Sneak into buildings with KNXnet/IP

Claire Vacherot

Sneak into buildings with KNXnet/IP

Claire Vacherot*, Orange Cyberdefense, 2020

Abstract. Building Management Systems (BMS) centralize and automate essential assets in a building. They are often linked to the LAN and sometimes reachable on the Internet, exposing building automation devices and network protocols that are usually not designed to handle cybersecurity issues. The paper focuses on the BMS protocol KNX, which has been left aside by the cybersecurity community so far. We discuss its technical details and the cybersecurity concerns raised by implementations. We provide a Python library to perform basic KNX discovery, communication operations and to write advanced testing scripts. We explain how to use it through fuzzing script examples. We hope that this library will be used to find and fix vulnerabilities in building management systems and as a handy tool for other research material on BMS protocols.

1 Introduction

As part of the "internet of things" trend which tends to connect field-level devices and protocols to information-level networks, building components such as lighting, shutters, HVAC, air quality measurement, access control, fire detection, or security systems are now expected to be monitored, controlled and automated from one (or more) central location. Such systems are referred to as Building Management System (BMS) or Building Automation System (BAS) and raise many cybersecurity concerns. Devices on BMS interact using specific field protocols, which are now often connected to the LAN and may be exposed on the Internet. However, these protocols and their implementations on devices usually do not cope with cybersecurity-related events. By breaching into a building management system, a malicious user may not only disrupt the facilities' operations (causing damages to the building, the production and/or impacting people's safety and trust in such systems), but also give them a foothold or a way to move to critical network areas and systems [FS2019].

Although there are research materials on building management systems [MIRSKY2017], protocols [PEACOCK2019] and devices' [MCKEE2019] security, we believe that the subject is far from being exhausted and requires more

*claire.vacherot@orangecyberdefense.com, https://orcid.org/0000-0001-8236-6599

attention and awareness from a cybersecurity point of view. In this paper, we focus on the field-bus protocol KNX and its transmission mode over IP: KNXnet/IP.

1. We share our experience and findings on the overall exposure and (lack of) integration of cybersecurity measures in KNXnet/IP specifications and implementations and show that we can already cause severe damages using legitimate KNX features (Section 2).
2. In order to extend attack scenarios to operations not explicitly provided by the standard, we propose to use crafted KNX frames to target unprotected or vulnerable KNXnet/IP implementations and devices. Still, this approach requires technical knowledge about the frames' specifications.
3. To address this issue, we propose a Python library to interact with KNX devices via KNXnet/IP and to read and write frames. We explain how this library can be used to write legitimate KNX communication scripts and penetration testing tools (Section 3).
4. Finally, we show how to use our library for more advanced vulnerability testing and research on KNXnet/IP by presenting a simple fuzzing script that relies on it (Section 4).

2 What's wrong with KNX?

The most common BMS protocols include BACnet and KNX [BACS2017]. They are used to exchange data over field bus (twisted pair, power line, etc.) and both of them provide specifications for communication over IP (BACnet/IP and KNXnet/IP). As stated in KNXnet/IP's system specification overview, *"Widespread deployment of data networks using the Internet Protocol (IP) presents an opportunity to expand building control communication beyond the local KNX control bus"* [KNX030801]. Needless to say, this also extends the attack surface and increases the number of attack vectors. On April 14th, 2020, searching for "BACnet" on Shodan gives 7396 results, including 5271 in the Unites States (71%). Using the keyword "KNX", Shodan finds 17 767 results, mostly located in Western Europe. Yet, this may not give the actual distribution of such protocols in BMS, as BACnet is mostly used in industries and KNX is also widespread for domestic use and is more likely to be exposed.

2.1 KNXnet/IP basics

KNXnet/IP allows interfacing *"from LAN to KNX and vice versa"*. It requires a KNX-to-IP network connection device (router/gateway), referred to as KNXnet/IP server in the standard. The server is linked to the IP network and to one or more KNX subnetworks containing a set of KNX objects (sensors, actuators, controllers). The objects are reachable from the server using KNX-specific addresses. A supervision software is commonly used to manage KNX projects and control devices, such as the commercial "Engineering Tool

Software" (ETS) developed and promoted by the KNX Association. Figure 1 shows a typical example of a BMS network architecture using KNXnet/IP.

