MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Analysis of security features of smart lock protocol

Bachelor's Thesis

PETR HANÁK

Brno, Spring 2024

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Analysis of security features of smart lock protocol

Bachelor's Thesis

PETR HANÁK

Advisor: doc. RNDr. Petr Švenda, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2024

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source. During the preparation of this thesis, I used Grammarly for grammar check. I declare that I used this tool in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Petr Hanák

Advisor: doc. RNDr. Petr Švenda, Ph.D.

Acknowledgements

I would like to thank

my advisor doc. RNDr. Petr Švenda, Ph.D., for the professional guidance and patience,

MSc. Arnis Paršovs, Ph.D. from the University of Tartu, for the initial idea for the thesis, for a great amount of help the whole time and for arranging the purchase of the hardware,

doc. RNDr. Zdeněk Matěj, Ph.D., for help with repairing the hardware,

and my family and my girlfriend for the support.

Abstract

The thesis analyses the design and security of the Noke HD smart lock. In the design analysis, we reverse-engineer the lock's protocol because its documentation is not publicly available. We further analyse its design elements. In the security analysis, we assess the protocol's security against common attacks on authentication protocols and other attacks on smart locks. The result of the thesis is a description of the padlock's protocol and its security analysis.

Keywords

smart lock, security analysis, design analysis, reverse-engineering, security protocol

Contents

List of Tables

List of Figures

This thesis is supported by the European Union under Grant Agreement No. 101087529.

Introduction

Smart technologies and the Internet of Things (IoT) have become increasingly popular in recent years. Companies are constructing various "smart" devices ranging from phones to lightbulbs, scooters and refrigerators. The manufacturers and users then form the IoT from the smart devices. They connect these devices to networks, allow them to communicate and make them available over the internet. The security aspect of the smart devices is often neglected, which leads to IoT being commonly listed as one of the top security risks [1], [2], [3].
In this work, we examine a smart device which has security as the main selling point - the Noke HD smart padlock [4]. Complete documentation for the padlock's protocol is not publicly available, nor is any security analysis. This thesis aims to make this information available to the company's customers and security experts. The focus is not solely on the padlock but on the whole padlock's platform, which includes a mobile app and an API of the backend servers.
In the theoretical part of the thesis, we introduce the topic of smart locks, summarise the current knowledge of the Noke HD platform and describe the methodology used during the design and security analysis.
The practical part is split into design and security analysis. In the design analysis, we reverse-engineer the protocol using various passive and active techniques and analyse the design choices made by the manufacturer. In the security analysis, we use the knowledge of the protocol and analyse its security against common attacks on authentication protocol. We also assess the security of the padlock's platform in general.

1 Smart locks

The thesis studies a specific example of a smart lock. This section gives an overview of the topic of smart locks in general - their features, form factors and architectures. We also mention common security weaknesses of smart locks.

1.1 Smart lock definition

A smart lock is a vague term that is not sufficiently defined in the literature. The word "smart" has become a widespread buzzword in recent years. Companies call "smart" any device that adds features from the digital era to the existing functionality. Silverio-Fernández identifies five aspects commonly associated with smart devices: autonomy, connectivity, context-awareness, user interaction and portability [5]. A smart lock is thus *a device that keeps something (such as door, window, or box) from being opened and that is usually opened by using a key* [6], which includes additional "smart" features. Thus, smart locks should be able to operate autonomously, offer Bluetooth or WiFi connectivity, gather information through different sensors or provide enhanced user interaction.

1.2 Typical features of smart locks

Smart locks usually offer features such as:

- Mobile app - The main feature of a smart lock is often the possibility of connecting the lock wirelessly to a smartphone app. The user can unlock the lock from the app, see its activity, or change its settings.

- Alternative unlocking methods - A conventional lock is typically opened by a key or by entering a secret number. Smart locks are often unlocked using biometrics (for example, by fingerprint), by a mobile app (over Bluetooth or WiFi connection) or by a key fob or keycard (using RFID or NFC).

- Access control - The owner of a conventional lock can perform simple access control by giving the physical keys to specific individuals or taking the keys back. The smart lock allows access control by changing the lock's settings.

- Logging - A smart lock can, for example, log when a user opens the lock or when an intruder tries to force open the lock. This information can be valuable for monitoring the security of the locked asset.

1.3 Common form factors

Smart locks come in different form factors, similar to the form factors of conventional locks. These are the most common ones:

- Complete door lock - This type of smart lock completely replaces the conventional door lock, including the bolt and the handle. An example is the Yale Doorman lock [7].

