

UNIVERSITY OF TARTU
Institute of Computer Science
Informatics Curriculum

Uku Parts

Assessing the Security of a Smart Lock

Bachelor's Thesis (9 ECTS)

Supervisor(s):
Danielle Melissa Morgan

Tartu 2023

Assessing the Security of a Smart Lock

Abstract:

Smart locks are locking devices that make use of advanced technology. They often provide additional features compared to mechanical locks but these features may also provide opportunity for malicious actors to exploit software vulnerabilities in order to gain unauthorised access to the lock and thereby the user's home or belongings. Due to this danger, the security of smart locks must be assured. This thesis assesses the security of the Master Lock 4401EURLHEC, a smart lock. This is done by investigating the lock's communication protocol and the software the lock communicates with as well as carrying out a number of attacks against it. Though some minor vulnerabilities were found, ultimately it was concluded that the lock's security is fairly robust.

Keywords:

smart lock, penetration test, Bluetooth Low Energy

CERCS:

P175

Nutiluku turvalisuse hindamine

Lühikokkuvõte:

Nutilukud on lukustusseadmed, mis kasutavad elektroonilist tehnoloogiat. Tihti, nad pakuvad lisafunktsioone võrreldes mehaaniliste lukkudega, kuid need lisafunktsioonid võivad pakkuda pahatahtlikele osapooltele võimaluse tarkvara nõrkuste kuritarvitamise kaudu ligipääseda lukule ning seega kasutaja kodule või varale. Selle ohu tõttu, peab nutilukkude turvalisuses veenduma. Käesolev lõputöö hindab nutiluku Master Lock 4401EURLHEC turvalisust. Hindamise aluseks uuritakse luku kommunikatsiooni protokoll, analüüsitakse lukuga seotud mobiilirakendust ning viiakse läbi mitmeid rünnakuid luku vastu. Kuigi mõned väiksed turvavead leiti, jõuti järeldusele, et lukk on üsna turvaline.

Võtmesõnad:

nutilukk, läbistustestimine, Bluetooth Low Energy

CERCS:

P175

Table of Contents

Introduction.....	4
1 Background.....	5
1.1 Smart Locks.....	5
1.2 Bluetooth Low Energy.....	5
1.3 BLE Smart Lock Vulnerabilities.....	6
1.3.1 Replay Attack.....	6
1.3.2 Relay Attack.....	7
1.3.3 Fuzzing Attack.....	7
1.3.4 Battery Exhaustion Attack.....	7
1.3.5 Man-in-the-Middle Attack.....	7
1.3.6 Software Vulnerability Exploitation.....	8
1.4 The Master Lock 4401EURLHEC.....	8
2 Methodology and Results.....	10
2.1 Sniffing.....	10
2.2 Mobile Application Analysis.....	13
2.3 Attacks.....	14
2.3.1 Replay Attack.....	14
2.3.2 Fuzzing, Battery Exhaustion and Denial of Service Attacks.....	15
2.3.3 Man-in-the-Middle attack.....	15
2.3.4 Passcode Brute Forcing.....	18
3 Discussion.....	20
Conclusions.....	22
References.....	23

Introduction

Smart locks do not have a universal description but could be described as electronic locking devices that make use of advanced technology. However, this technology could also provide opportunities for malicious actors to exploit software vulnerabilities and poor design in order to gain unauthorised access to the lock and thereby the user's home or belongings. As a vulnerability in this kind of a device can cause the user significant material damage or even pose a threat to their safety, the security of these devices must be assured.

This thesis focuses specifically on vulnerabilities of smart locks using Bluetooth Low Energy (BLE), a wireless communication protocol [1]. Barua et al. [2] have written a comprehensive overview of how BLE works and what kinds of attacks it is vulnerable to. There have been several studies into the security of specific smart locks such as one written by Caballero-Gil et al. [3] about the security of the Nuki and Sherlock S2 smart locks. More such studies will be discussed in the Background chapter.

This thesis assesses the security of the Master Lock 4401EURLHEC, a device which has previously not been studied. The thesis investigates whether or not the lock is vulnerable to several common kinds of attacks specifically targeting the Bluetooth Low Energy technology used by the lock.

In chapter 1 the theoretical background necessary to understand the rest of the thesis will be given. Chapter 2 will go over the various methods used to gather information about the lock and test its ability to resist attacks. Chapter 2 will also cover the results of the information gathering and the tests. Chapter 3 will be a discussion about the results of the thesis and potential future work on the subject. The two appendices contain files generated during one attack.

1 Background

This chapter will provide the background information necessary to understand Bluetooth Low Energy smart locks and their vulnerabilities. First, the term “smart lock” will be defined then a description of Bluetooth Low Energy will be given and a number of ways smart locks can be vulnerable will be described. This chapter will also cover details about how the Master lock 4401EURLHEC operates.

1.1 Smart Locks

A smart lock is a locking device that can be operated electronically. There are many different methods that can be used to control a smart lock. For example, the U-Bolt Pro¹ can be unlocked with a smartphone app, a fingerprint reader, an Apple Watch app, a keypad or a physical key.

Some smart locks also provide additional features for the user’s convenience. For example, the U-Bolt Pro allows its owner to grant temporary access to guests and monitor who has unlocked the lock and when.

1.2 Bluetooth Low Energy

The following description of Bluetooth Low Energy (BLE) is based on an article by Barua et al. [2].

BLE is a wireless communication technology introduced in 2010 that is designed for IoT devices. It allows for short-range communication using very little power.

Like most other wireless communication protocols, BLE data is transmitted as packets. There are 2 types of BLE packets that are transmitted along 2 different types of channels. These are advertising and data packets.

The protocol that governs how BLE devices discover and connect to each other is the Generic Access Profile (GAP). Typically, a low power device (e.g. a smart lock) will transmit advertising packets and a higher power device (e.g. a smartphone) can receive these packets and initiate a connection with the low power device. The process of two devices establishing a connection is known as pairing. The initiator of the connection then takes on the role of the “master” and the device that is being connected to takes the role of the “slave”. After the connection is initialised, data packets can be sent between the master and the slave.

BLE supports encrypted communication. A BLE connection is secured by a “Security Manager”. This service handles the exchange of keys during pairing and the encryption and decryption of BLE data using AES (Advanced Encryption Standard [4]).

The communication protocol that lays the foundation of how BLE devices communicate is the Attribute Protocol (ATT). According to ATT, a device can either be a client or a server. A server stores information (in the form of “attributes”) and a client can request information from a server. A device can act as either a client or a server regardless of whether it’s a master or a slave. Attributes on the server are identified by a 16-bit handle and a 16-bit UUID.