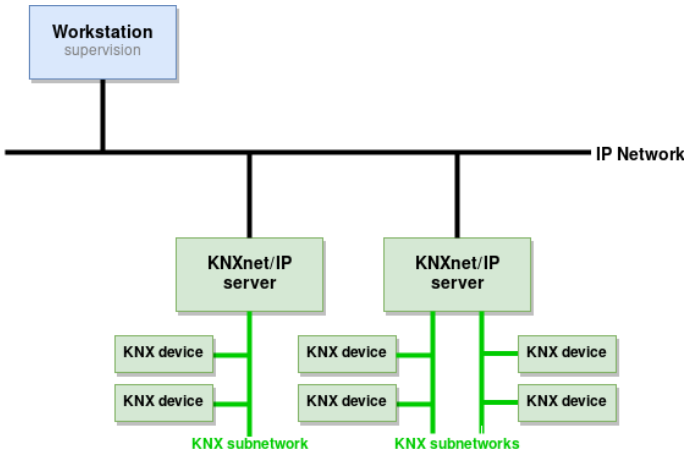

Figure 1: BMS network architecture example using KNX and KNXnet/IP

A KNXnet/IP frame is usually sent over UDP (although the specifications state that TCP can also be used [KNX030801]) and contains a header and a body with varying content, as represented in Figure 2. The structure of a frame is quite complex: the format, order and content of blocks and fields in the body change for each type of message that can be carried by a KNX frame (read or send information about devices, change configurations, etc.).

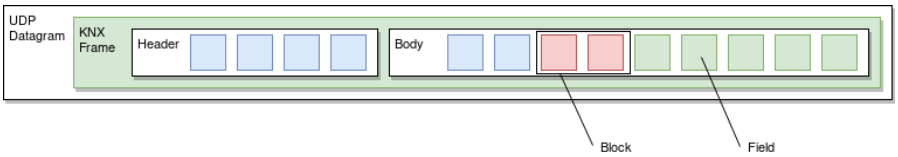

Figure 2: KNX frames representation

2.2 Security considerations

According to the standard's specification, *"For KNX, security is a minor concern, as any breach of security requires local access to the network"* [KNX030801]. Yet, KNXnet/IP aims at making KNX networks more accessible, by interconnecting it to *"existing data networking technology"* (IP), both locally and remotely. In

other words: the KNX protocol has not been designed with cybersecurity in mind and used to rely only on the low exposure of KNX systems. Consequently, the threats brought by the use of KNXnet/IP have to be handled by the IP network layer and/or upper-level layers and are not covered by the standard. Some mitigations are suggested, including the use of VPNs, network isolation, port filtering, restricting access to the supervision software (ETS) and "use [of] authentication when opening point-to-point connections". The KNX association addressed these weaknesses by publishing two extensions to the standard, KNX IP Secure and KNX Data Secure that manufacturers are free to implement (or not). Both extensions are independent but can be combined: IP Secure adds datagram encryption and Data Secure adds an authentication code and a sequence number to every KNX frame, which should be verified by implementations. Since we have not encountered any use of these extensions so far, we excluded them from the scope of this study.

To sum up, the specifications do not provide consistent cybersecurity requirements and security features are extensions of the standard, which make them look optional. KNX devices and systems manufacturers are not "required" to follow these recommendations and when they do, they are indeed described as a bonus. For instance, a KNXnet/IP gateway we used for this study implemented authentication, but it was not enabled by default.

2.3 Addressed threats

Threats targeting BMS systems and threats specific to KNX have been discussed by Brandstetter and Reisinger [BRANDSTETTER2017]. Based on these observations, we decided to focus on two main threats, described in tables Threat 1 and Threat 2 below.

Threat 1	**BMS degradation using regular KNX features**
Prerequisites	Lack of protection on KNX services (authentication, network segmentation, etc.)
Process	Send valid KNX frames to unprotected KNXnet/IP servers to act on underlying KNX objects
Expected results	Alter BMS operations (turn objects on/off, change configuration values and thresholds, etc.)

Threat 2	**Vulnerability discovery using malicious frames**
Prerequisites	Vulnerability in the implementation of a KNXnet/IP server
Operation	Send invalid inputs (frames) to test the server's robustness (fuzzing)
Expected results	Elevate privileges on the server for further usage (backdooring, network pivoting, etc.)

We believe that the following attack scenario, related to Threat 1, is very likely to happen: A malicious user massively sends valid KNX frames (e.g. configuration frames) to KNXnet/IP servers exposed on Shodan (nearly 18.000 on April 2020). This scenario may cause severe damages on a high number of BMS installations behind exposed KNXnet/IP servers that do not use or implement authentication. Existing tools implementing the KNX protocol for home automation (e.g. for Arduino) or the auditing tool KNXmap [TOOL_KNXMAP] can be used to do so, and do not require any knowledge about the protocol from the user.

As for Threat 2, a large amount of known attacks and vulnerability research projects targeting IoT, industrial and BMS devices relied on testing network protocol implementations, often with fuzzing. For instance, McKee and al. discovered critical vulnerabilities in a BMS controller using a BACnet/IP fuzzer [MCKEE2019]. However, we haven't found any targeting KNXnet/IP. Why not? Our assumption is that crafting frames that are valid enough not to be rejected by a KNXnet/IP server and invalid enough to cause unexpected behaviors requires knowledge on how to build KNX frames. Moreover, they can hardly be achieved with the tools mentioned previously or supervision tools such as ETS, mainly because they do not offer (sufficient) control over frames' content.