- Addition to an existing door lock - This type is mounted on a conventional door lock and adds the "smart" functionality. It is essentially a device that automatically turns the key in the lock. An example is Glue Smart Door Lock Pro [8].

- Padlock - Many smart locks come in the form of a padlock, which can be mounted in many different ways and is easily replaceable and transportable. The main subject of this work, the Noke HD padlock, comes in this form factor.

- Special locks - The features of a smart lock can be present in a wide range of other locking devices.

1.4 Common architectures

Usually, a smart lock is not a standalone device but part of a bigger platform with multiple actors. The platform can have several different architectures. The architecture determines which actors are present and their roles. These are the possible architectures:

- Simple architecture without backend - The only two actors present in the system are the lock and the mobile app. There is no backend, meaning the platform does not depend on an internet connection. As remote access to the lock is often the main selling point for smart locks, we could not find an example of a product with this architecture. The Noke HD padlock's feature Offline unlock [9, Section Unlocking offline] is close to this architecture. It allows the user to cache offline keys to the mobile app and unlock the padlock without interacting with the backend.

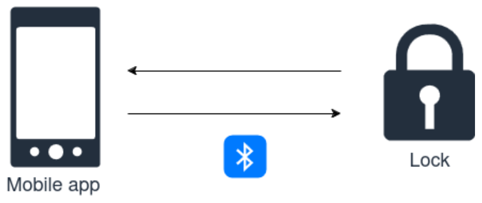

Figure 1.1: Simple architecture without backend

- Lock without direct access to the backend server - This architecture has three actors: the lock, the mobile app and the backend server. The lock communicates only with the mobile app; if it needs to communicate with the backend server, the mobile app forwards the traffic. Thus, the lock only has temporal access to the server. The server can store access logs and user information or even play a role in the unlocking process. This architecture is one of the most commonly used. The Noke HD padlock's platform has this architecture.

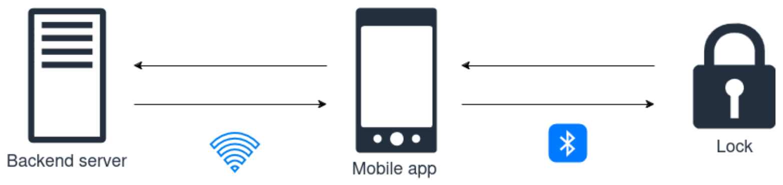

Figure 1.2: Lock without direct access to the backend server

- Lock with direct access to the backend server - This architecture adds a hub to the previous one. The hub connects to the lock over WiFi, which ensures the lock's constant connection to the internet. This allows the user to control the lock remotely. The second option is to use Bluetooth as in the previous architectures. An example of this architecture is the Glue Smart Lock [8].

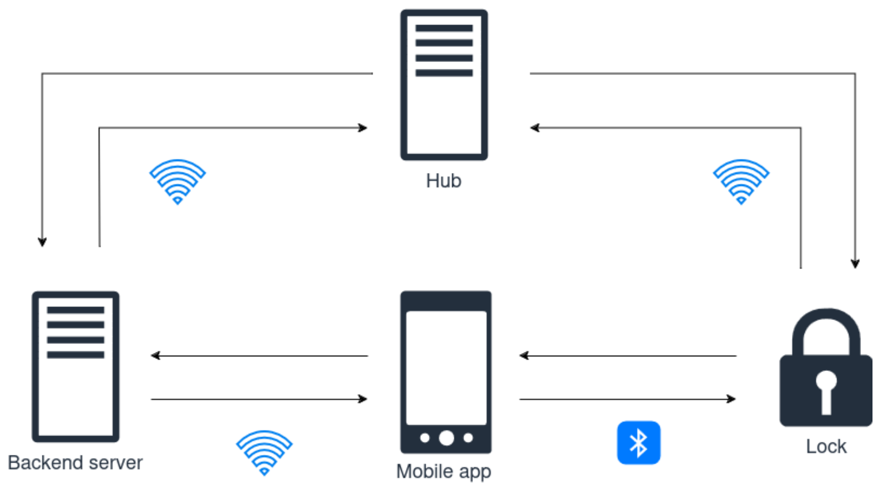

Figure 1.3: Lock with direct access to the backend server

1.5 Advantages and disadvantages of smart locks

1.5.1 Advantages of smart locks

The main advantage of a smart lock is the flexibility it offers. The access control is much easier than with a conventional lock. The administrator can easily grant specific users the right to unlock specific locks and flexibly change the settings. For non-smart locks, this level of access control requires manufacturing new keys, replacing the lock and distributing the new keys to the users [10].
The second advantage is higher security if the smart lock is designed correctly. Copying a physical key is usually easier than breaking a strong cryptographic key. Biometric authentication, which smart locks

often provide, also improves security. It is easier to copy or steal a physical key than to copy or steal the fingerprint for a well-constructed fingerprint reader. Moreover, smart locks often notify users about an intrusion.
The lock can also be integrated into a so-called smart home. For example, unlocking or locking the lock can trigger other actions around the house, such as turning on the lights.