The Generic Attribute Profile (GATT) is built on top of ATT. It defines a set of standard profiles, though it also allows the creation of custom profiles. A “profile” in this case is essentially a collection of attributes. GATT also defines certain read/write commands that BLE devices can use to communicate.

¹ <https://u-tec.com/products/ultraq-u-bolt-pro-series>

1.3 BLE Smart Lock Vulnerabilities

In this subchapter, various attacks possible against smart locks using BLE will be described.

1.3.1 Replay Attack

According to Barua et al. [2], a replay attack has 2 steps. The first step is eavesdropping on communication between 2 legitimate devices. The attacker must then capture packets of interest (e.g. unlock commands sent to a smart lock). The second step is retransmitting the captured packets in order to cause some kind of harm (e.g. unlock a smart lock without authorization). See Figure 1 for a diagram.

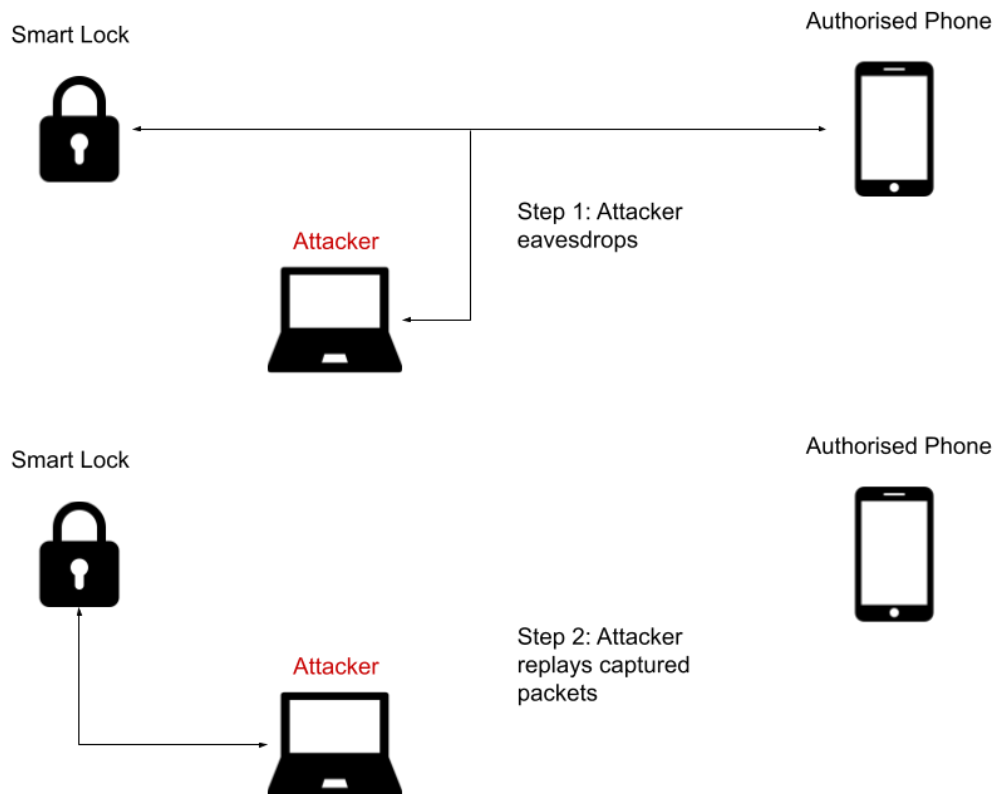


Figure 1: Replay attack

This attack was demonstrated against smart locks by Caballero-Gil et al. [3]. The researchers tested the attack against 2 different smart locks, both operated using a smartphone app. In both cases, they were able to easily eavesdrop on the communication between the lock and the phone.

They managed to successfully unlock their first target, the Sherlock S2, by resending the captured packets. However, the same attack was unsuccessful against their second target, the Nuki lock, which used a more sophisticated challenge-response system. Instead of there being a single unlock command, the lock first sent a random “challenge” to the phone, which the unlock instruction had to include. The attack failed because the captured unlock commands could not be reused since the challenge would not be the same and the commands also could not be modified since the packet was encrypted.

1.3.2 Relay Attack

According to Staat et al. [5], in a relay attack, a communication channel is opened between 2 devices that would otherwise not be close enough to communicate. This is done using a signal amplification device. If the communication channel needs to be bidirectional (as is the case for some smart locks) then 2 amplification devices are necessary. See Figure 2 for a diagram.

The goal of this is to trick the devices into thinking they are close together. This is relevant for smart locks (and other devices) that unlock when they detect that an authorised phone is nearby.

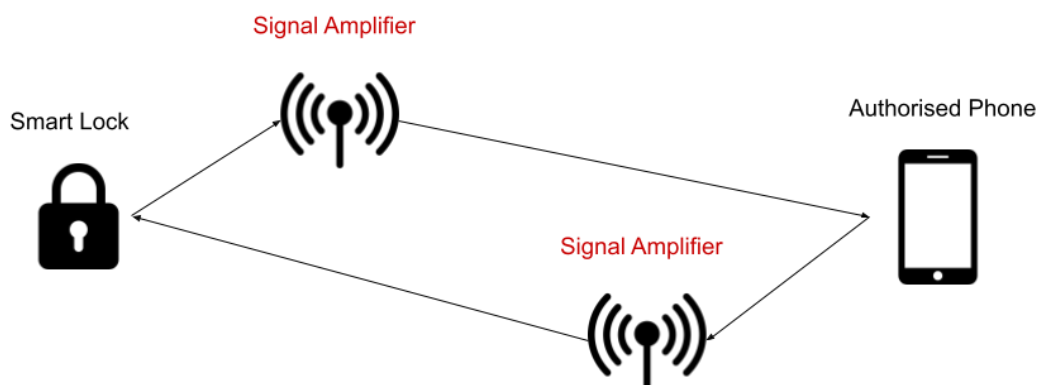


Figure 2: Bidirectional relay attack

Staat et al. [5] managed to construct a device capable of bidirectionally amplifying BLE signals in a way to accomplish just that task. As a result they managed to unlock a Nuki smart lock when the authorised phone was 65 metres away.

1.3.3 Fuzzing Attack

According to Barua et al. [2], a fuzzing attack involves sending malformed write requests to a BLE device. This can cause the device to crash or otherwise behave in some unexpected manner.