3 Overview of BOF

The *security considerations* section from KNXnet/IP's specifications overview concludes: "*It is quite unlikely that legitimate users of a network would have the means to intercept, decipher, and then tamper with the KNXnet/IP without excessive study of the KNX Specifications.*" [KNX030801]. This assumption emphasizes that the standard's security mostly relies on ignorance about the protocol (supervision software such as ETS do not require extended knowledge of the standard) and the lack of people willing to dive into the specifications. Here, the word "*decipher*" does not seem to be related to cryptography and could rather be replaced by "*understand*". However, the concept of "security by obscurity" that is pledged has still not been proven efficient, neither for security nor for hiding an actual lack of security considerations [CWE656].

We wrote a Python library that does not (necessarily) require this excessive study of the KNX specifications from the end user: **BOF** (Boiboite Opener Framework). It can be imported and used in Python 3.6+ scripts to interact with field protocols implementations and devices, and provides means to create, parse and manipulate frames from supported protocols. The library currently supports KNXnet/IP, which is our focus, but it can be extended to other types of BMS or industrial network protocols.

The project's source code and documentation can be found at https://github.com/Orange-Cyberdefense/bof. Please note that targeting BMS systems can have a severe impact on buildings and people and that BOF must be used carefully.

We identified three use cases depending on the end users' level of knowledge about the protocol and the specifications, as shown in Figure 3.

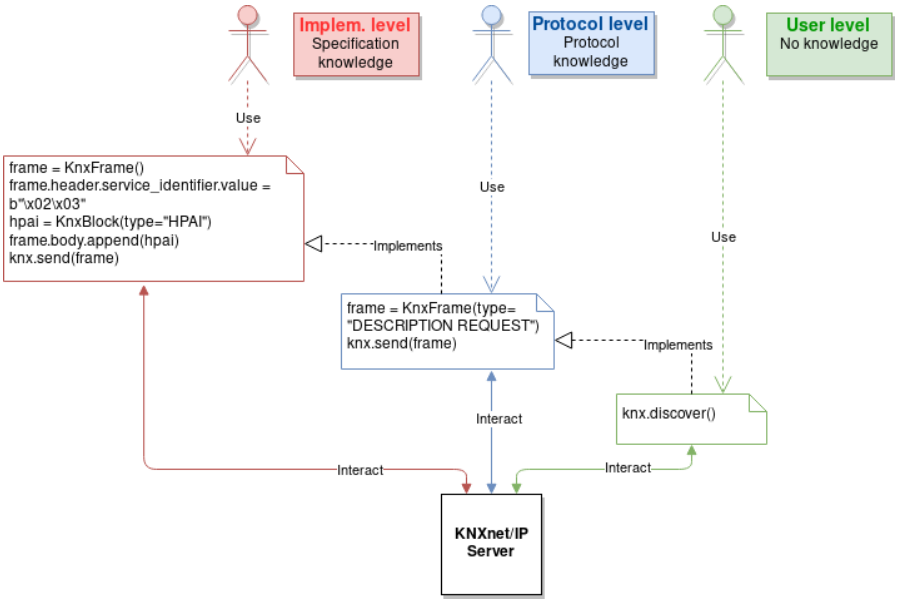

Figure 3: BOF use cases

Levels 2 and 3 features are intended for higher-level interaction according to the standard and require from no to basic knowledge about the protocol. They are suitable to write scripts related to Threat 1. The following code sample (Listing 1) is an example on how to gain basic information about a KNXnet/IP server using BOF's Level-2 features. The expected output is given in Appendix 1.

```
from bof import knx, BOFNetworkError

knxnet = knx.KnxNet()
try:
    knxnet.connect("192.168.1.1", 3671)
    frame = knx.KnxFrame(type="DESCRIPTION REQUEST")
    knxnet.send(frame)
    response = knxnet.receive()
    print(response)
except BOFNetworkError as bne:
    print(str(bne))
finally:
    knxnet.disconnect()
```

Level 1-related content provides means for building from scratch and modifying every part of a frame, complying with the specifications or not. For instance, the line `frame = knx.KnxFrame(type="DESCRIPTION REQUEST")` from Listing 1 can also be written as in Listing 2 :

```
frame = knx.KnxFrame()
frame.header.service_identifier.value = b"\x02\x03"
hpai = knx.KnxBlock(type="HPAI")
hpai.ip_address.value = "127.0.0.1"
hpai.port.value = 44000
frame.body.append(hpai)
```

As BOF is primarily a testing framework, the protocol specifications are meant to be misused and altered. Therefore, we tried to bind the code to the specifications as loosely as possible, so that a user can make changes to built-in structures and behaviors. This allows a deeper control over frames in order to write more advanced testing scripts.

