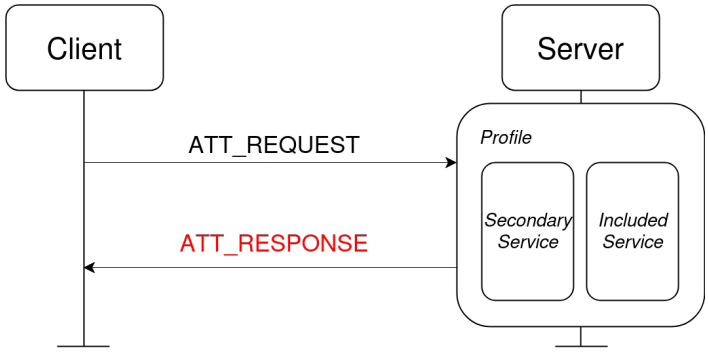


Fig. 13. Scenario 9

All the scenarios executed from the client side were also replicated with the server roles whenever applicable.

4.2 Fuzzer implementation

Host Side

The attack scenarios were implemented as fuzzing scripts in Python using the *WHAD* [17] framework.

While SweynTooth [19] and KNOB [16] leveraged the *InternalBlue* tool [5] for message crafting, we opted for *WHAD* as our chosen framework for its specific capabilities in the BLE context. This framework equips us with all the essential tools for testing the GATT layer, including transparent PDU logging and its own BLE stack implementation. It offers a simple method to create and send BLE packets, encompassing the entire spectrum of ATT possibilities. Additionally, it facilitates the seamless setup of a GATT server and enables straightforward modification of its behavior.

The fuzzer folder structure is illlustrated by Figure 14 below.

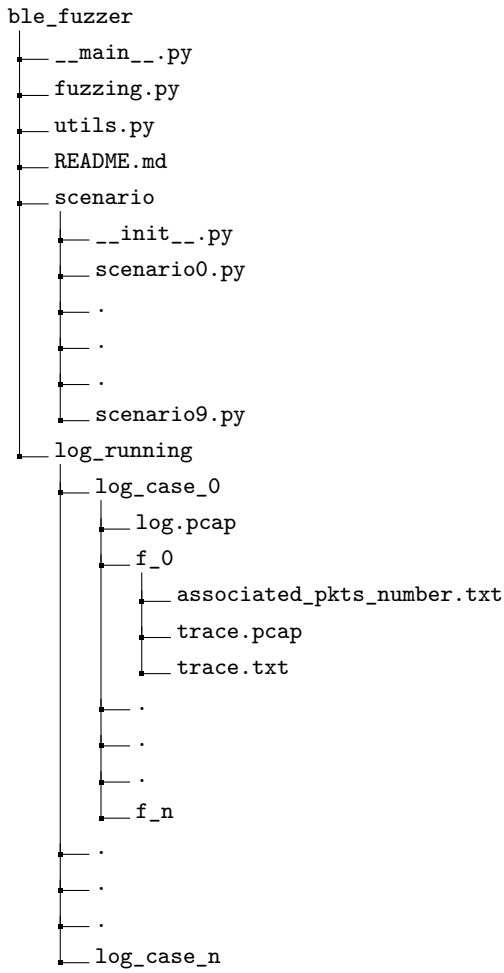


Fig. 14. Fuzzer folder structure

Several parameters are set through the CLI interface, the help option displays the following content:

```
Usage: __main__.py [OPTIONS] {client|server}

Bluetooth Low Energy GATT Fuzzer based on multiple scenario.

 Options
 --bt_addr          -bt   TEXT                  Bluetooth address of the device.
 --is_addr_random   -r                          Is the given Bluetooth address random. [default: False]
 --post_url         -u    TEXT                  Notify address to use. [default: https://ntfy.sh/test_ntfy_server]
 --interface        -i    TEXT                  Interface to use. [default: hci0]
 --gatt_handle      -g    INTEGER               The last GATT handle of the device. [default: 100]
 --scenario         -s    [0|1|2|3|4|5|6|7|8|9] The scenario to play. [default: 0]
 --none_cnt         -nc   INTEGER               The max unreceived responses before triggering an error. [default: 20]
 --prep_write_max   -pwm  INTEGER               Number of prepare write PDUs to send. [default: 100]
 --help             -h                          Show this message and exit.
```

Fig. 15. CLI options

The `fuzzing.py` file contains:

— *mutator()*: A function that crafts and returns a random ATT PDU.
— *mutate_fill_payload()*: A function that fills the random ATT PDU.
— *expected_response()*: A function that maps the request to the expected response.
— *check_expected_get()*: A function that compares the received PDU to the expected one.
— *att_dict*: A dictionary of all ATT PDUs.
— *att_request_dict*: A dictionary of requests ATT PDUs.

The fuzzer architecture was designed with certain key choices in mind. One notable decision involved the type of connection established during a fuzzing session. Instead of initiating a new connection for each test case, we opted for a single connection for the entire fuzzing session. This choice was driven by the belief that retaining previous states of the stack could potentially aid in triggering and identifying bugs. The alternative approach would have been to create a new connection for every two consecutive Requests, but the decision to maintain a persistent connection was considered more suitable for bug discovery.

Another significant choice in the fuzzer's architecture pertained to the selection of GATT handles. In the configuration of the fuzzer, the last handle present in the GATT server is provided, and subsequently, the crafted PDUs are populated with a handle chosen randomly from the range of 1 to the specified handle. This decision was intentional, allowing the fuzzer to send PDUs to handles that might not be associated with an Attribute. The randomness in handle selection aimed to explore a broader range of handles for potential bug

discovery, providing a more comprehensive evaluation of the GATT server's behavior.

Additionally, a main PCAP file for the fuzzing case, PCAP files for each transaction (Request/Response), and text files containing the exchanges are generated by the framework used. However, it would have been beneficial to have a PCAP file for the entire fuzzing session, as one fuzzing case may impact the next one. Moreover, we incorporated a real-time monitoring system using the *Ntfy* service. This feature enabled continuous monitoring and timely notifications during the fuzzer's execution, enhancing the overall visibility into the testing process.

Apart from the GATT handle field, we opted for a randomized approach to populate the PDUs. An example illustrating the Prepare Write Request PDU mentioned earlier is provided in 1. Additionally, it is worth noting that each fuzzing session focused solely on testing a single scenario.

Listing 1: Example of how are filled the PDUs

```
def mutate_fill_payload(att_proc, gatt_handle):
  """
  Fills and returns the ATT primitive
  """

  match att_proc:
    ...
    case scapy.layers.bluetooth.ATT_Prepare_Write_Request:

      _gatt_handle = random.choice(gatt_handle)
      _offset      = random.randint(0,65535)
      n            = random.randint(0,80)
      _data        = random.randbytes(n)

      return ATT_Prepare_Write_Request(gatt_handle = _gatt_handle,
                                       offset      = _offset,
                                       data        = _data) , 5 + n
```

Targeted device Side

We leveraged the provided GATT server example from each SDK, identifying valuable functions and adapting them to integrate our Protobuf messages. Additionally, we developed a C library that was included in all GATT server examples, providing essential functions for transmitting and receiving Protobuf messages over the UART port. With these Protobuf messages we were able to get registers state or to reset the device when a crash occured.

The library structure is as follows:

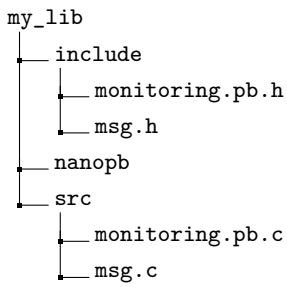


Fig. 16. my_lib folder structure

with:

— *monitoring.pb.h*: file generated by nanopb compiler to use our Protobuf messages,
— *msg.h*: our header file,
— *nanopb*: nanopb folder contains necessary files,
— *monitoring.pb.c*: file generated by nanopb compiler to use our Protobuf messages,
— *msg.c*: our implementation.

The `msg.c` file contains:

— *send_pb_message()*: main function to send the protobuf messages,
— *fill_msg()*: a function that fills protobuf messages,
— *encode()*: a function that encodes the protobuf messages,
— *send_tx()*: a function that sends the protobuf messages over UART,
— *rcvd_action()*: a function to receive the protobuf messages,
— *process_action()*: a function to process the received protobuf messages.

Figure 17 below gives an overview of the fuzzer.