1.5.2 Disadvantages of smart locks

The main drawback of electrical smart locks is that electric energy is required for the lock to work. Whether it takes power from the power grid or a battery, the lock usually cannot be controlled without electric power [10].
The second issue of smart locks comes from the complexity of the device. Developing a secure smart lock includes developing a secure hardware product, a secure app, a secure firmware and a secure API. This means it is more challenging to ensure security. Compared to an ordinary lock, there is more room for an error [10].

1.6 Common security weaknesses

When designing a conventional lock, security is the main focus. The primary function of a lock is to keep things, places, and people secure from unauthorised access. A smart lock must also be secure, but sometimes, all designers' attention is attracted to the "smart" features. Therefore, security is often neglected, especially in cheaper lock development. Furthermore, as was mentioned in Section Disadvantages of smart locks, the complexity of smart locks leaves more space for a poor design choice, leading to a vulnerability.
This section summarises the most common security vulnerabilities of smart locks found in related work.

1.6.1 Hardcoded secrets and plaintext passwords

Usually, the lock's security depends on the confidentiality of some cryptographic keys. These could be the keys used to encrypt the communication or authenticate the transmitted data. Sometimes, these

keys are not protected enough or even not protected at all. This security risk is number one on the OWASP IoT Top 10 list [11], where common IoT vulnerabilities (as of 2018) are listed.
For example, Lounis and Zulkernine [12] found this vulnerability in the Nulock bike lock. For opening the lock, the mobile app sends a user-set password to the padlock. This password is transmitted in plaintext over a Bluetooth connection. The attacker can sniff this password and use it to open the lock.
Pasknel [13] found a hardcoded AES key after decompiling the mobile app of the Noke padlock (predecessor of the Noke HD padlock). This vulnerability allowed him to decrypt sniffed Bluetooth traffic between the mobile app and the padlock of any user. From the decrypted traffic, the attacker can learn the padlock's secret key used to unlock the padlock and open the padlock anytime.

1.6.2 Static unlock command

Some smart locks that are unlockable even without learning any secrets. The attacker can just eavesdrop on the commands sent to the lock and replay them later. The danger of this attack lies in the fact that the attacker does not have to understand the protocol. The only thing needed is to capture the command and replay it. The main cause of this vulnerability is the lack of freshness in the commands. This means that they are identical for every session.
For example, the Tapplock padlock does not implement any replay protection. The unlock command is identical every time the padlock is unlocked. This means the attacker can capture one and unlock the padlock anytime [14]. Rose and Ramsey [15] found the same vulnerability in multiple locks: Ceomat, Elecycle, Vians and Lagute Sciner.
Xiaomi Sherlock S2 tries to provide some replay protection by changing the keys every several minutes. However, replaying the command is still possible before the keys expire [16].

1.6.3 Insufficient physical security

When designing a smart lock, companies tend to emphasise the software aspect of the lock and forget about its physical security. The lock's hardware is then vulnerable to lockpicking or brute-force attacks. This vulnerability is also number 10 on the OWASP IoT Top 10 list [11]. Insufficient physical security is mainly an issue of cheaper locks. Tapplock [17] and Fipilock [18] have a weak shackle that is not resistant to bolt cutters. Lodge [18] showed on the Fipilock (and Ray [19] on the Master lock) that using a magnet to open the lock is also possible. On some locks, the manufacturer neglects security almost entirely and leaves screws accessible on the body of the lock. The attacker can then take the lock apart and open it from inside. This vulnerability was found in Slok [20], Nokelock[1] and Uervoton [17] locks.