4 Testing KNX devices with BOF

Let's see how we can use BOF to test KNX devices. As mentioned previously, sending legitimate KNX frames mostly grants access to legitimate KNX features (Threat 1). One way to go beyond the features provided on a KNX installation is to discover exploitable vulnerabilities on underlying devices (Threat 2).

Our approach in this paper is to target them via the KNXnet/IP protocol, by testing implementations of KNXnet/IP and KNX layers for vulnerabilities. To do so, we chose to use **fuzzing**, relying on BOF to craft invalid and unexpected data embedded in KNXnet/IP frames and send them to devices. As KNXnet/IP implementations on devices are frequently either written in native languages (C, C++) or interpreted languages, we expect to find both memory corruption-based (native languages) and data processing vulnerabilities.

4.1 KNX frames' representation and use in BOF

Before going further, it's important to understand how BOF represents and gives access to KNX frames' contents. Frames are sent and received as byte arrays. They can be divided into a set of blocks, which contain a set of fields of varying sizes. Conforming to the KNX Standard v2.1 [KNX03], the header's structure never changes and the body's structure varies according to the type of the frame given in the header's `service identifier` field. For instance, the format of a `DESCRIPTION REQUEST` message is highlighted in Figure 4 extracted from the specifications [KNX030802]. Figure 5 illustrates how a KNX frame (as a byte array) is shaped according to this format.

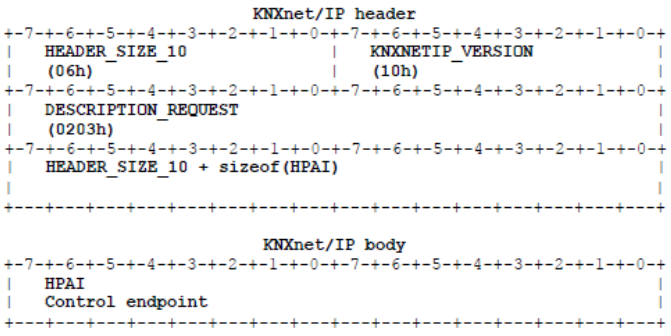

Figure 4: KNX frame format for "DESCRIPTION REQUEST" messages

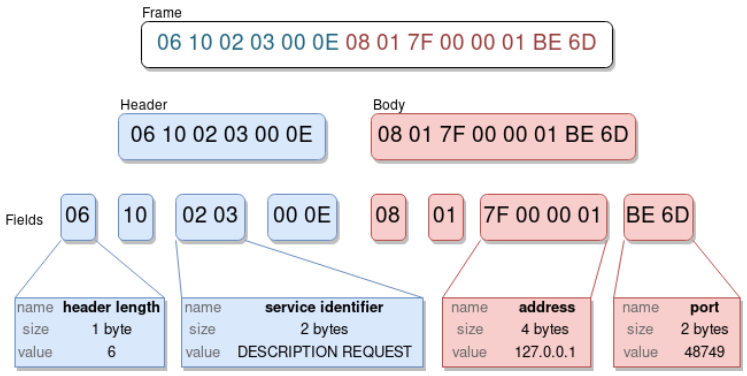

Figure 5: Content of a "DESCRIPTION REQUEST" frame

In BOF, frames, blocks and fields are represented as objects (classes). A frame (`KnxFrame`) has a header and a body, both of them being blocks (`KnxBlock`). A block contains a set of raw fields (`KnxField`) and/or nested `KnxBlock` objects with a special structure (ex: `HPAI` is a type of block with fixed fields). Finally, a `KnxField` object has three main attributes: a `name`, a `size` (number of bytes) and a `value` (as a byte array). A `KnxFrame` object based on a frame with the `DESCRIPTION REQUEST` service identifier should have the pattern illustrated in Figure 6.

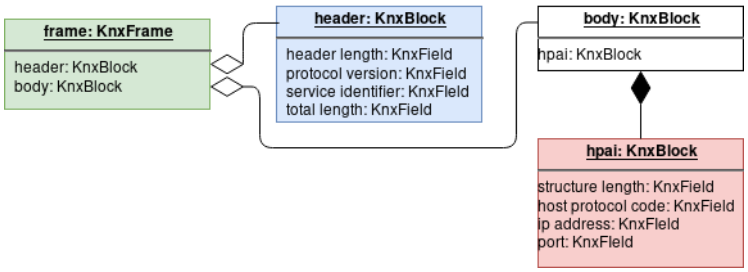

Figure 6: BOF representation of a "DESCRIPTION REQUEST" frame

As no standard-specific content or behavior shall be found or written to the code, BOF builds type-dependent frames and blocks according to an external JSON file containing the definition of message codes, block types and frame structures. Appendix 2 shows the portion of the JSON file written according to the specification and used by BOF to create the `DESCRIPTION REQUEST` frame.