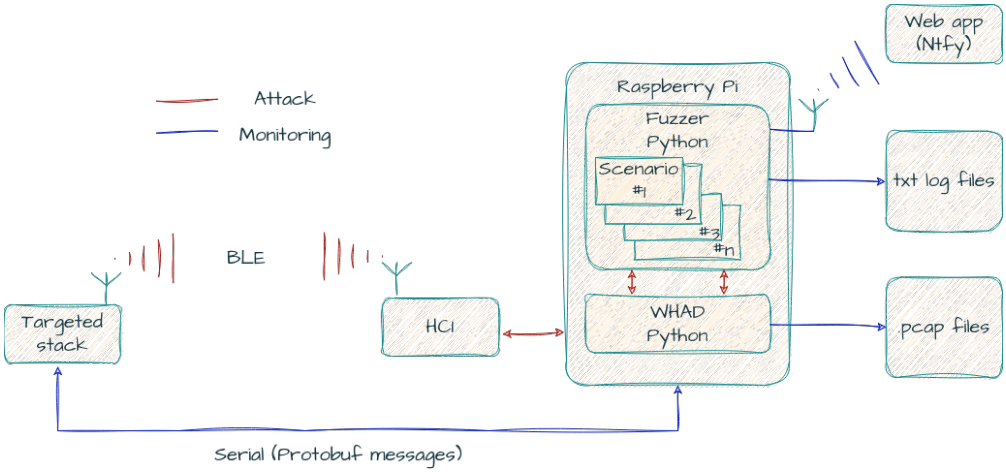


Fig. 17. Fuzzer overview

5 Test bench

In establishing an effective test bench for our fuzzer, we carefully considered various requirements to ensure optimal performance and usability. The key constraints and criteria included accessibility, Python compatibility, sufficient disk space for extensive log files, ample processing power, Bluetooth and WiFi connectivity, and the presence of USB ports for additional device connections.

After thorough consideration, we chose the *Raspberry Pi* [6] as our preferred platform. It offers a well-rounded solution, meeting all our criteria.

As described previously, we chose *WHAD* to handle BLE aspects.

To ensure consistency and standardization in handling device data, we employed *Protocol Buffers* [7] data structures. This decision allowed us to efficiently serialize and deserialize data in a format that is both platform-independent and easily manageable.

Recognizing the importance of prompt error notification without constant manual checks, we integrated the *Ntfy* [8] tool. This tool allowed us to receive easy and immediate notifications, keeping us informed of any issues encountered during the fuzzing process without the need for continuous manual monitoring.

Following initial attempts, we also explored the option of cross-verifying packet exchanges using a *BLE sniffer*. Recognizing that the initial method for logging packet exchanges might not be entirely reliable, the dual-verification approach with a dedicated *BLE sniffer* added an extra layer of confidence to the analysis.

Figure 18 below shows our actual test bench with 3 *Raspberry Pi*, 1 *nRF52 dev board*, 1 *ESP-32 board*, 1 *Pico-W* and 1 *nRF52 dongle*.

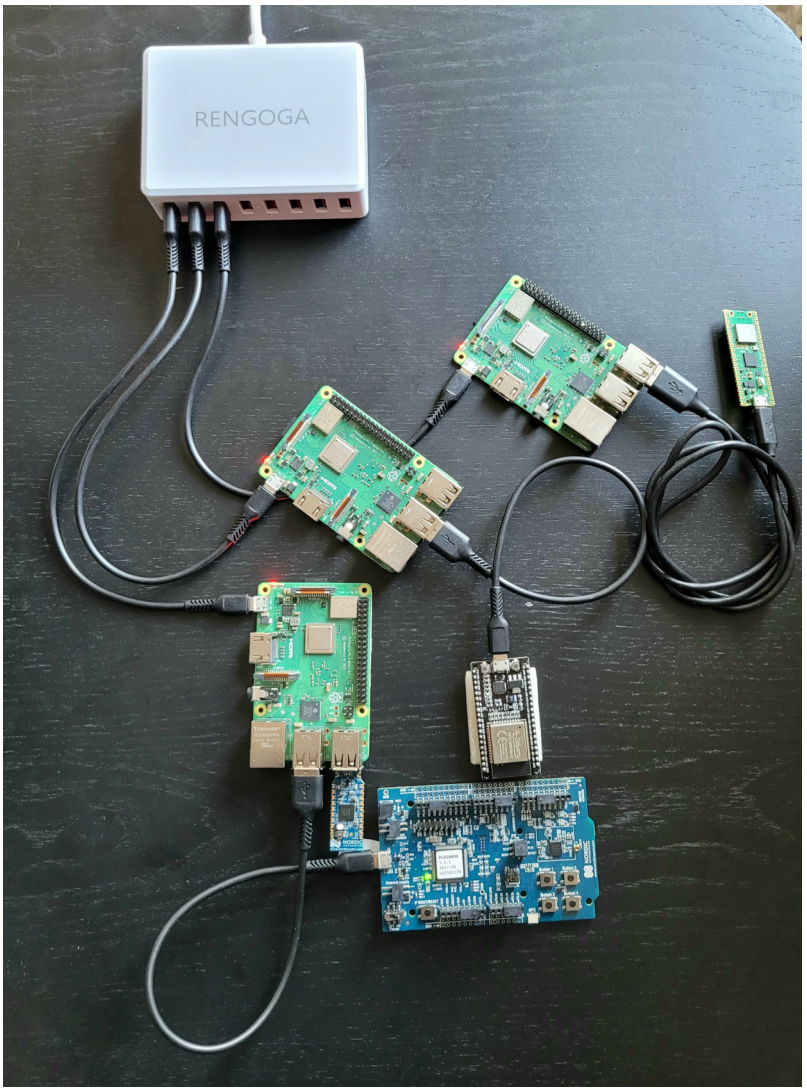


Fig. 18. Test bench

5.1 Raspberry Pi

The *Raspberry Pi* is a compact and affordable single-board computer that was created with simplicity, low power consumption, and cost-effectiveness in mind. It features various connectivity options, such as HDMI ports for display, built-in WiFi for wireless networking, USB ports for connecting peripherals, and best of all, an integrated BLE adapter! These features make it suitable for a wide range of applications, from educational projects to DIY computing endeavors.

The official operating system for the *Raspberry Pi* is Raspberry Pi OS, which is regularly updated to ensure compatibility and provide the latest features. This OS is tailored to the specific hardware architecture of the *Raspberry Pi* and comes pre-loaded with essential software and tools. The combination of its hardware capabilities and the supported operating system makes the *Raspberry Pi* a versatile and accessible platform for users to explore and implement various computing projects.

Fig. 19. Raspberry Pi

Utilizing the *Raspberry Pi* proved to be a straightforward and cost-effective solution for running our fuzzing scripts continuously for extended periods.

5.2 WHAD

Wireless HAcking Devices (*WHAD*) [17] is an open-source wireless hacking tool framework developed by Damien Cauquil and Romain Cayre.

It is intended to unify the way wireless hacking hardware devices communicate with computers in order to make them compatible with generic tools provided by the framework. The main concept behind *WHAD* is to let the hardware handle hardware tasks and keep the logic on the computer. *WHAD* provides a host side written in Python, and firmwares for compatible devices such as ESP32-WROOM-32 and nRF52840 dongles. Other devices were also made compatible with *WHAD* without the need of updating the devices firmware such as Bluetooth HCI dongles and Ubertooth One.

Our primary emphasis was on exploring BLE features. Initially, we familiarized ourselves with the framework by utilizing the provided CLI and referring to the documentation. The CLI and various examples proved instrumental in understanding the capabilities and potential of the tool. The next step involved the initiation of fuzzing script development.

WHAD offers several key features for BLE testing, including:

— Crafting and sending custom or legitimate PDUs
— Handling and processing received PDUs
— Populating and spawning a GATT server
— Logging PDU exchanges in PCAP files

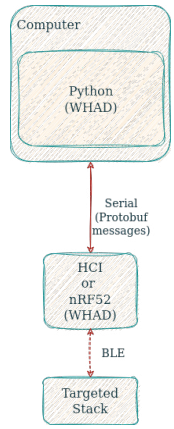


Fig. 20. WHAD

5.3 Protocol Buffers

"Protocol Buffers are language-neutral, platform-neutral extensible mechanisms for serializing structured data."

Fig. 21. Protocol Buffers logo

Protocol Buffers, commonly abbreviated as *protobuf*, originated as an internal tool developed by Google and later evolved into an open-source project. It serves as a language-agnostic data serialization format, enabling the definition of data structures in a concise and simple syntax. What sets *protobuf* apart is its cross-language support, allowing for the reuse of these data structures across different programming languages.

The key feature of *protobuf* lies in its ability to generate methods for multiple programming languages. These methods facilitate the serialization (encoding) and deserialization (decoding) of structured data into a compact binary format. This binary representation is not only efficient in terms of space but also faster to serialize and deserialize compared to other text-based formats like JSON or XML.

Protobuf is designed to be both simple and efficient. The simplicity comes from its intuitive syntax for defining message types and their fields. Additionally, the efficiency is achieved through the binary format, making it suitable for scenarios where bandwidth and processing speed are critical considerations.