1.6.4 API vulnerabilities

Most smart locks need a backend server to work. They use it to store access logs, manage users, or unlock the lock. The server provides API (Application Programming Interface) endpoints to the mobile app.
The smart locks' backend API can have various vulnerabilities. A common problem is using only HTTP for communication, which was found in Tapplock [14] and Nokelock [21]. A network attacker can eavesdrop on the internet traffic and learn confidential information from the unencrypted HTTP traffic.
The same research found an Indirect Object Reference (IDOR) vulnerability in the Nokelock API. Nokelock's mobile app uses the API to obtain a cryptographic key used to open the lock. The request is not authenticated and contains only the lock's MAC address, which is public. This means the attacker can request any lock's key by knowing the lock's MAC address only.
The API of the Master Lock padlock has authorisation issues. The user opens the lock by entering a code on a padlock's directional pad. The platform has two user levels: an admin and a guest. The admin uses a primary code and issues temporary codes to the guests. The researchers found an API endpoint where even the less-privileged guest user can successfully retrieve the admin's primary code. An attacker with the guest role can thus keep access to the lock even after their temporary code is revoked [22].

1. The Nokelock padlock will appear several times in this work, but the lock is unrelated to the Noke HD padlock. Nokelock is a cheap Chinese padlock with an unknown manufacturer.

2 The Noke HD padlock

The subject of this work is a smart padlock Noke HD manufactured by the company Noke Inc. (part of Janus International Group Inc.), together with the entire platform enabling control and usage of the padlock (a mobile app and the backend servers).

2.1 Overview of the platform

The Noke HD lock has the form factor of a padlock (see Figure 2.1). The architecture is "Lock without direct access to the backend" (see Section Common architectures). The primary option for unlocking the padlock is via Bluetooth from a mobile app. Secondary options for unlocking the padlock without a mobile phone are access codes (called Quick click codes) and Noke Fob - a separate device simulating a physical key. The platform offers a web portal for administration that provides features typical for smart locks, such as access control and inspecting the padlock's logs.
The primary security principle is that the keys required to unlock the padlock are stored on the Noke's backend server, and the mobile app must fetch them over the internet. Communication between the padlock, the mobile app, and the backend is encrypted.
Noke Inc. does not focus on individual customers. The company sells the hardware (the padlock) and access to the Noke's backend to other companies. We will call them resellers. Resellers usually build the whole platform (the app and the web portal) for the padlock themselves. They can use the Noke Mobile library to build the mobile app and the Noke Core API to communicate with the Noke backend. The reseller can decide whether to implement all the features Noke provides or add more features on top of them. Some resellers provide the padlock to individual customers for sale, while others use the padlocks to provide other services (such as self-storage).
We obtained the padlock and access to the platform for this work from the Geokey company [23]. Other resellers include Movatic [24], who sells the padlocks to individual customers; Astute Access [25], who sells them to enterprises; and Alexela [26], who uses the padlock to provide car trailer rental service.

2.1.1 Hardware perspective

This section describes the padlock from the hardware point of view. Although the main focus is on the padlock's smart features, it helps to provide a complete picture of the padlock.

External structure

The padlock consists of a body and a shackle (see Figure 2.1). It is certified with CEN (Central European Norm) Grade 4 - High Security (according to standard EN 12320:2012 [27]). The padlock should thus be highly resistant to physical attacks. It is also presented as being resistant to water and high and low temperatures, which should allow using the padlock outdoors.
The main difference from a standard padlock is a round touch sensor on the bottom of the padlock where the keyhole on a conventional padlock usually is. This touch sensor turns on the padlock. Around the touch sensor are LEDs indicating the padlock's activity by changing colours. They turn blue when the touch sensor is touched, white when held longer, green when a correct command is received, red if the command is wrong, and pink in case some other error occurs.