Finally, within a script using BOF, a `KnxFrame` can be built either from scratch (creating each block and field one by one), from a raw byte array that is parsed (usually a received frame) or by specifying the type of the frame in the constructor. The content of the frame (blocks and fields) can be accessed using properties created according to their names, such as in the example below. Here, `frame.body` and `frame.body.control_endpoint` are identical, since the body contains only one block (`control endpoint`):

```
>>> bytes(frame)
b'\x06\x10\x02\x03\x00\x0e\x08\x01\x7f\x00\x00\x01\xbe\x6d'

>>> bytes(frame.body)
b'\x08\x01\x7f\x00\x00\x01\xbe\x6d'

>>> bytes(frame.body.control_endpoint)
b'\x08\x01\x7f\x00\x00\x01\xbe\x6d'

>>> bytes(frame.body.control_endpoint.ip_address)
```

```
b'\x7f\x00\x00\x01'
```

4.2 Writing fuzzing scripts

For this demonstration, we use BOF to write a fuzzing script to test how devices handle read and write orders interpreted or relayed by the KNXnet/IP server to the KNX bus. Such orders are sent as medium-independent KNX data with a generic structure included in some frames as a special block (`cEMI`, for Common External Message Interface) [KNX030603].

We generate inputs following the format of a valid frame that includes a `cEMI` block and mutate its fields in order to trigger unexpected behaviors. The aim is to detect field properties and values that are not securely handled either by the server or underlying devices so that further tests concentrate on these weak points. To write this script, we rely on the fuzzing scheme proposed by Jurczyk [JURCZYK2016] and reproduced in Figure 7. The different steps along with code samples (in Python 3.6) are given below.

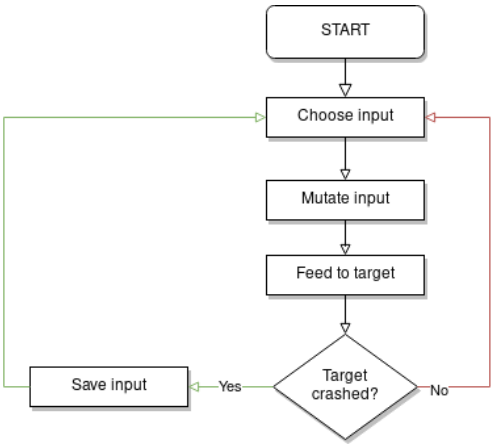

Figure 7: Fuzzing scheme (Jurczyk, 2016)

4.2.1 Choose input

We limit our test set to inputs built regardless of targeted devices' vendor, model or implementation details, as the content of a `cEMI` block should not vary according to these factors. This implies that, while inputs sent to targets closely comply with the format defined by KNX specifications (frames with invalid format are usually ignored by devices), their content is not bound to a specific context ("*dumb inputs*"). However, we are confident that BOF can also be used

for fine-grained testing of specific implementations with higher code coverage requirements.

Our script generates and sends inputs based on configuration request frames [KNX030803], which contain the special block we target, following the format depicted in Figure 8. The included `cEMI` block acts as an independent generic message with its own type within the frame, and can have a different structure and content depending on the type of operation it carries.

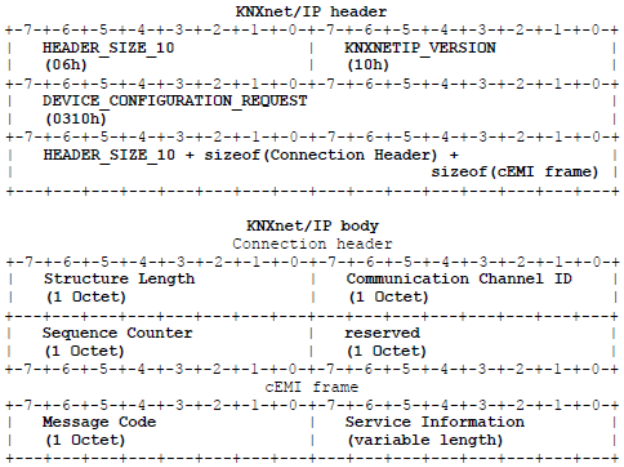

Figure 8: Format of a "CONFIGURATION REQUEST" frame

We start by targeting one type of cEMI message, `PropRead.req` to read properties on a device following a format such as in Figure 9 extracted from the specification [KNX030603]. We could also test for property write requests (`PropWrite.req`, which use the same format) and change, for instance, the content of the data field, which carries a new value for a given property. Such tests would be likely to cause unexpected behaviors on actual (not simulated) devices.