Protobuf includes a dedicated compiler known as protoc, which is used to compile files with the .proto extension. These .proto files serve as the specifications where data structures, messages, and serialization rules are defined. The protoc compiler takes these high-level protocol buffer definitions and generates code in various programming languages, facilitating the integration of *protobuf* into applications across different language ecosystems. This compilation step is crucial as it produces language-specific code that can be used to serialize and deserialize data according to the specified *protobuf* schema.

```
syntax = "proto3";

package monitoring;

enum Status {
    DEFAULT          = 0;
    HARD_FAULT       = 1;
    NEW_CONNECTION   = 2;
    CONNECTION_LOST  = 3;
}

message ConnectionStatus {
    Status status    = 1;
    string info      = 2;
    optional esf esf = 3;
}

message esf{
    string r0   = 1;
    string r1   = 2;
    string r2   = 3;
    string r3   = 4;
    string ip   = 5;
    string lr   = 6;
    string pc   = 7;
    string xpsr = 8;
    string msp  = 9;
}

message Action_H_2_D{  // Host To Device
    oneof action {
        bytes generic = 1;
        bytes reset   = 2;
    }
}
```

Listing 1. monitoring.proto

```
monitoring.ConnectionStatus.info max_size:32

monitoring.esf.r0   max_size:12
monitoring.esf.r1   max_size:12
monitoring.esf.r2   max_size:12
monitoring.esf.r3   max_size:12
monitoring.esf.ip   max_size:12
monitoring.esf.lr   max_size:12
monitoring.esf.pc   max_size:12
monitoring.esf.xpsr max_size:12
monitoring.esf.msp  max_size:12

monitoring.Action_H_2_D.generic max_size:10
monitoring.Action_H_2_D.reset max_size:10
```

Listing 2. monitoring.options

Following that, we generated our Python *protobuf* file using the following command line:

```
$ protoc --python_out=./ monitoring.proto
```

Now that we have the *protobuf* file for the host, the next step is to generate the target *protobuf* file. Unfortunately, *protobuf* does not directly support plain C. However, there is an open-source project called nanopb that is built on *protobuf*.

"Nanopb is a small code-size Protocol Buffers implementation in ansi C. It is especially suitable for use in microcontrollers, but fits any memory restricted system." *Nanopb* comes with its own .proto compiler, which is used to generate .pb.c and .pb.h files.

5.4 Ntfy

"ntfy (pronounced notify) is a simple HTTP-based pub-sub notification service."

Fig. 22. Ntfy logo

Ntfy is an open-source project designed to provide straightforward notifications on a mobile device or web app. By utilizing a simple PUT or POST operation, users can easily publish messages to specific topics. The process of creating a topic is streamlined, requiring only the act of publishing or subscribing to initiate its creation. This simplicity enhances the user experience, making it convenient to set up and manage notifications for various purposes.

```
1    import pip._vendor.requests as requests
2    post_url = "https://ntfy.sh/Raspberry_Pi_ESP32_Stack_BlueDroid_QB"
3    requests.post(post_url, data="This is a test".encode(encoding='utf-8'
         ))
```

Listing 3. Simple Python post to Ntfy topic

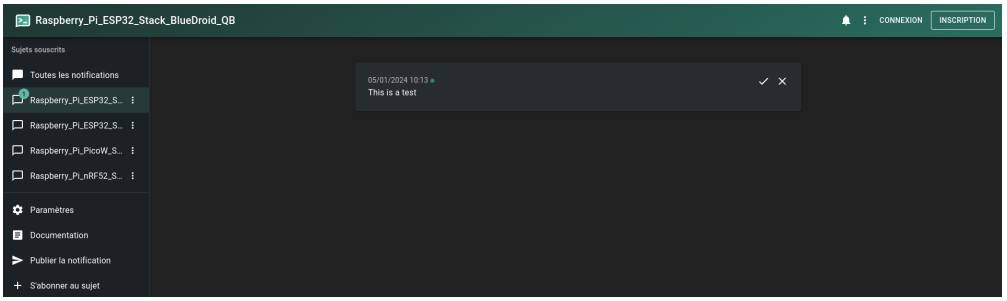


Fig. 23. Ntfy web app

During our initial fuzzing campaigns, we encountered an issue coming from the use of the basic, free-provided service. The free service imposed limitations on the number of messages, which were exceeded during the fuzzing campaign. Consequently, we made the decision to transition to a self-hosted *Ntfy* server. This switch allowed us to overcome the message limitations and ensured a more tailored and scalable notification system for our fuzzing activities.

An alternative approach would have involved creating a dedicated email address and sending data to it as a means of receiving notifications. However, the introduction of *Ntfy* proved to be a more streamlined and efficient solution, providing a convenient and direct method for notifications during our fuzzing activities.

5.5 BLE sniffer

For our *BLE sniffer*, we opted for the nRF52840 dongle paired with Nordic Semiconductor's nRF Sniffer for Bluetooth LE tool [9]. This tool facilitated the near real-time visualization of Bluetooth LE packets, enabling developers to promptly identify and address issues by providing a live view of on-air activities. The sniffer software comprised firmware programmed onto a Development Kit (DK) or dongle, complemented by a capture plugin for Wireshark. This comprehensive setup allowed for the recording and in-depth analysis of the detected data during the sniffing process.

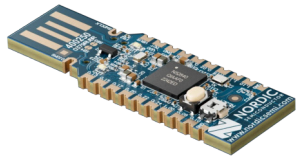


Fig. 24. nRF52840 dongle

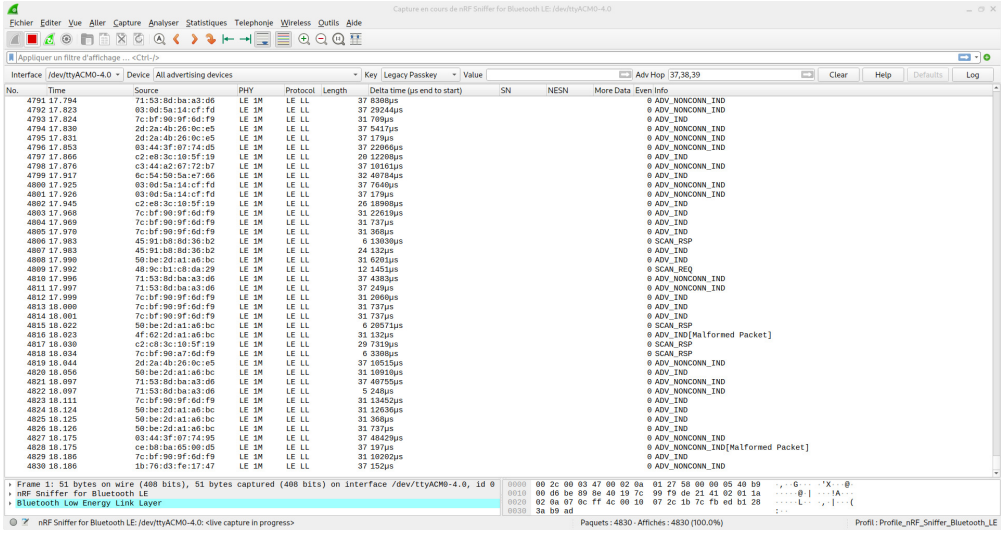


Fig. 25. nRF Sniffer interface

6 BLE stacks

Upon establishing our test bench, the pivotal decision awaited us – the selection of the BLE stack for testing and fuzzing. The landscape offered two distinct categories: proprietary and open-source. Guided by the available hardware resources, we opted to assess four prominent open-source stacks and one proprietary counterpart. The chosen open-source stacks included Zephyr [10], NimBLE [11], Bluedroid [12], and BTstack [13], each recognized for its popularity and reliability. Additionally, we included the proprietary BLE stack bundled with the CC2540 TI Bluetooth chip [14], enriching our evaluation with a diverse set of solutions. This strategic selection laid the foundation for a comprehensive assessment of BLE stacks across different paradigms, allowing us to explore their strengths and vulnerabilities within our test environment.

6.1 Zephyr

Zephyr is an open-source real-time operating system (RTOS) designed for resource-constrained devices and embedded systems. It is developed under the Linux Foundation and is known for its flexibility, scalability, and community-driven development. Zephyr supports a wide range of architectures and platforms, making it suitable for various embedded applications, including Internet of Things devices.

Fig. 26. Zephyr logo

In Zephyr, a flexible BLE stack is provided, encompassing key functionalities such as BLE profiles, BLE services, and characteristics. This stack also supports critical operations including advertising and discovery, along with robust security features.

For our project, we took advantage of Zephyr's extensive board support. Specifically, we chose to develop on the Nordic Semiconductor nRF52832 development kit. Our choice was influenced by Zephyr's default inclusion in the Nordic Semiconductor SDK nRF Connect. The nRF52832 development kit, equipped with BLE functionality, aligns seamlessly with our project requirements, and the integration of Zephyr with the Nordic Semiconductor SDK facilitates efficient development and testing processes.