Rose and Ramsey [6] demonstrated this attack working against a smart lock. By sending malformed data to the lock, it was possible to make it crash in such a way that it opened without receiving a proper unlock command.

1.3.4 Battery Exhaustion Attack

According to Barua et al. [2], a battery exhaustion attack aims to drain a BLE device's battery in order to make it unusable. There are several ways of accomplishing this. One such method is constantly sending requests to the device, always keeping it active. Another is forcing the device to perform heavy computation.

1.3.5 Man-in-the-Middle Attack

According to Barua et al. [2], in a Man-in-the-Middle (MitM) attack the attacker spoofs a BLE device and gets a legitimate device to connect to the spoofed device (the fake peripheral). Messages sent to the spoofed device are then forwarded to the true recipient through a second attacker device (the fake central). See Figure 3 for a diagram. The attacker

can then easily eavesdrop on the communication and manipulate the data being sent between the devices, which was not possible with a relay attack.

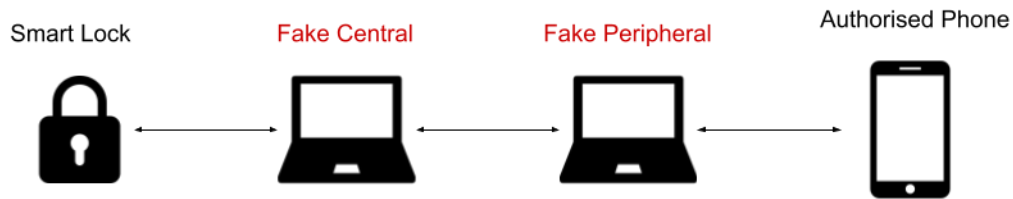


Figure 3: Man-in-the-Middle attack

1.3.6 Software Vulnerability Exploitation

Software vulnerabilities are not specifically related to BLE itself but they can still lead to BLE devices being compromised. It's possible to decompile android applications, so if there are any vulnerabilities in the code, attackers may be able to find them. By exploiting such vulnerabilities an attacker can gain unauthorised access to a lock operated with such an app.

For example, Rose and Ramsey [6] managed to decompile the source code of a smart lock control app and discovered that the lock used some hardcoded passwords. They were able to use this information in order to gain unauthorised access to the lock.

If the application communicates with some API then that is also a potential attack vector. For example, using a guest account, Knight et al. [7] were able to exploit a Master Lock API to generate temporary single-use codes that were valid for several years, even after the guest account's access had expired.

1.4 The Master Lock 4401EURLHEC

The Master Lock 4401EURLHEC is a smart lock made by the company Master Lock, which their product page claims is a Bluetooth padlock with “medium to high” security [8]. See Figure 4 for a picture.



Figure 4: The Master Lock 4401EURLHEC

There are 2 ways of operating this lock. Firstly, there is a 5-button keypad and a “primary code”. Secondly, there is a smartphone app that can be connected to the lock - Master Lock

Vault Home (there is also Master Lock Vault Enterprise, which serves the same purpose but is made for businesses, this app was not looked into in this thesis). The lock communicates with the app using BLE. It's normally in sleep mode and cannot communicate with the app at all. To activate the BLE capabilities of the lock, a button on the keypad must be pressed in order to get it to wake up. The lock's BLE communication has a range of about 10 metres.

The app can be in one of 3 different modes. In mode one, known as "Wake" mode, the lock will automatically unlock once it has been woken up and an authorised phone is nearby. In mode two, known as "Wake + Tap" mode, the user must wake the lock, open the app on their phone and tap an "unlock" button in order to unlock the lock. In mode three, known as "Disabled" mode, the lock can only be unlocked using the keypad. The keypad has 5 buttons, a centre button and 4 directional keys.

The lock comes with a random default 7 character primary code that can be used to unlock the lock if it is entered into the keypad. This code can be viewed and changed using the app. If the code is changed, it will take effect the next time the phone establishes a connection with the lock, not immediately. It has a minimum length of 7 characters and a maximum length of 13 characters.

The app also allows for the generation of temporary codes. These generated codes are valid on a specific date during set 8 hour intervals. The temporary codes are generated by some algorithm based on the date, interval and presumably the lock's encryption key. That means that the codes are deterministic and they don't require a BLE connection with the lock to work.

Despite the keypad having 5 buttons, the codes only use 4 of them. The primary code must start with ↑ (the app does not allow a code starting with any other character to be set) and all temporary codes seem to start with →. An example of a primary code would be ↑←→↓→↓→.

The app also allows giving access to guests. Guest access can be restricted in various ways. They may only be able to unlock the lock during certain hours of the day or only using temporary codes, for example. A guest will also need to download the same app to accept an invite.

Various other things can be configured using the app as well. For example, the amount of time before the lock automatically relocks itself and whether or not to send notifications whenever the lock is unlocked. The same screen shows various information about the lock such as its model and last known location. The app also logs events such as Bluetooth unlock events and temporary code viewings along with the associated person and the date and time.

2 Methodology and Results

In this chapter methods used to test the lock's security and the results of these tests will be described. The chapter is split into 3 sections. First, the communication protocol of the lock is analysed and described. Second, the lock's mobile application is analysed. Third, various attacks against the lock and their results are described.

2.1 Sniffing

In order to assess whether or not the smart lock is secure it's necessary to understand how the lock communicates with an authorised phone. One method that can be used to achieve this understanding is eavesdropping on the communication between the devices.

To this end, the Nordic Semiconductor nRF 52840 dongle was used. The nRF 52840 dongle is a small programmable USB device that supports BLE as well as a variety of other wireless communication protocols [9]. See Figure 5 for a picture. The device can fulfil various functions depending on the firmware it is programmed with. In this case, the nRF Sniffer for Bluetooth LE firmware² was appropriate. Using this firmware, the device captures BLE packets in real time and displays them in Wireshark³.

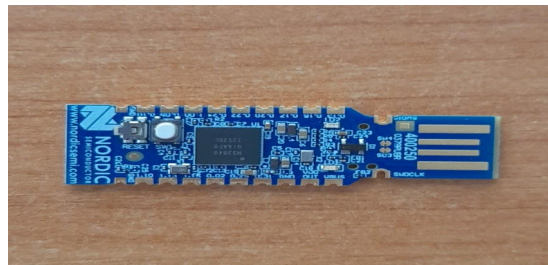


Figure 5: The nRF 52840 dongle

The following steps were performed in order to analyse the BLE traffic:

- 1) The firmware was installed onto the nRF 52840 dongle and Wireshark was configured according to the user guide⁴.
- 2) The app was opened and set to Wake + Tap mode.
- 3) The app was closed.
- 4) Wireshark was opened and the “nRF Sniffer for Bluetooth LE” interface was selected.
- 5) A button was pressed on the lock's keypad in order to get it to wake up and start transmitting packets.
- 6) The device with the name “Master Lock” was selected in Wireshark's Device tab in order to configure the nRF to only capture packets from the lock.
- 7) The app was opened in order to get it to connect to the app.
- 8) The unlock button was pressed in the app.
- 9) After the lock had relocked itself, the capturing was stopped.
- 10) The results were then either saved to a .pcap file to be analysed later or analysed immediately using Wireshark.