Message Code	Interface Object Type		Object Instance	Property ID	number of elements	start index	Data
MC	**IOTH**	**IOTL**	**OI**	**PID**	**NoE**	**SIx**	**Data**
1 octet	2 octets		1 octet	1 octet	4 bit	12 bits	var. length

Figure 9: Format of a PropRead.req cEMI message

4.2.2 Mutate input

We first rely on a very basic method to mutate inputs: we write random values to random fields (one at a time) in the targeted `cEMI` block and send a frame to the device for each mutation. This gives us a first overview of how, where and when targeted devices behave unexpectedly. We write a generator function that yields one frame (`KnxFrame` object) per field mutation, as written in Listing 4. Since the field `message code` defines the type of message (`PropRead.req`), we exclude it from the mutations.

```
def mutate(propread_req:knx.KnxFrame, trials:int=10000):
    fields_to_mutate = [x for x in propread_req.body.cemi.fields if \
     x.name != "message code"]
    for _ in range(trials):
        field = choice(fields_to_mutate)
        save = field.value
        field.value = bytes(map(getrandbits,(8,)*field.size))
        yield propread_req, str(field)
        field.value = save
```

4.2.3 Feed to target

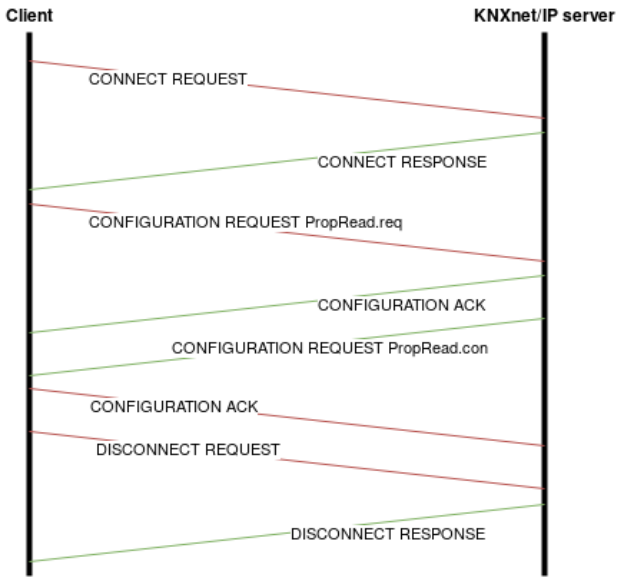

Figure 10: Minimum exchange required for reading a property on a device

Figure 10 schematizes a regular frame exchange involving a `CONFIGURATION`

REQUEST frame. In the fuzzing script, we want to reproduce such exchange and detect unusual behaviors that could indicate bugs or flaws, based on the way the test device behaves and responds to mutated frames. Input frames, fields and data that triggered them are saved for further investigation. To begin with, we focus on two types of behaviors to detect:

1. Acknowledgement messages with an error status (the configuration request is invalid but the device did not ignore it)
2. Time out errors (the test device did not respond)

The code sample in Listing 5 shows how we handle the exchange and detect both behaviors. The complete code can be found on BOF's repository.

```
knxnet, channel = connect("192.168.1.1", 3671)
initial = knx.KnxFrame(type="CONFIGURATION REQUEST", cemi="PropRead.req")
initial.body.cemi.number_of_elements.value = 1
initial.body.communication_channel_id.value = channel
sequence_counter = 0
for propread_req, data in mutate(initial):
    propread_req.body.sequence_counter.value = sequence_counter
    try:
        knxnet.send(propread_req)
        received_ack = knxnet.receive()
        if received_ack.body.status.value == STATUS_OK:
            propread_con = knxnet.receive()
            if propread_con.sid == "CONFIGURATION REQUEST":
                ack_to_send = knx.KnxFrame(type="CONFIGURATION ACK")
                ack_to_send.body.communication_channel_id.value = channel
                ack_to_send.body.sequence_counter.value = sequence_counter
                knxnet.send(ack_to_send)
         else:
             save("Error in acknowledgement", propread_req, data, received_ack)
     except BOFNetworkError:
         save("Timeout", propread_req, data)
     sequence_counter += 1
disconnect(knxnet, channel)
```

Here, we respond with an acknowledgement frame as well, since our testing device would send the same message 10 times if we didn't. Besides, this behavior could be taken advantage of for denial of service attacks: sending a single frame with the source address changed would result in the device sending 10 frames to a target.

4.2.4 Going further

This simple fuzzing script is only able to give an overview of weaknesses on a KNX device targeting specific parts of a frame, as a first approach. Here, we detect unexpected behaviors and associated inputs only by looking at what we

receive on the client side, but a more efficient way to interpret the results would be to monitor what happens on the target as well, e.g. with a debugger.

Still, we were able to trigger **20** unexpected behaviors, out of around 1,2 million test frames sent in one hour of fuzzing our testing device. The next step is obviously to investigate: to dig into the input frames and conditions that triggered errors to try to find exploitable bugs. A way to do it is to gradually restrict inputs and mutations to specific types of messages, fields to modify and range of values according to how the target responds. The advantage of using BOF to do so lies in the ability to create and mutate identified and named frames, blocks and fields, and therefore fuzz while sticking to the specifications.