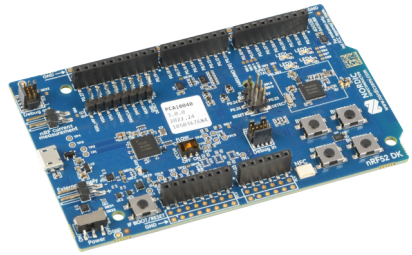


Fig. 27. nRF52832 dk

6.2 Bluedroid

Bluedroid serves as the BLE stack for Android, introduced in 2012. Originally developed by Broadcom, its primary objective was to replace Bluez, the default Bluetooth stack embedded in Linux. Over time, Bluedroid underwent a renaming process and is now known as Fluoride within the Android ecosystem. This stack has played a crucial role in enabling BLE functionalities on Android devices, offering an alternative to the Linux-based Bluez stack and contributing to the seamless integration of Bluetooth technology into the Android operating system.

In addition to its role as the BLE stack for Android, Bluedroid is one of the two stacks integrated into the Espressif IoT Development Framework (ESP-IDF) Software Development Kit (SDK) for the Host. This inclusion is particularly relevant for Espressif products, as they employ their proprietary Controller. Notably, Espressif's ESP32, on which we based our development, benefits from the integration of Bluedroid into the ESP-IDF SDK.

6.3 NimBLE

NimBLE, an integral part of the Mynewt ecosystem, stands as a BLE stack designed for resource-constrained devices in the IoT landscape. Offering a lightweight and efficient solution, NimBLE seamlessly integrates with Mynewt to provide robust BLE connectivity tailored for devices with limited memory and processing capabilities.

Fig. 28. Mynewt logo

NimBLE's core strength lies in its ability to implement the BLE protocol stack efficiently. This includes support for crucial BLE operations such as advertising, establishing connections, and facilitating data exchange. Notably, NimBLE prioritizes configurability, allowing developers to fine-tune the stack to suit the specific requirements of their IoT devices.

In addition to its role within the Mynewt ecosystem, NimBLE extends its influence to the ESP-IDF as a BLE stack for the Host. This strategic integration within ESP-IDF positions NimBLE as a key component for facilitating BLE connectivity on Espressif's ESP32 platform.

However, in the pursuit of enhancing security and addressing vulnerabilities, our development journey took a turn. Upon identifying vulnerabilities, we transitioned our development efforts to the Mynewt operating system, specifically on an nRF52 platform.

Fig. 29. ESP32

6.4 BTstack

BTstack is BlueKitchen's implementation of the official Bluetooth stack. It is well suited for small, resource-constraint devices such as 8 or 16 bit embedded systems as it is highly configurable and comes with an ultra small memory footprint.

BTstack plays a crucial role as the BLE stack embedded within the Raspberry Pi Pico SDK. This integration ensures that Raspberry Pi Pico, a popular microcontroller, is equipped with robust BLE functionality.

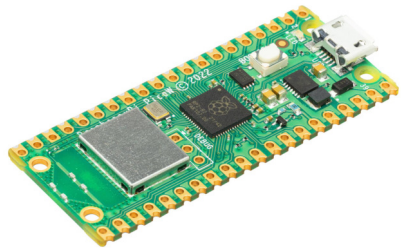


Fig. 30. Pico W

6.5 TI CC2540

The Texas Instruments CC2540 is a widely spread and use BLE system-on-chip (SoC) that serves as a versatile solution for wireless communication in various applications. With integrated BLE capabilities, the CC2540 empowers developers to create energy-efficient and feature-rich devices.

The BLE stack provided with the CC2540 is proprietary to Texas Instruments. The SDK used to run application on the CC2540 is the IAR Embedded Workbench. The stack supports fundamental BLE functionalities, such as advertising, connections, and data exchange, ensuring seamless communication in BLE-enabled applications.

7 Results

After analyzing the numerous log files (PCAP and text) generated by our fuzzer during our fuzzing campaigns, we were able to identify non-conformities, bugs, and vulnerabilities.

7.1 Non-conformities

Bluedroid

The initial non-conformity identified is associated with the Read Blob Request PDU. One of the parameters provided to this PDU is the offset indicating where to initiate the reading in the attribute. The problem arises when a high offset is provided to the PDU, yet the server responds with a simple read when attempting to read specific characteristics values. The standard behavior of the stack should be to return an Error PDU, as written in the specification: "*If the value offset of the Read Blob Request is greater than the length of the attribute value, an ATT_ERROR_RSP PDU shall be sent with the Error Code parameter set to Invalid Offset (0x07).*" [Spec Vol.3 Part.F 3.4.4.5]

No.	Time	Sourc	Destination	Protocol	Length	Info
2127	943.888764			ATT	28	Sent Read Blob Request, Handle: 0x0014 (Unknown), Offset: 62725
2166	964.648802			ATT	28	Sent Read Blob Request, Handle: 0x0007 (Unknown), Offset: 5721
2188	973.168807			ATT	28	Sent Read Blob Request, Handle: 0x0013 (Unknown), Offset: 25522
2305	1026.013508			ATT	28	Sent Read Blob Request, Handle: 0x000d (Unknown), Offset: 2194
2307	1026.273496			ATT	28	Sent Read Blob Request, Handle: 0x0007 (Unknown), Offset: 10706
2314	1029.273795			ATT	28	Sent Read Blob Request, Handle: 0x0008 (Unknown), Offset: 21377
2315	1029.453906			ATT	40	Rcvd Read Blob Response, Handle: 0x0008 (Unknown)
2373	1061.013478			ATT	28	Sent Read Blob Request, Handle: 0x0006 (Unknown), Offset: 11180
2374	1061.253887			ATT	25	Rcvd Read Blob Response, Handle: 0x0006 (Unknown)
2380	1064.138435			ATT	28	Sent Read Blob Request, Handle: 0x0010 (Unknown), Offset: 48014
2391	1067.638467			ATT	28	Sent Read Blob Request, Handle: 0x0010 (Unknown), Offset: 50720
2423	1084.795146			ATT	28	Sent Read Blob Request, Handle: 0x000a (Unknown), Offset: 27520
2447	1096.659376			ATT	28	Sent Read Blob Request, Handle: 0x0009 (Unknown), Offset: 51305
2459	1102.763430			ATT	28	Sent Read Blob Request, Handle: 0x000e (Unknown), Offset: 36260
2513	1129.773653			ATT	28	Sent Read Blob Request, Handle: 0x0003 (Unknown), Offset: 8538
2583	1158.282524			ATT	28	Sent Read Blob Request, Handle: 0x000b (Unknown), Offset: 59446

Fig. 31. Read Blob Request on specific Characteristics Values

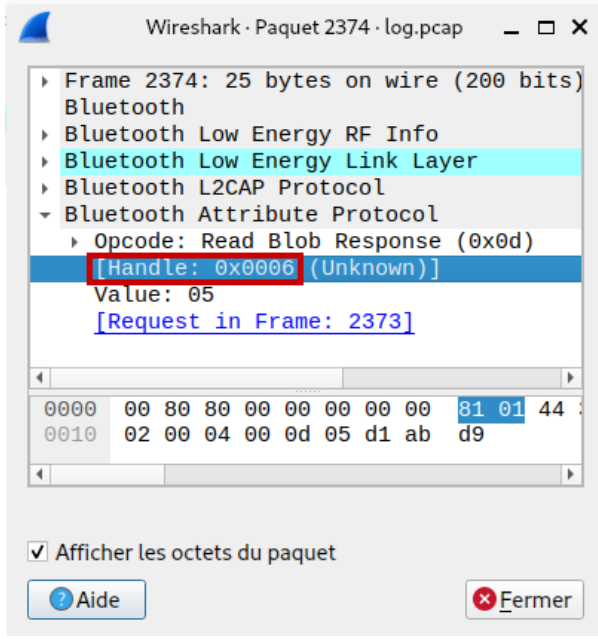


Fig. 32. Characteristic 1 details

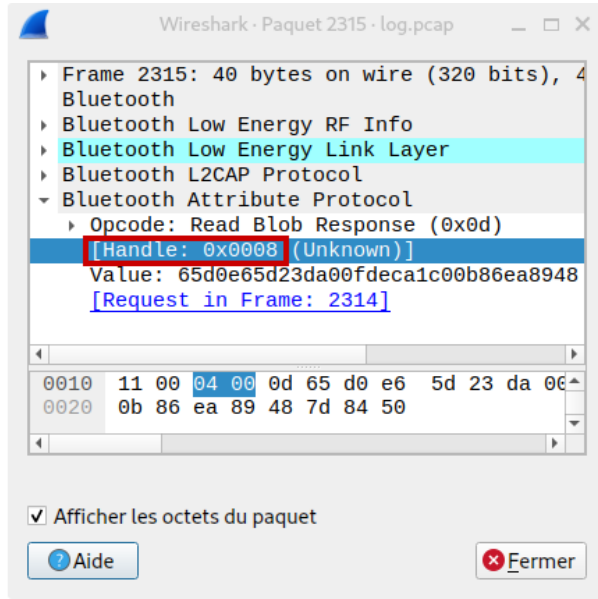


Fig. 33. Characteristic 2 details

Upon inspecting the corresponding characteristics in the GATT server, it becomes evident that they are, respectively, *Client Supported Features* and *Database Hash* characteristics.