Variations of steps 2-10 were then performed:

- The app was switched to Wake mode.
- The app was switched to Disabled mode.

² <https://www.nordicsemi.com/Products/Development-tools/nRF-Sniffer-for-Bluetooth-LE>

³ <https://www.wireshark.org/>

⁴ https://docs.nordicsemi.com/bundle/nrfutil_ble_sniffer_pdf/resource/nRF_Sniffer_BLE_UG_v4.0.0.pdf

- The unlock command was not sent.

All of these experiments were repeated several times in order to verify that the results were consistent. The following is a description of how the communication between the lock and the phone functions.

The lock does not transmit any packets when it is in sleep mode. When a button is pressed on the keypad and the lock wakes up, it starts transmitting advertising packets. The contents of one such advertising packet can be seen in Figure 6.

```

▼ Bluetooth Low Energy Link Layer
  Access Address: 0x8e89bed6
  > Packet Header: 0x1b00 (PDU Type: ADV_IND, ChSel: #1, TxAdd: Public)
    Advertising Address: TexasInstrum_bd:1a:fe (6c:79:b8:bd:1a:fe)
  ▼ Advertising Data
    > Flags
    ▼ 128-bit Service Class UUIDs
      Length: 17
      Type: 128-bit Service Class UUIDs (0x07)
      Custom UUID: 94e00001-5d5b-11e4-846f-4437e6b36dfb (Unknown)
    CRC: 0xbdfeba

```

Figure 6: Contents of an advertising packet in Wireshark

The app detects these advertising packets (even if it is running in the background and even if it is set to Disabled mode) and establishes a connection to the lock. After the connection is established the phone sends a sequence of requests that are the same every time the phone connects to the lock. Specifically three read by group type, nine read by type, one find information and two write requests. See Figures 7, 8, 9 and 10 respectively for examples of the contents of these requests.

```

▼ Bluetooth Attribute Protocol
  > Opcode: Read By Group Type Request (0x10)
    Starting Handle: 0x0001
    Ending Handle: 0xffff
    UUID: Primary Service (0x2800)

```

Figure 7: Contents of a Read By Group Type request in Wireshark

```

▼ Bluetooth Attribute Protocol
  > Opcode: Read By Type Request (0x08)
    Starting Handle: 0x0001
    Ending Handle: 0x0007
    UUID: Include (0x2802)

```

Figure 8: Contents of a Read By Type request in Wireshark

- ▼ Bluetooth Attribute Protocol
 - Opcode: Find Information Request (0x04)
 - Starting Handle: 0x000e
 - Ending Handle: 0x000e

Figure 9: Contents of a Find Information request in Wireshark

- ▼ Bluetooth Attribute Protocol
 - Opcode: Write Request (0x12)
 - ▼ Handle: 0x000e (Unknown: Unknown: Client Characteristic Configuration)
 - [Service UUID: 94e000015d5b11e4846f4437e6b36dfb]
 - [Characteristic UUID: 94e000025d5b11e4846f4437e6b36dfb]
 - [UUID: Client Characteristic Configuration (0x2902)]
 - ▼ Characteristic Configuration Client: 0x0001, Notification
 - 0000 0000 0000 00.. = Reserved: 0x0000
 -0. = Indication: False
 -1 = Notification: True

Figure 10: Contents of an initialization sequence Write request in Wireshark

After the final request in the previously described sequence, the devices enter a loop where the lock sends a notification with a random value (see Figure 11 for an example) and the lock responds with a write request containing a seemingly random value (see Figure 12 for an example).

- ▼ Bluetooth Attribute Protocol
 - Opcode: Handle Value Notification (0x1b)
 - ▼ Handle: 0x000d (Unknown: Unknown)
 - [Service UUID: 94e000015d5b11e4846f4437e6b36dfb]
 - [UUID: 94e000025d5b11e4846f4437e6b36dfb]
 - Value: 0000000001959100015b95e099576c20b0b0

Figure 11: Contents of a challenge in Wireshark

- ▼ Bluetooth Attribute Protocol
 - Opcode: Write Request (0x12)
 - ▼ Handle: 0x000d (Unknown: Unknown)
 - [Service UUID: 94e000015d5b11e4846f4437e6b36dfb]
 - [UUID: 94e000025d5b11e4846f4437e6b36dfb]
 - Value: 0100053174e15b66c6348ee900e6e9d5

Figure 12: Contents of a response to a challenge in Wireshark

This is likely a challenge-response system where the lock sends a random value to the phone and it responds with an appropriate write request. The value of this write request depends on the value of the notification and the command being sent to the lock. Due to this, there is no single unlock command, for example. These notification-write pairs are sent constantly even if no command is being executed, meaning the phone constantly sends keep-alive requests in order to maintain the connection.

The values in the write requests do have a pattern. While most of the digits appear to be random, the first 6 digits of the values seem to follow some scheme. In all captured write requests of this kind the first five digits were always 01000 and the sixth digit determined the length of the value. It is possible that the sixth digit determines the type of the command being sent to the lock, however this is not consistent with the fact that values with various sixth digits were being sent even when the phone should only have been sending keep-alive requests.

The connection is maintained until either an unlock command is sent or about five seconds have passed, at which point the lock terminates the connection and the whole process repeats starting with the lock sending advertising packets, which the phone detects again after about half a second. This continues until the lock goes back to sleep, which happens after about thirty seconds.

2.2 Mobile Application Analysis

Analysing the source code of the lock and the mobile application can also lead to better understanding of how the devices communicate or reveal exploitable vulnerabilities. The lock's software is not public and getting it directly from the lock is difficult. The mobile application however, is public. For Android devices, apps are in the form of APK files [10]. APKPure⁵ was used to download this apk file (Master Lock Vault Home version 1.11.0.14).