5 Related work

Work regarding KNX and KNXnet/IP networks' security is quite introductory so far. Global security concerns and main threats related to this protocol have been discussed in BMS security overviews papers and reports [BACS2017], [BRANDSTETTER2017]. The occasional talks in cybersecurity technical conferences on this subject usually describe the protocol, highlight its lack of network security protections and potential impacts, and show how to abuse them by taking control of targeted installations with valid frames. Among the most recent ones, Litvinov's talk at Zero Nights 2015 [LITVINOV2015] and Hui Yu et al.'s talk at Hack in the Box 2018 [HUIYU2018] follow this pattern.

At the same time, research material on the analogous protocol BACnet/IP include more advanced approaches regarding cybersecurity. In 2015, Kaur et al. presented a traffic normalization method to prevent attacks on BACnet networks [KAUR2015]. Gasser et al. [GASSER2017] evaluated the vulnerability of BACnet to amplification attacks and provided a basic BACnet response parsing tool [TOOL_BACNETPY]. More recently, Peacock published a thesis demonstrating anomaly detection methods on BACnet/IP networks [PEACOCK2019]. A few implementations of BACnet/IP for fuzzing exist (bacnet-scapy [TOOL_SCAPYBAC], Fuzzowsky [TOOL_FUZZOWSKY]) but we haven't found a comprehensive one.

As for related tools, BOF was built following the model of some existing KNX implementations: First, we have to mention the auditing tool KNXmap [TOOL_KNXMAP], that we worked with a lot but could unfortunately not use as a basis for BOF, as it has not been designed to craft and send invalid frames. Wireshark's KNXnet/IP dissector is also a very good means to understand KNX traffic, both when used inside Wireshark and when looking at the dissector's source code [SRC_WSDISSECT], which is much more understandable than the specifications. Finally, some widespread tools include basic contents for KNX such as the network security scanner nmap, which has scripts for KNX discovery [TOOL_NMAP].

6 Discussion and future work

The cybersecurity world does not seem to know much about Building Management Systems, and the BMS world does not know much about cybersecurity. There are a lot of steps to take to make both worlds meet and we wrote BOF to contribute: we expect BOF, and any future project on BMS cybersecurity we plan, to provide ways to understand BMS environments and to discover how they work, how they can be misused and how to test them in order to secure them. First of all, we wanted to write a tool that can be used without prior requirements on cybersecurity and / or BMS, but we believe that our next move should be to contribute to existing cybersecurity projects by adding BMS-related content and protocol implementations.

BOF is still an ongoing project and so far, we focused on the detailed implementation of the specification. One can already do the main basic operations, but there are many types of frames and features in the KNX standard (without even mentioning the extensions) and not all of them have been implemented yet. Furthermore, there will still be room for higher-level functions that will make tests easier. And of course, every contribution is welcome.

7 Appendix

7.1 Sample output for BOF's KNX discovery script

Output of `print(frame)` where `frame` is the `KnxFrame` object representation of a `DESCRIPTION RESPONSE` message received from a server.

```
KnxFrame object: <bof.knx.knxframe.KnxFrame object at 0x7f7b3b7e42b0>
[HEADER]
  <header length: b'\x06' (1B)>
  <protocol version: b'\x10' (1B)>
  <service identifier: b'\x02\x04' (2B)>
  <total length: b'\x00D' (2B)>
[BODY]
  KnxBlock: device hardware
    <structure length: b'6' (1B)>
    <description type code: b'\x01' (1B)>
    <knx medium: b'\x02' (1B)>
    <device status: b'\x00' (1B)>
    <knx individual address: b'\xff\xff' (2B)>
    <project installation identifier: b'\x00\x00' (2B)>
    <knx serial number: b'\x00\x00T\xff\xf4\x13' (6B)>
    <multicast address: b'\xe0\x00\x17\x0c' (4B)>
    <mac address: b'\x00\x00T\xff\xf4\x13' (6B)>
    <friendly name: b'boiboite\x00\x00\x00\x00\x00\x00\x00 [...]' (30B)>
  KnxBlock: supported service families
```

```
    <structure length: b'\x08' (1B)>
    <description type code: b'\x02' (1B)>
    KnxBlock: service family
      <id: b'\x02' (1B)>
      <version: b'\x01' (1B)>
```

7.2 JSON specification file extract

Portion of the JSON file written according to the specifications that is used to build a `KnxFrame` object of type `DESCRIPTION REQUEST`.

```
{
 "service identifiers": {
 "DESCRIPTION REQUEST": {"id": "0203"}
},
"bodies": {
 "DESCRIPTION REQUEST": [
     {"name": "control endpoint", "type": "HPAI"}
 ]
},
"blocktypes": {
 "HEADER": [
     {"name": "header length", "type": "field", "size": 1, "is_length": true},
     {"name": "protocol version", "type": "field", "size": 1, "default": "10"},
     {"name": "service identifier", "type": "field", "size": 2},
     {"name": "total length", "type": "field", "size": 2}
 ],
 "HPAI": [
     {"name": "structure length", "type": "field", "size": 1, "is_length": true},
     {"name": "host protocol code", "type": "field", "size": 1, "default": "01"},
     {"name": "ip address", "type": "field", "size": 4},
     {"name": "port", "type": "field", "size": 2}
 ]
}
}
```