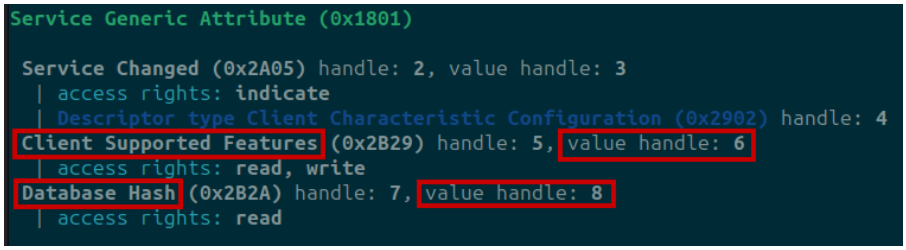


Fig. 34. Bluedroid GATT server

The *Client Supported Features* and the *Database Hash* are specific characteristics introduced in BLE 5.1. From this observation, it can be inferred that the intent of stack suppliers to keep their stacks up to date might introduce non-conformities if they do not pay sufficient attention. In this context, features introduced by BLE version 5.1 are not handled correctly with a simple ATT PDU.

NimBLE

The second non-conformity is also associated with the Read Blob Request PDU. In this case, sending a Read Blob Request with a high offset on services or characteristics may result in a response that is not an Error PDU.

No.	Time	Source	Destination	Protocol	Length	Info
16	12.877404			ATT	28	Sent Read Blob Request, Handle: 0x000f (Unknown), Offset: 38030
17	13.124062			ATT	26	Rcvd Read Blob Response, Handle: 0x000f (Unknown)
31	21.985001			ATT	28	Sent Read Blob Request, Handle: 0x000e (Unknown), Offset: 49362
39	27.626083			ATT	28	Sent Read Blob Request, Handle: 0x0002 (Unknown), Offset: 24637
40	27.884147			ATT	29	Rcvd Read Blob Response, Handle: 0x0002 (Unknown)
46	30.992637			ATT	28	Sent Read Blob Request, Handle: 0x0012 (Unknown), Offset: 45285
47	31.250169			ATT	29	Rcvd Read Blob Response, Handle: 0x0012 (Unknown)
73	45.758072			ATT	28	Sent Read Blob Request, Handle: 0x000f (Unknown), Offset: 54739
74	46.004452			ATT	26	Rcvd Read Blob Response, Handle: 0x000f (Unknown)
80	49.248861			ATT	28	Sent Read Blob Request, Handle: 0x0004 (Unknown), Offset: 9171
81	49.492261			ATT	29	Rcvd Read Blob Response, Handle: 0x0004 (Unknown)
176	107.984655			ATT	28	Sent Read Blob Request, Handle: 0x000f (Unknown), Offset: 41295
177	108.164724			ATT	26	Rcvd Read Blob Response, Handle: 0x000f (Unknown)
179	108.485704			ATT	28	Sent Read Blob Request, Handle: 0x000b (Unknown), Offset: 2995
180	108.772825			ATT	29	Rcvd Read Blob Response, Handle: 0x000b (Unknown)
196	115.494317			ATT	28	Sent Read Blob Request, Handle: 0x000a (Unknown), Offset: 1883
197	115.732969			ATT	26	Rcvd Read Blob Response, Handle: 0x000a (Unknown)
295	146.753573			ATT	28	Sent Read Blob Request, Handle: 0x0013 (Unknown), Offset: 68
305	152.377445			ATT	28	Sent Read Blob Request, Handle: 0x0007 (Unknown), Offset: 20479
306	152.565140			ATT	29	Rcvd Read Blob Response, Handle: 0x0007 (Unknown)
311	153.109127			ATT	28	Sent Read Blob Request, Handle: 0x0005 (Unknown), Offset: 33148
335	161.145823			ATT	28	Sent Read Blob Request, Handle: 0x0015 (Unknown), Offset: 24653
336	161.329728			ATT	29	Rcvd Read Blob Response, Handle: 0x0015 (Unknown)

Fig. 35. Read Blob Requests

The mapping between the received PDUs' handle parameter in the Wireshark capture above and the attribute handles from the server is provided below. Please note that the services handles are not represented in this capture.

```
ble-central|a8:42:e3:ca:bf:fe> profile
Service Generic Access (0x1800)

 Device Name (0x2A00) handle: 2, value handle: 3
  | access rights: read
 Appearance (0x2A01) handle: 4, value handle: 5
  | access rights: read

Service Generic Attribute (0x1801)

 Service Changed (0x2A05) handle: 7, value handle: 8
  | access rights: indicate
  | Descriptor type Client Characteristic Configuration (0x2902) handle: 9

Service Alert Notification (0x1811)

 Supported New Alert Category (0x2A47) handle: 11, value handle: 12
  | access rights: read
 New Alert (0x2A46) handle: 13, value handle: 14
  | access rights: notify
  | Descriptor type Client Characteristic Configuration (0x2902) handle: 15
 Supported Unread Alert Category (0x2A48) handle: 16, value handle: 17
  | access rights: read
 Unread Alert Status (0x2A45) handle: 18, value handle: 19
  | access rights: notify
  | Descriptor type Client Characteristic Configuration (0x2902) handle: 20
 Alert Notification Control Point (0x2A44) handle: 21, value handle: 22
  | access rights: write

Service 59462f12-9543-9999-12c8-58b459a2712d

 33333333-2222-2222-1111-111100000000 handle: 24, value handle: 25
  | access rights: read, write, indicate, notify
  | Descriptor type Client Characteristic Configuration (0x2902) handle: 26
```

Fig. 36. NimBLE GATT server

The issue arises from the fact that the specification suggests using the Read Blob Request method to read attributes in general in the ATT section of the specification. It further provides guidelines on how to use it to read *Long Characteristic Values* or *Long Characteristic Descriptors* in the GATT section associated with features. However, it fails to specify the expected behavior when attempting to read a service with a Read Blob Request. Consequently, it becomes the responsibility of the stack developer to interpret and implement the specification correctly, and in this instance, the specification wasn't adhered to.

7.2 Bugs

Bluedroid

The initial bug identified in the Bluedroid stack results in the application becoming unresponsive to our requests after the transmission of specific PDUs.

The bug was triggered when sending a Prepare Write Request followed by an Execute Write Request on a specific characteristic (*Client Supported Features*) or descriptor (*Client Characteristic Configuration*). This issue was particularly notable as these resources underwent a special process within the stack implementation. Additionally, the affected characteristic and descriptor were enabled by default for all applications.

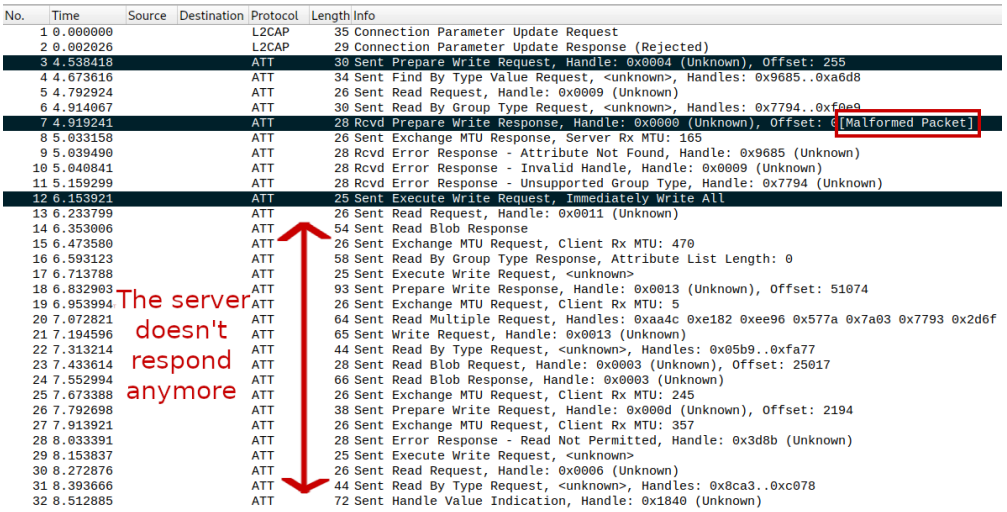


Fig. 37. Prepare Write Request on Client Characteristic Configuration

It is worth noting that the application responds with a Prepare Write Response (No.7) specified as "Malformed Packet" by Wireshark. After the Execute Write Request (No.12), we observed that the application indeed stops responding.

This bug can potentially be transformed into a vulnerability since packet injection is feasible in unencrypted BLE connections. We calculated a CVSS v3.1 score of 5.3 for this vulnerability, based on the following vector: CVSS:3.1/AV:A/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:H.

The second bug discovered in the Bluedroid stack enables the GATT server to process a Prepare Write Request on a *Client Characteristic Configuration* handled by the application, even though it is not supposed to be possible.