Several tools exist for automatically detecting vulnerabilities in Android applications. One of these is MobSF, which is a framework capable of static APK analysis. Running the downloaded apk through MobSF's static analyser on their website⁶, it got a security score of 55/100. There were no major security issues detected, however the app did receive several warnings. After manual review, it was concluded that none of the detected issues were serious. The analyser also reported that several hardcoded secrets were found, the seemingly most important of which was a Google API key.

To get human readable code from apk files, they must be decompiled back to Java code. Jadx was used for this purpose. Jadx is a piece of software for getting Java code out of APK files [11]. When an APK file is opened in jadx's GUI it is automatically decompiled into Java code, which can then be browsed. Most of the source code decompiled without issues, however it did have trouble with some sections of the code, which it failed to decompile.

The code was heavily obfuscated with many classes, methods and variables randomly renamed and new randomly generated code being added. The project contained 7820 java files, most of which were not really used by the application while also not being dead code. Getting any meaningful information out of the code like this was deemed to be too time-consuming so the idea was abandoned.

⁵ <https://apkpure.com/>

⁶ <https://mobsf.live/>

2.3 Attacks

In this section, attempted attacks (replay, fuzzing, battery exhaustion, denial of service and man-in-the-middle) and their results will be described.

2.3.1 Replay Attack

The goal of this attack was to resend some of the captured packets in order to cause the lock to unlock without proper authorisation.

This attack was carried out on a Dragon OS⁷ virtual machine using an Asus BT-400 Bluetooth adapter⁸ and the nRF 52840 dongle. The BT-400 was used to communicate with the lock and the nRF 52840 was used to monitor packets being sent to and from the lock using the same setup as described in the sniffing section. Gatttool⁹, a command line utility for sending BLE requests, was used to send requests to the lock.

The handle and value of the write request that unlocks the lock was taken from the captured communication from the sniffing section, along with the MAC address of the lock. This information can then be used to send an identical write request to the lock using gatttool. An example of such a command can be seen in Listing 1. The argument -b is used to specify the MAC address of the device the command is being sent to, -a specifies the handle and -n the value.

```
gatttool -b 6c:79:b8:bd:1a:fe --char-write-req -a 0x000d -n
0100021c22d1f90681c7acbb00
```

Listing 1: Example gatttool write request

This had no effect, however. The lock acknowledged the write request by sending a write response but it did not unlock. In order to get the lock to respond in any meaningful way, the two write requests in the initialisation sequence must first be sent.

Since the delay between sequential requests must not be too long in order for the connection between the BT-400 and the lock to not be terminated, a bash script was used to string multiple gatttool commands together. An example of such a script can be seen in Listing 2. The first 2 gatttool commands send write requests identical to the ones in the real initialisation sequence. The third command sends an unlock request.

```
gatttool -b 6c:79:b8:bd:1a:fe --char-write-req -a 0x000e -n 0100
gatttool -b 6c:79:b8:bd:1a:fe --char-write-req -a 0x000d -n
0080b27e4ff5a30036eb7018a2af3f78987a251926c8df8461c6c2b323f5dc234f
2b7068b74cbb06b0f8c0d73fc9d417c60ae7c084170eab3c264fdbd3c5633cd08f
539774d8d9
gatttool -b 6c:79:b8:bd:1a:fe --char-write-req -a 0x000d -n
0100021c22d1f90681c7acbb00
```

Listing 2: Example replay attack script

After the first two write requests are finished, the lock responds with a notification showing that it is responsive and expecting a write request with an appropriate value. However,

⁷ <https://cemaxecuter.com/>

⁸ <https://www.asus.com/networking-iot-servers/adapters/all-series/usbbt400/>

⁹ <https://manpages.debian.org/unstable/bluez/gatttool.1.en.html>

because the notification's value is random, it is exceedingly unlikely for the sniffed communications to contain a notification with the same value. As such, it is not possible to send a write request with an appropriate value as the appropriate value for this notification is unknown. Sending the same unlock value as before still has no effect. The lock acknowledges the write request but it does not open and it does not send a second notification. Sending the full captured communication sequence (including keep-alive packets) also has no effect.

Based on this it can be concluded that a replay attack is not possible due to the challenge-response system used by the lock.

2.3.2 Fuzzing, Battery Exhaustion and Denial of Service Attacks

These attacks attempt to open the lock or to disrupt its operations by sending it malformed data. Specifically, the attacks would ideally have any of the following effects:

- Cause the lock's software to crash and the lock to open.
- Keep the lock awake for long periods of time, draining its battery.
- Disrupt the lock's communication with an authorised phone.

The equipment and software used is the same as for the replay attack. Instead of a bash script, python was used to repeatedly execute gatttool commands. The main loop of such a script can be seen in Listing 3. It sends the same write request 100 times.

```
for _ in range(100):  
    system(f"gatttool -b 6c:79:b8:bd:1a:fe --char-write-req -a  
0x000d -n 0100")
```

Listing 3: Python loop for repeatedly sending write requests

Repeatedly sending the lock a single random write request did somewhat disrupt its ability to communicate with an authorised phone. While the script was running, most unlock commands given by the phone were not successful. However, some unlock commands did get through so this disruption was not 100% effective.

The lock also did not stay awake while receiving requests. It would still go to sleep after about 30 seconds.

Receiving write requests like this also did not seem to cause the lock to malfunction or crash in any way. Likely, because it disregards any write requests made before it receives the initialisation sequence.

Sending the initialisation sequence before any other write requests had no effect in terms of battery exhaustion and denial of service. In terms of fuzzing, it did seem a bit more promising. After the two write requests in the initialisation sequence, the first malformed write seems to cause the lock to enter some kind of an error state where all further writes fail for the remainder of the connection. However, there is no clear way of exploiting this. Ultimately, no matter what kind of malformed data was sent to the lock it did not open without authorisation and otherwise continued to function as expected.

Based on this it can be concluded that the lock handles malformed data in a robust manner and cannot be exploited in this way.

2.3.3 Man-in-the-Middle attack

The goal of this attack is to trick the lock and the phone into thinking they are nearby. This is the same as a relay attack, however since the attack doesn't involve amplifying physical

signals, it should have greater range and require less specialised equipment. If the app is in Wake mode then opening a communication channel between it and the lock, it may be possible to open the lock without authorisation.

This attack was carried out on two Kali Linux¹⁰ virtual machines (hosted on two separate physical machines) using two Asus BT-400 Bluetooth adapters and the nRF 52840 dongle. The BT-400s were used to communicate with the lock and an authorised phone and the nRF 52840 was used to monitor the communication.