8 References

BACS2017 Building Automation & Control Systems: An Investigation into Vulnerabilities, Current Practice & Security Management Best Practice -ASIS Foundation, Security Industry Association, Building Owners and Managers Association - 2017 -https://www.securityindustry.org/wp-content/uploads/2018/08/BACS-Report_Final-Intelligent-Building-Management-Systems.pdf

BRANDSTETTER2017 (in)security in building automation how to create

dark buildings with light speed - Thomas Brandstetter, Kerstin Reisinger - Presented at BlackHat USA 2017 -https://www.blackhat.com/docs/us-17/wednesday/us-17-Brandstetter-insecurity-In-Building-Automation-How-To-Create-Dark-Buildings-With-Light-Speed-wp.pdf

CWE656 CWE-656: Reliance on Security Through Obscurity - Common Weakness Enumeration (accessed on 2020-04-23) -https://cwe.mitre.org/data/definitions/656.html

FS2019 Cybersecurity in Building Automation Systems (BAS) - Daniel dos Santos, Clément Speybrouck, Elisa Costante (Forescout) - 2019 -https://www.forescout.com/places-in-network/building-automation-system-bas/

GASSER2017 Security Implications of Publicly Reachable Building Automation Systems - Oliver Gasser, Quirin Scheitle, Carl Denis, Nadja Schricker, Georg Carle - 2017

HUIYU2018 Hacking Intelligent Building - Pwning KNX & ZigBee Networks - HuiYu Wu, YuXiang Li (Tencent) - HITB Amsterdam 2018 -https://conference.hitb.org/hitbsecconf2018ams/materials/D1T2%20-%20YuXiang%20Li,%20HuiYu%20Wu%20&%20Yong%20Yang%20-%20Hacking%20Intelligent%20Buildings%20-%20Pwning%20KNX%20&%20ZigBee%20Networks.pdf

JURCZYK2016 Effective File Format Fuzzing - Thoughts, techniques and results - Mateusz "j00ru" Jurczyk - Presented at Blach Hat Europe 2016 -https://www.blackhat.com/docs/eu-16/materials/eu-16-Jurczyk-Effective-File-Format-Fuzzing-Thoughts-Techniques-And-Results.pdf

KAUR2015 Securing BACnet's Pitfalls - Jaspreet Kaur, Jernej Tonejc, Steen Wendzel, and Michael Meier - 2015

KNX03 KNX Standard v2.1 - 03 - System Specification KNX

KNX030603 KNX Standard v2.1 - 03.06.03 - System Specification -Standardised interfaces - External Message Interface

KNX030801 KNX Standard v2.1 - 03.08.01 - System Specification - KNXnet/IP -Overview

KNX030802 KNX Standard v2.1 - 03.08.01 - System Specification - KNXnet/IP -Core

KNX030803 KNX Standard v2.1 - 03.08.03 - System Specification - KNXnet/IP -Device management

LITVINOV2015 Security in KNX or how to steal a skyscraper - Egor Litvinov -Zero Nights 2015 -http://2015.zeronights.org/assets/files/20-Litvinov.pdf

MCKEE2019 HVACking: Understanding the Delta Between Security and Reality - Douglas McKee and Mark Bereza - Presented at Defcon 27, 2019

-https://www.mcafee.com/blogs/other-blogs/mcafee-labs/hvacking-understanding-the-delta-between-security-and-reality/

MIRSKY2017 HVACKer: Bridging the Air-Gap by Attacking the Air Conditioning System - Yisroel Mirsky, Mordechai Guri, and Yuval Elovici - 2017

PEACOCK2019 Anomaly Detection in BACnet/IP managed Building Automation Systems -Matthew Peacock - 2019 - https://ro.ecu.edu.au/theses/2178/

SRC_WSDISSECT Wireshark's KNXnet/IP dissector's source code on GitHub -https://github.com/wireshark/wireshark/blob/master/epan/dissectors/packet-knxip.c

TOOL_BACNETPY bacnet.py's GitHub page - https://github.com/tumi8/bacnet.py

TOOL_FUZZOWSKY Fuzzowsky's GitHub page - https://github.com/nccgroup/fuzzowski

TOOL_KNXMAP KNXmap's GitHub page - https://github.com/takeshixx/knxmap

TOOL_NMAP Nmap's documentation - https://nmap.org/nsedoc/index.html

TOOL_SCAPYBAC scapy-bacnet's GitHub page -https://github.com/desolat/scapy-bacnet/tree/master/scapy_bacnet