This bug is related to a wrong parameter given to the gatts_write_attr_perm_check() function.

```
UINT16          offset = 0;
memset(&sr_data, 0, sizeof(tGATTS_DATA));
/*...*/
STREAM_TO_UINT16(offset, p); //get offset from p_data
/*...*/
status = gatts_write_attr_perm_check (gatt_cb.sr_reg[i_rcb].p_db,
                                      op_code,
                                      handle,
                                      sr_data.write_req.offset, /*VULN*/
                                      p,
                                      len,
                                      sec_flag,
                                      key_size);
```

Fig. 38. Wrong given offset parameter

```
    else if ( (p_attr->uuid_type == GATT_ATTR_UUID_TYPE_16) &&
              (p_attr->uuid == GATT_UUID_CHAR_CLIENT_CONFIG ||
               p_attr->uuid == GATT_UUID_CHAR_SRVR_CONFIG) )
    {
        if (op_code == GATT_REQ_PREPARE_WRITE && offset != 0) { /*VULN*/
            status = GATT_NOT_LONG;
            GATT_TRACE_ERROR( "gatts_write_attr_perm_check - GATT_NOT_LONG");
        } else if (len != max_size) { /* data does not match the required format */
            status = GATT_INVALID_ATTR_LEN;
            GATT_TRACE_ERROR( "gatts_write_attr_perm_check - GATT_INVALID_PDU");
        } else {
            status = GATT_SUCCESS;
        }
    }
}
```

Fig. 39. gatts_write_attr_perm_check()

The given offset is always 0 and allows the function to return a GATT_SUCCESS status.

The commit where we can find the fixes is: `https://github.com/espressif/esp-idf/commit/b693094ee850d34892407c56fbcc4e2ac2c8276c`.

Fluoride

We also identified a similar bug involving the transmission of a Prepare Write Request followed by an Execute Write Request on the *Client Supported Features characteristic (CCC)* for the Fluoride stack. This similarity is due to the shared code base between the Espressif Bluedroid stack and the Android Fluoride stack. It's important to note that this bug in the CCC was already patched in Fluoride.

The second bug discovered also results in the application not responding to our requests. To configure and run a GATT server on an Android device, we used the Nordic nRF Connect Application with the default GATT server example. The bug is likely related to the application's handling of Prepare Write Requests rather than the stack itself. As the source code of the Nordic nRF Connect application is not available, we cannot confirm the root cause of the bug.

To reproduce this bug, we need to send a Prepare Write Request with an offset greater than the actual size of the writable characteristic.

Let's read a readable and writable characteristic, and retrieve its length, which is 6 bytes.

```
ble-central|7d:96:86:20:fa:13> read 156
00000000: 57 6F 6A 73 6B 69                              Wojski
```

Fig. 40. Value of targeted characteristic

Now, we trigger the bug by sending a Prepare Write Request with an offset (15) greater than the size (6) of the characteristic.

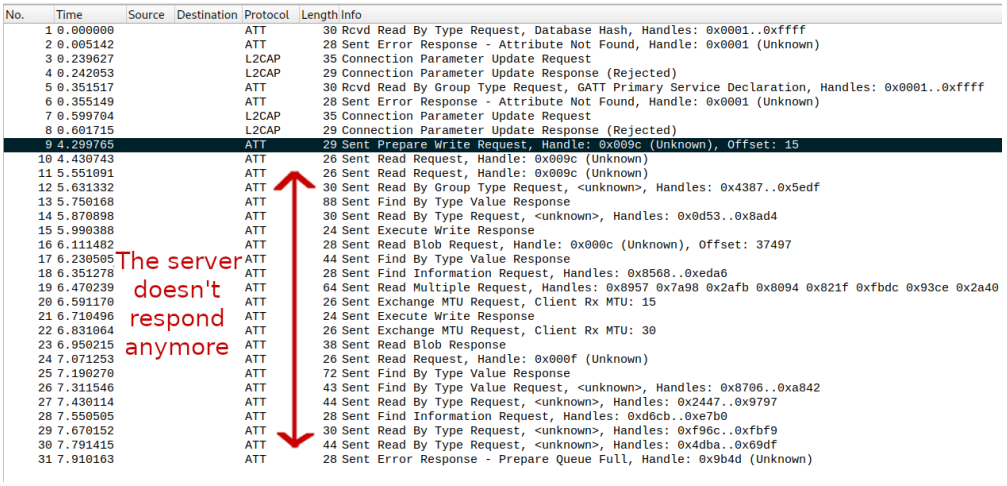


Fig. 41. Prepare Write Request on targeted characteristic with wrong offset

As shown above, any PDU sent after our Prepare Write Request is not answered by the remote device.

Similar to the previous case, this bug has the potential to be transformed into a vulnerability, as packet injection is possible in unencrypted BLE connections. We calculated a CVSS v3.1 score of 5.3 for this vulnerability, based on the following vector: AV:A/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:H.

7.3 Vulnerabilities

Bluedroid

The first vulnerability is associated with the Bluedroid stack provided in the ESP-IDF framework. More specifically, it is related to the GATT server example provided for Espressif devices, intended for using Bluedroid. This vulnerability exposes a risk of heap overflow.

We uncovered this vulnerability during the code review of the Bluedroid stack. Building on our prior discoveries of bugs, our aim was to assess the impact on user applications. Upon scrutinizing the provided GATT server example, we finally identified this Out-of-Bounds write issue.

Their initial misguided decision involves allowing the user to verify the conformity of parameters provided in the ATT Prepare Write Request by default.

The root cause of the vulnerability was located in the *example_write_event_env()* function during the copy of a received Prepare Write Request's value into the allocated buffer prepare buf as we can see in the code below:

```
memcpy(prepare_write_env->prepare_buf + param->write.offset,
       param->write.value,
       param->write.len);
```

Fig. 42. GATT server example vulnerability

An attacker controls both the offset and the value and is able to write outside of the allocated buffer. A check on the offset is done but only when the allocated buffer has already been allocated, and therefore this check is not performed for the first Prepare Write Request.

```c
if (prepare_write_env->prepare_buf == NULL) {
    prepare_write_env->prepare_buf = (uint8_t *)malloc(PREPARE_BUF_MAX_SIZE*sizeof(uint8_t));
    prepare_write_env->prepare_len = 0;
    if (prepare_write_env->prepare_buf == NULL) {
        ESP_LOGE(GATTS_TAG, "Gatt_server prep no mem\n");
        status = ESP_GATT_NO_RESOURCES;
    }
} else {
    if(param->write.offset > PREPARE_BUF_MAX_SIZE) {
        status = ESP_GATT_INVALID_OFFSET;
    } else if ((param->write.offset + param->write.len) > PREPARE_BUF_MAX_SIZE) {
        status = ESP_GATT_INVALID_ATTR_LEN;
    }
}
```

Fig. 43. GATT server example vulnerability root cause

Our proposed fix aimed to implement a continuous check for the offset and length in the ATT Prepare Write Request, aligning them with the attribute value length.

We calculated a CVSS v3.1 score of 8.8 for this vulnerability, based on the following vector: AV:A/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H.

Espressif addressed the issue in seven of their code examples related to the GATT server. They implemented a fix by introducing a continuous check for both the length and the offset as proposed.

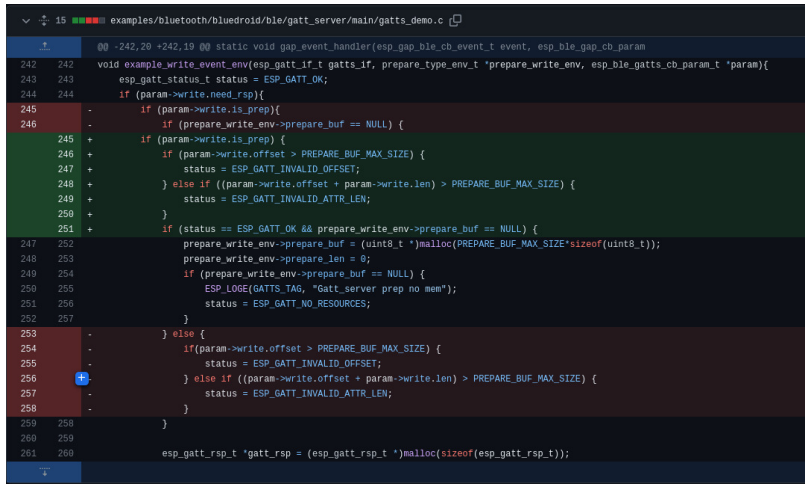


Fig. 44. GATT server example fix

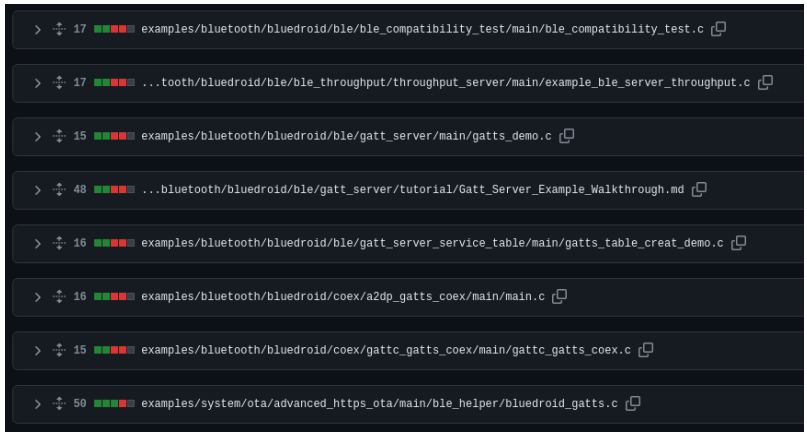


Fig. 45. Fixed files

The commit where we can find the fixes is : `https://github.com/espressif/esp-idf/commit/ed036e0ebe00e42fab7d95a15c6ccb2508aee283`.