The tool used for this attack was gattacker. Gattacker is a node.js package capable of scanning for BLE devices, finding out information about those devices and carrying out Man-in-the-Middle attacks [12].

Due to the fact that gattacker is quite out of date (the GitHub repository has not been updated in 6 years), some modifications were needed to get the software running. The repository was cloned and the following changes were made:

- 1) The bluetooth-hci-socket dependency was replaced with the latest version of @abandonware/bluetooth-hci-socket¹¹.
- 2) The noble and bleno directories in the lib directory were replaced with the abandonware noble¹² and bleno¹³ versions.
- 3) In ws-slave.js, line 43 was changed from state: noble.state to state: "poweredOn".

Changes 1 and 2 are due to the named dependencies also largely being abandoned and no longer working reliably on modern systems. Change 3 is due to noble.state returning “unknown” even if the true state was “poweredOn” in that specific spot, likely because of a bug. All of these changes were necessary to get gattacker working at a baseline level. Without them, the application either failed to install or start. Other than these changes, the normal gattacker installation and configuration guide found on their GitHub page was followed.

The gattacker setup consists of two devices: a fake central and fake peripheral. The fake central was configured by uncommenting the configuration line starting with NOBLE_HCI_DEVICE_ID. The virtual machine running the fake central was assigned a static IP address to make it reachable on the local network. The fake peripheral was configured by uncommenting the configuration line starting with BLENO_HCI_DEVICE_ID, the WS_SLAVE value was set to the IP address assigned to the fake central and the BLENO_ADVERTISING_INTERVAL value was set to 200 so it is close to the real lock’s advertising interval.

After the two gattacker instances were configured, the attack could begin. The ws-slave.js script was started on the fake central. The command to do so is in Listing 4.

```
sudo node ws-slave.js
```

Listing 4: Command that starts the application on the fake central

This opens a network port on the fake central so the two fake devices are able to establish a websocket connection. This websocket connection allows the fake devices to send data to each other over an internet connection (or local network connection, as was the case here).

¹⁰ <https://www.kali.org/>

¹¹ <https://github.com/abandonware/node-bluetooth-hci-socket>

¹² <https://github.com/abandonware/noble>

¹³ <https://github.com/abandonware/bleno>

The first step of the attack was gathering information about the lock in order to spoof it. With gattacker, this is done by first scanning for all nearby devices, which saves the advertising data for all of them, and then by scanning a specific device, which saves the device's service data. The commands in Listing 5 were run on the fake peripheral.

```
sudo node scan
sudo node scan -o 6C:79:B8:BD:1A:FE
```

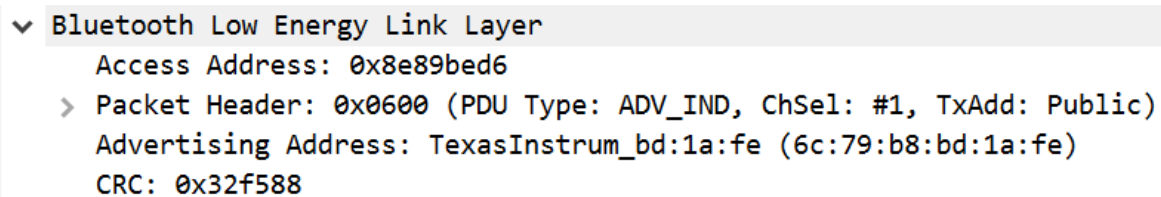
Listing 5: Commands to gather information on the target device

This creates two JSON files containing the lock's advertising (Appendix 1) and service (Appendix 2) data. These files can now be used to spoof the peripheral. At this point, the devices were put in separate rooms with enough distance between them for BLE signals from one to not reach the other (this was verified using both the nRF 52840 and the BLE Scanner app¹⁴). The lock was near the fake central and the phone was near the fake peripheral. The Master Lock app was set to Wake mode. A button was pressed on the lock to wake it up and the command in Listing 6 was run on the fake peripheral.

```
sudo ./mac_adv -a devices/6c79b8bd1afe_Master-Lock.adv.json -s
6c79b8bd1afe.srv.json
```

Listing 6: Command to spoof the lock and start advertising

However, this did not work. The command ran successfully and the fake peripheral started advertising but the phone did not connect to it. Diagnosing the issue by capturing the fake peripheral's advertising packets using the nRF 52840, it seems like the advertising packets are missing all advertising data. Compare Figures 6 (real advertising packet) and 13 (fake advertising packet). The Access Address and Advertising Address of the fake packet are the same as the real one but the Advertising Data is missing (the Packet Header is different because the length of the packet is different and the Cyclic Redundancy Check is different because the contents are different). Note that the advertising JSON file (Appendix 1) contains the correct advertising data.



```
▼ Bluetooth Low Energy Link Layer
  Access Address: 0x8e89bed6
  > Packet Header: 0x0600 (PDU Type: ADV_IND, ChSel: #1, TxAdd: Public)
    Advertising Address: TexasInstrum_bd:1a:fe (6c:79:b8:bd:1a:fe)
    CRC: 0x32f588
```

Figure 13: Contents of an advertising packet sent by gattacker in Wireshark

Furthermore, the services of the fake peripheral are wrong. The three correct ones are there but there are also two additional ones. This can be seen using the BLE Scanner app. See Figure 14 for a list of the real lock's services and Figure 15 for a list of the fake lock's services. Again, note that the services JSON file (Appendix 2) correctly contains the 3 correct services (and only those 3).

¹⁴ <https://play.google.com/store/apps/details?id=com.macdom.ble.blescanner&hl=en&gl=US>

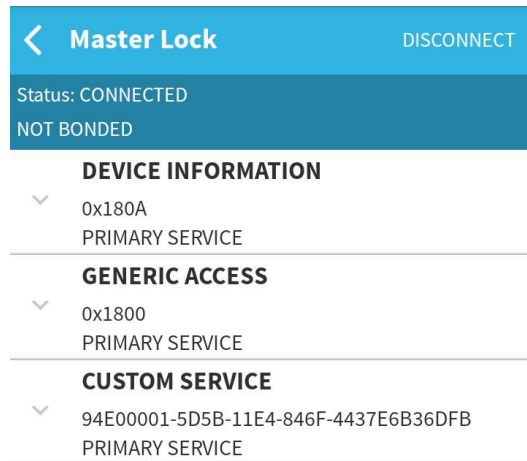


Figure 14: Services of the real lock in BLE Scanner

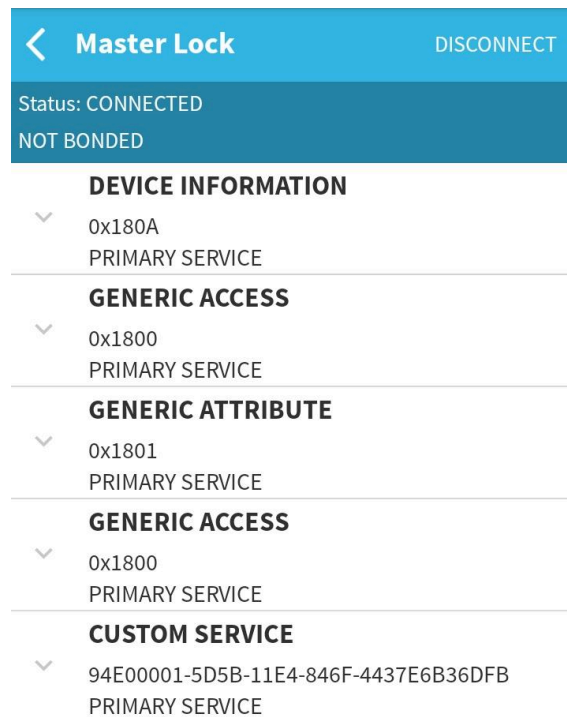


Figure 15: Services of the spoofed lock in BLE Scanner

It's likely that the technical issues were caused by the combination of the software being outdated and my modifications. As a result, the lock's ability to resist MitM attacks could not be verified.

2.3.4 Passcode Brute Forcing

Due to the low number of symbols used in passcodes as well as their short length, a brute force attack may be feasible. There are 4 symbols (\leftarrow , \uparrow , \rightarrow and \downarrow) used in the codes and the minimum passcode length is 7 characters. That means that there would be $4^7 = 16384$ possible passcodes of the minimum length. However, the first character of the primary code must always be \uparrow , so the actual number of possible primary codes of the minimum length is only $4^6 = 4096$.

In addition, there's always an active temporary code that is 8 characters long. The first character of these seems to always be \rightarrow . This means that there are $4^7 = 16384$ possible temporary codes.

When an incorrect code is entered into the keypad the lock won't open. The lock will not immediately show that the entered code was wrong, it will wait until a code of the maximum allowed length for the type is entered (13 characters if the first character was \uparrow , 8 if it was \rightarrow). At that point, it will show a red indicator. This means that the lock does not give away information about the length of the primary code.

If three wrong codes are entered into the lock in quick succession, then the lock will not accept any more attempts for about one minute. However, assuming an attacker tries three codes per minute, it would take them just $4096 / 3 = 1365.33$ minutes or about 22.75 hours in order to try all possible passcodes and open the lock. It's also very unlikely that the last code the attacker tries is the correct one. Just by chance, they will probably get the correct code before then.

Therefore, it's realistically possible for an attacker to brute force the primary code of this lock with 1 or 2 days of work, assuming the code is of the minimum possible length. If the code is even slightly longer, then this kind of an attack quickly becomes infeasible as the amount of time it would take increases exponentially.

An attacker can also attempt to guess the temporary code. Guessing all possible temporary codes would take $16384 / 3 = 5461.33$ minutes or about 91 hours. Though this code changes constantly so an attacker trying to guess this code could not be sure that the code hasn't already changed to one they tried before (i.e. it is impossible to try all possible codes in the timeframe where one code is active, which is 8 hours).

3 Discussion

In this chapter the results and areas of future research will be discussed.

In total the lock was tested against 5 different attacks. Of these, only 1 was partially successful and 1 attack failed due to technical issues. See Table 1 for a list of the attacks and their results.

Table 1: Results of the attacks

Attack	Result
Replay	Unsuccessful
Fuzzing	Unsuccessful
Battery Exhaustion	Unsuccessful
Denial of Service	Partially Successful
Man-in-the-Middle	Failed due to technical issues

Based on the captured communication between the lock and an authorised phone and the result of the replay attack, it can be concluded that the lock uses a robust challenge-response system in order to securely receive commands from authorised devices. This is similar to the Nuki lock in the study by Caballero-Gil et al. [3], which was also found to be impervious to replay attacks due to a challenge-response system.

The denial of service attack was somewhat effective in disrupting the communication between the lock and an authorised phone. This attack was not 100% effective as some commands did eventually get through. However, this attack alone could not cause any serious disruption even with higher effectiveness. As the use of temporary codes is not disrupted, it is easy for an authorised user to unlock the lock if it is completely unable to communicate with BLE. Due to this, the partially successful denial of service attack should not be considered a major issue.

A denial of service attack targeting the user's phone may be more effective. If the phone is prevented from using BLE and the user cannot open the app then in order to unlock the lock they would need to remember the primary code. If the user does not use the primary code often (which they almost certainly would not) then they likely would not remember it and would be unable to open the lock.

The lock is vulnerable to passcode brute forcing attacks. However, this vulnerability is not easy for an attacker to exploit. Even in the worst case scenario where the primary code is of the minimum length, an attacker would need to manually enter codes into the lock for many hours in order to unlock it (assuming the code is not easy to guess). Additionally, since the lock does not give away information about the true length of the code, an attacker cannot know if they are just wasting their time.

Assuming the primary code is well-chosen and cannot be brute forced, an attacker could still attempt to brute force a temporary code. Since these codes are 8 characters long (only 7 of which are random) they can be guessed. However, since the codes are longer and also change constantly, this attack is more difficult and relies on luck as the space of possible codes cannot be exhaustively searched in the timeframe where one code is active.

The attempted MitM attack should, in theory, be a superior form of a relay attack. A relay attack in the physical layer has significant limitations. For example, the relay attack performed by Staat et al. [5] had a fairly short range and the equipment to perform it cost 2200 euro. A MitM attack could achieve the same result but with significantly cheaper equipment and superior range.

In this case, the attack failed due to technical issues with gattacker. The issues could not be resolved and a suitable alternative could not be found due to time constraints so the effectiveness of this kind of a MitM attack could not be verified. In the future, a modern tool for testing MitM attacks against BLE devices should be developed (or an existing one, such as gattacker, modernised). At that point, the lock's ability to resist MitM attacks could be re-evaluated.

The “random” values used in the challenge-response system did show some patterns. It may be worth taking a deeper look into the matter in order to see if these patterns can be exploited in some way. If the values could somehow be predicted then that may allow the challenge-response system to be subverted and enable replay attacks, for example.