Unfortunately, they did not categorize this as a vulnerability within their bounty program, treating it solely as a bug despite its clear manifestation as a heap overflow issue. However, following our report, this vulnerability has been fixed in a timely manner by Espressif without any communication about it, affecting 7 of their code examples. We are curious to know about the number of applications that relied on these code examples and remain susceptible to our discovery.

NimBLE

This vulnerability pertains to the NimBLE stack and poses a risk of Denial of Service (DoS). Initially, it was tested within the ESP-IDF framework made by Espressif on an ESP32-WROOM-32 module. Subsequently, the vulnerability was verified with Mynewt on a Nordic Semiconductor nRF52 module to confirm its presence not only on NimBLE Host paired with the ESP32 controller but across all controllers.

This bug came to our attention during the analysis of log files from our initial fuzzing campaign. In the PCAP files associated with this campaign, we observed instances where the device has terminated the connection by itself. Upon further investigation, we identified a recurring pattern associated with disconnections: transactions involving Prepare Write Requests that were not completed within a 30-second timeframe.

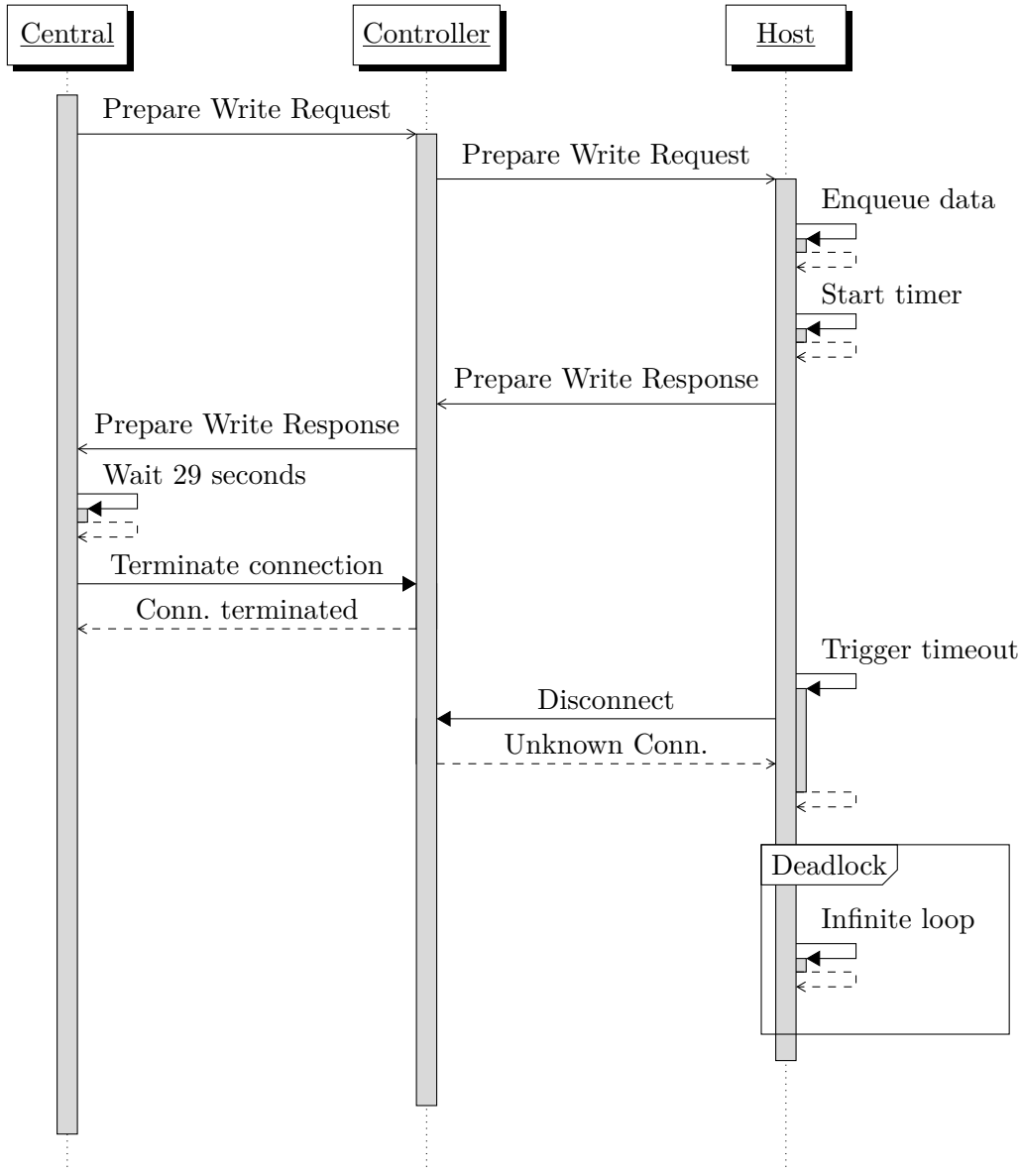
Indeed, the BLE specification stipulates that "*A transaction not completed within 30 seconds shall time out. Such a transaction shall be considered to have failed, and the local higher layers shall be informed of this failure.*" [Spec Vol.3 Part.F 3.3.3] NimBLE implements this feature with the BLE_ATT_SVR_QUEUE_WRITE_TMO timer.

```
BLE_ATT_SVR_QUEUED_WRITE_TMO:
    description: >
        Expiry time for incoming ATT queued writes (ms).  If this much
        time passes since the previous prepared write was received, the
        connection is terminated.  A value of 0 means no timeout.
    value: 30000
```

Fig. 46. BLE_ATT_SVR_QUEUE_WRITE_TMO

When the initiating device sends a Prepare Write Request to the BLE peripheral, the NimBLE stack starts a timer to remove later the pending write operations if no Execute Write Request or Prepare Write Request is received within the specified time frame. If the initiating device disconnects from the peripheral just before the timeout is reached, then the peripheral will be stuck in an infinite loop and therefore will not advertise anymore even if the current connection has been terminated.

The following diagram shows an overview of what is happening:



We calculated a CVSS v3.1 score of 6.5 for this vulnerability, based on the following vector: CVSS:3.1/AV:A/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H.

The commit where we can find the fixes is: `https://github.com/apache/mynewt-nimble/commit/d42a0ebe6632bd0c318560e4293a522634f60594`.

The vulnerability report was sent to Apache in January and CVE-2024-24746 was published at the beginning of April because they were waiting for the latest version of NimBLE to come out. More details about this CVE can be found here: `https://www.cve.org/CVERecord?id=CVE-2024-24746`

7.4 Limitations

Understanding the limitations of our work was crucial for identifying areas of improvement in our fuzzer and optimizing results. Two primary limitations were evident: the BLE version tested and the characteristics of the GATT servers used in our testing. Another limitation arose when considering the methodology used for results analysis.

BLE version

The initial limitation we encountered relates to the choice of the BLE version for our tests. The attack scenarios were crafted based on the BLE specification version 4.2 instead of the latest version, which is 5.4. This decision was influenced by two factors. Firstly, many products currently available still operate on BLE version 4.2. Secondly, the *WHAD* framework, the foundation of our fuzzer, currently only supports BLE version 4.2.

GATT servers

Another limitation pertains to the GATT servers employed in our testing. As we utilized the basic GATT servers provided by each SDK, they were relatively limited in resources and had a modest number of attributes. Utilizing GATT servers with more diverse attributes, permissions, and unique features could potentially yield more comprehensive results, covering a wider range of scenarios.

Result Analysis

The volume of logs necessitated an efficient approach to quickly identify potential non-conformities or bugs. While manual inspection of the logs was performed, the absence of an automated system meant that some instances might have been missed due to time constraints and the complexity of analyzing large datasets. The development of an automated analysis method would have provided a systematic and comprehensive examination of the logs, enabling the identification of patterns, anomalies, and additional potential issues.