Conclusions

In this thesis, the communication protocol of the lock and the source code of the lock's mobile application were analysed. A total of five kinds of attacks were carried out against the lock and one attack was described.

Overall, the lock was well secured. It uses a robust challenge-response system for Bluetooth Low Energy communication and its mobile application is obfuscated and contains no easily findable vulnerabilities or hard coded secrets. The lock successfully resisted replay, fuzzing and battery exhaustion attacks. The lock is susceptible to denial of service attacks to a degree. However these cannot cause significant disruption thanks to the lock's easily usable unlock code system, which allows the lock to operate even if its Bluetooth Low Energy capabilities are non-functional.

The lock is considerably weak to passcode brute forcing attacks. A determined attacker may be able to brute force the primary code of a lock in a matter of hours assuming it is of the minimum length. Even if the primary code is longer, an attacker may still be able to brute force a temporary code though this is significantly more time-consuming. A lock with a well-chosen primary code is still reasonably secure against brute forcing attacks.

Future research could assess the lock's susceptibility to Man-in-the-Middle attacks and take a deeper look into the random numbers used by the challenge-response system.

References

- [1] Intro to Bluetooth Low Energy | Bluetooth® Technology Website n.d.
<https://www.bluetooth.com/bluetooth-resources/intro-to-bluetooth-low-energy/> (accessed December 4, 2023).
- [2] Barua A, Al Alamin MA, Hossain MdS, Hossain E. Security and Privacy Threats for Bluetooth Low Energy in IoT and Wearable Devices: A Comprehensive Survey. *IEEE Open Journal of the Communications Society* 2022;3:251–81.
<https://doi.org/10.1109/OJCOMS.2022.3149732>.
- [3] Caballero-Gil C, Álvarez R, Hernández-Goya C, Molina-Gil J. Research on smart-locks cybersecurity and vulnerabilities. *Wireless Netw* 2023.
<https://doi.org/10.1007/s11276-023-03376-8>.
- [4] Dworkin MJ. Advanced Encryption Standard (AES). Gaithersburg, MD: National Institute of Standards and Technology (U.S.); 2023.
<https://doi.org/10.6028/NIST.FIPS.197-upd1>.
- [5] Staat P, Jansen K, Zenger C, Elders-Boll H, Paar C. Analog Physical-Layer Relay Attacks with Application to Bluetooth and Phase-Based Ranging. *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, New York, NY, USA: Association for Computing Machinery; 2022, p. 60–72.
<https://doi.org/10.1145/3507657.3528536>.
- [6] Rose A, Ramsey B. Picking Bluetooth Low Energy Locks from a Quarter Mile Away 2016.
<https://doi.org/10.5446/36217>.
- [7] Knight E, Lord S, Arief B. Lock Picking in the Era of Internet of Things. 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), 2019, p. 835–42.
<https://doi.org/10.1109/TrustCom/BigDataSE.2019.00121>.
- [8] Model No. 4401EURLHEC | Master Lock n.d.
<https://www.masterlock.eu/home-personal/product/4401EURLHEC> (accessed May 15, 2024).
- [9] nRF52840 Dongle n.d.
<https://www.nordicsemi.com/Products/Development-hardware/nRF52840-Dongle> (accessed May 11, 2024).
- [10] Application fundamentals. Android Developers n.d.
<https://developer.android.com/guide/components/fundamentals> (accessed May 15, 2024).
- [11] skylot. skylot/jadx 2024. <https://github.com/skylot/jadx> (accessed May 15, 2024).
- [12] securing. securing/gattacker 2024. <https://github.com/securing/gattacker> (accessed May 14, 2024).

Appendix

I. Contents of 6c79b8bd1afe_Master-Lock.adv.json

```
{
  "id": "6c79b8bd1afe",
  "eir": "",
  "scanResponse": null,
  "decodedNonEditable": {
    "localName": "Master Lock",
    "manufacturerDataHex": "4b019b8f000005424d2400008fe3e85a",
    "manufacturerDataAscii": "K      BM$      Z",
    "serviceUids": [
      "94e000015d5b11e4846f4437e6b36dfb"
    ]
  }
}
```


II. Contents of 6c79b8bd1afe.srv.json

```
[
  {
    "uuid": "1800",
    "name": "Generic Access",
    "type": "org.bluetooth.service.generic_access",
    "startHandle": 1,
    "endHandle": 7,
    "characteristics": [
      {
        "uuid": "2a00",
        "name": "Device Name",
        "properties": [
          "read"
        ],
        "value": "4d6173746572204c6f636b",
        "descriptors": [],
        "startHandle": 2,
        "valueHandle": 3,
        "asciiValue": "Master Lock"
      },
      {
        "uuid": "2a01",
        "name": "Appearance",
        "properties": [
          "read"
        ],
        "value": "0000",
        "descriptors": [],
        "startHandle": 4,
        "valueHandle": 5,
        "asciiValue": " "
      },
      {
        "uuid": "2a04",
        "name": "Peripheral Preferred Connection
Parameters",
        "properties": [
          "read"
        ],
        "value": "1000200000002c01",
        "descriptors": [],
```

```

        "startHandle": 6,
        "valueHandle": 7,
        "asciiValue": "      , "
    }
]
},
{
    "uuid": "180a",
    "name": "Device Information",
    "type": "org.bluetooth.service.device_information",
    "startHandle": 8,
    "endHandle": 10,
    "characteristics": [
        {
            "uuid": "2a29",
            "name": "Manufacturer Name String",
            "properties": [
                "read"
            ],
            "value": "4d6173746572204c6f636b",
            "descriptors": [],
            "startHandle": 9,
            "valueHandle": 10,
            "asciiValue": "Master Lock"
        }
    ]
},
{
    "uuid": "94e000015d5b11e4846f4437e6b36dfb",
    "name": null,
    "type": null,
    "startHandle": 11,
    "endHandle": 14,
    "characteristics": [
        {
            "uuid": "94e000025d5b11e4846f4437e6b36dfb",
            "name": null,
            "properties": [
                "read",
                "writeWithoutResponse",
                "write",
                "notify"
            ]
        }
    ]
}

```

```

],
"value": "",
"descriptors": [
  {
    "handle": 14,
    "uuid": "2902",
    "value": ""
  }
],
"startHandle": 12,
"valueHandle": 13,
"asciiValue": ""
}
]
]

```

I. License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Uku Parts,

1. grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis

Assessing the Security of a Smart Lock,

supervised by Danielle Melissa Morgan.

2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in points 1 and 2.
4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Uku Parts
15/05/2024