7.5 Synthesis

Here is a summary table outlining the non-conformities, bugs, and vulnerabilities identified:

Stack	Description	Impact
Bluedroid	*Non-conformity*: Read Blob Request considered as Read Request on specific characteristics	None
	Bug: Application becoming unresponsive after transmission of specific PDUs	DoS
	Vulnerability: GATT server example code vulnerable to heap overflow	RCE
Fluoride	*Bug*: Application becoming unresponsive after receiving Prepare Write Request with Offset > size(ATT_Value)	DoS
NimBLE	*Non-conformity*: Read Blob Request considered as Read Request	None
	Vulnerability: Race condition within the Prepare Write Request timeout feature	DoS

Table 10. Results synthesis

8 Additional results

Based on the DoS vulnerability identified in the NimBLE BLE stack, we decided to attempt to reproduce the vulnerability on real-world devices. Our work targeted commonly available audio devices, including Sony WH-1000XM4, WF-1000XM4 and WH-1000XM5, as colleagues had these models on hand.

Fig. 47. WH-1000XM4 & WH-1000XM5

We successfully managed to crash all these devices by attacking their exposed BLE GATT servers in the same manner as we did with the NimBLE GATT server. We will detail the DoS attack process for the Sony WH-1000XM4, noting that the same process was used for the Sony WH-1000XM5 and WF-1000XM4.

Sony XM4 headset

By default, when used with Bluetooth Classic to listen to music, this device like many Bluetooth audio devices exposes a BLE GATT server typically labeled as LE_WH-1000XM4, featuring a range of characteristics with various permissions.

We conducted a reconnaissance phase by scanning the services, characteristics, and capabilities, searching for an enabled *Prepare Write Request* characteristic. We discovered the FE2C service, known as the Fast Pair Service, which is present on many Bluetooth audio devices. This service has characteristics with write permissions. When we sent a *Prepare Write Request* to the first characteristic with UUID 1234, it responded with a *Prepare Write Response*.

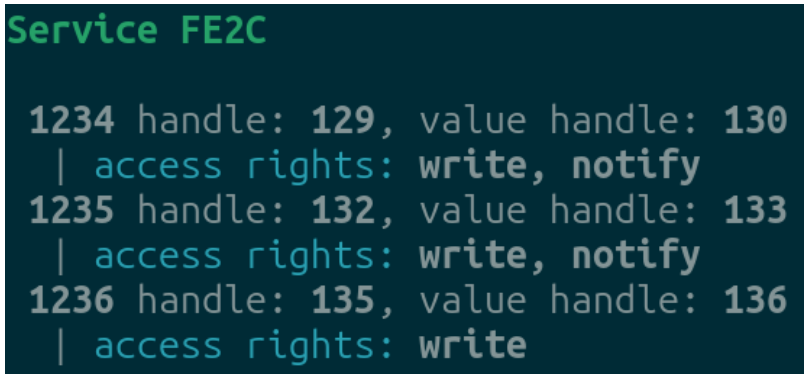


Fig. 48. Service FE2C

After finding one, we noticed that following a *Prepare Write Request* with an *Execute Write Request* caused the headset to disconnect by itself.

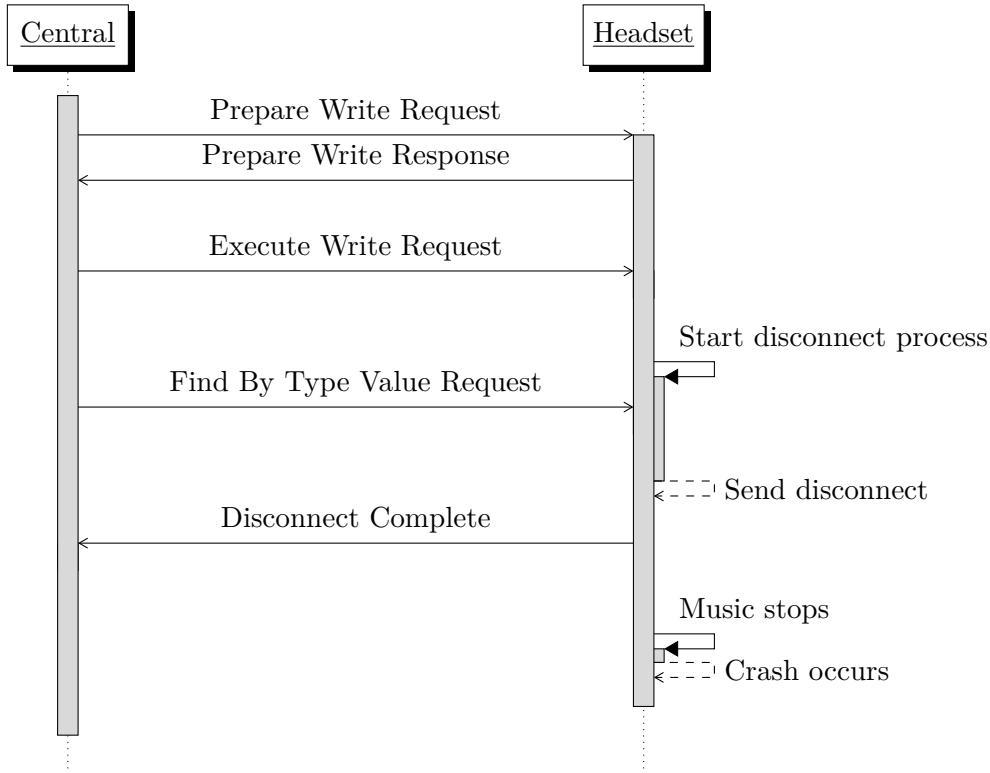
At this point, it became evident that we encountered a scenario similar to the NimBLE DoS vulnerability, indicating a potential race condition. After sending a single random request before each connection drop, we identified a packet that consistently caused the device to crash after few try.

The identified packet is the following one:

ATT_Find_By_Type_Value_Request(start, end, uuid, data)

with:
— *start* = 0x9ba9
— *end* = 0xde2d
— *uuid* = 0xbbe8
— *data* = 0xca945236aef2ab3a525bd94c06ac5806dcb2d918347add677f6b58229c44a660f6
f333893b96d9dc0793c781a83307139949534c3d9d71ceb459cb8db19140

The following diagram shows an overview of what is happening:



We calculated a CVSS v3.1 score of 6.5 for this vulnerability, based on the following vector: CVSS:3.1/AV:A/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H.

The vulnerability was reported to Sony and was fixed in October 2024:

— *WH-1000XM4*: Ver.2.6.0 (Released on October 17th)
— *WF-1000XM4*: Ver.2.1.0 (Released on October 17th)
— *WH-1000XM5*: Ver.2.3.1 (Released on October 2nd)

9 Conclusion

In this paper, we introduced our custom BLE fuzzing approach targeting the ATT and GATT layers. This approach relied on attack scenarios derived from an in-depth analysis of the specification. The implementation of these attack scenarios utilized the capabilities provided by the *WHAD* framework.

Our approach produced significant results, uncovering 2 non-conformities, 2 bugs, and 2 vulnerabilities. These findings highlight the insufficiency of information and details in the specification, emphasizing that such gaps may result in non-conformities, bugs, and vulnerabilities in stack implementations. The main vulnerability proved challenging to reproduce, indicating a need for additional tools to assist stack developers in testing their implementations more comprehensively.

In future work, our plan includes enhancing the fuzzer to address identified limitations and extending its testing scope to more sophisticated GATT servers on more BLE stacks. We strongly believed that a more comprehensive focus on the security assessment of the ATT and GATT layers is crucial, given their proximity to the application layer, where issues can directly impact functionalities.

References

1. https://manpages.debian.org/unstable/bluez/gatttool.1.en.html.
2. https://www.bluez.org.
3. https://www.bluetooth.com/specifications/assigned-numbers.
4. https://www.bluetooth.com/specifications/specs/core-specification-5-4.
5. https://github.com/seemoo-lab/internalblue.
6. https://www.raspberrypi.com/.
7. https://protobuf.dev/.
8. https://ntfy.sh/.
9. https://www.nordicsemi.com/Products/Development-tools/nRF-Sniffer-for-Bluetooth-LE.
10. https://www.zephyrproject.org/.
11. https://github.com/apache/mynewt-nimble.
12. https://android.googlesource.com/platform/external/bluetooth/bluedroid/+/refs/heads/main.
13. https://github.com/bluekitchen/btstack.
14. https://www.ti.com/product/CC2540.
15. ble-fuzzer. https://github.com/naren-jayram/ble-fuzzer, October 2020.
16. Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. The KNOB is broken: Exploiting low entropy in the encryption key negotiation of bluetooth BR/EDR. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1047–1061, Santa Clara, CA, August 2019. USENIX Association.
17. Damien Cauquil and Romain Cayre. WHAD. https://whad.io/, August 2024.
18. Tristan Claverie and José Lopes Esteves. Bluemirror: Reflections on bluetooth pairing and provisioning protocols. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 339–351, 2021.
19. Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. SweynTooth: Unleashing mayhem over bluetooth low energy. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 911–925. USENIX Association, July 2020.
20. Arun Magesh. blefuzz. https://github.com/arunmagesh/blefuzz, December 2017.