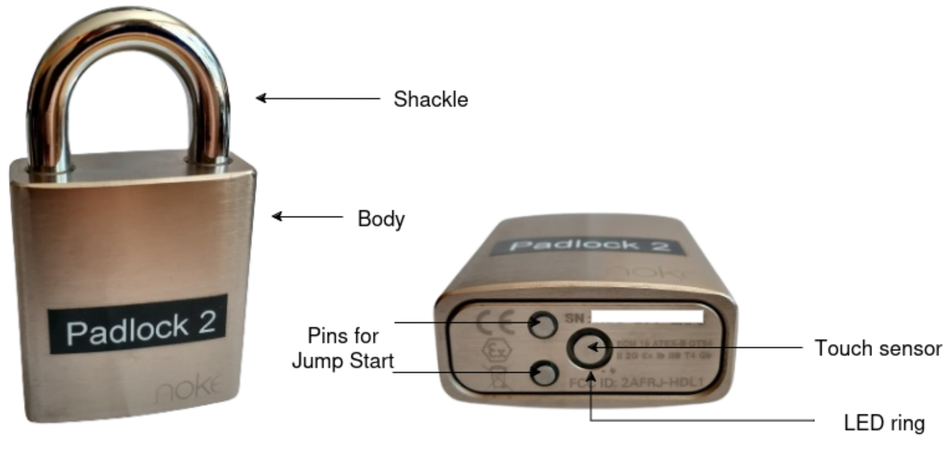

Figure 2.1: Noke HD padlock

Internal structure

To disassemble the padlock, the user has to unblock the shackle using the mobile app (in the same fashion as the padlock is unlocked) and remove the shackle. This opens access to a screw inside the padlock holding the padlock's bottom lid. Removing the lid reveals internal components visible in Figure 2.2.
The main component is the Nordic Semiconductors nRF52840 chip, which provides computational power and a Bluetooth interface to the padlock. Another important component is a servomotor, which unblocks the shackle when the padlock is unlocked.
The padlock is powered by a 3V CR2 battery, which should last at least three years, according to the official documentation. If the padlock runs out of battery, the padlock provides a feature called Jump Start. The user can place the contacts of a fresh battery on two pins next to the touch sensor on the bottom of the padlock, which provides enough power to unshackle the padlock and replace the battery.

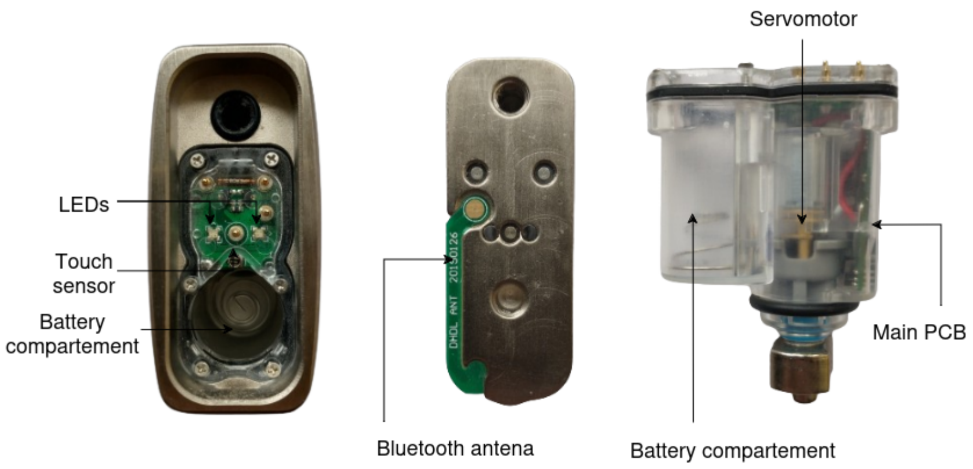

Figure 2.2: Internal structure of the padlock. From the left: bottom of the padlock with removed lid, bottom lid, internal container.

2.2 Actors present in the platform

The padlock is not fully operational on its own. Communication between three entities is required to operate the padlock: the padlock, the mobile app, and the Noke Core backend servers. In our case, an additional actor is the reseller's backend server. We refer to all these actors with the general term "platform". Figure 2.3 shows the entire platform and the communication between the actors.

2.2.1 The padlock

The padlock is an active actor in the communication. Its microchip with an integrated Bluetooth radio chip is used to encrypt and decrypt packets, verify them, and perform other tasks. The padlock is the central part of the system, and the goal of the whole platform is to control it securely.

2.2.2 The mobile app

An app installed on a smartphone lets the user open the padlock over Bluetooth. In the background, the app acts only as an intermediary between the padlock and the backend. Different users can log in to the mobile app and communicate with different padlocks. In our case, the mobile app is called Geokey App.

2.2.3 The reseller's backend server

The mobile app does not have to have direct access to the Noke Core backend server. A reseller's backend server can moderate the communication. This is the case of the Geokey platform. The server offers a web portal allowing administrators to see the logs, perform access control and manage the padlocks. From the perspective of the whole solution, the reseller's backend is only an intermediary between the mobile app (or effectively the padlock) and the Noke Core backend.

2.2.4 The Noke Core backend server

The Noke Core backend server is the last actor in the chain and, together with the padlock, is the most important. The Noke Core backend generates the unlock commands and stores the cryptographic keys for the padlocks. The Noke company manages the Noke Core backend server; neither the users nor the administrators can access it directly.

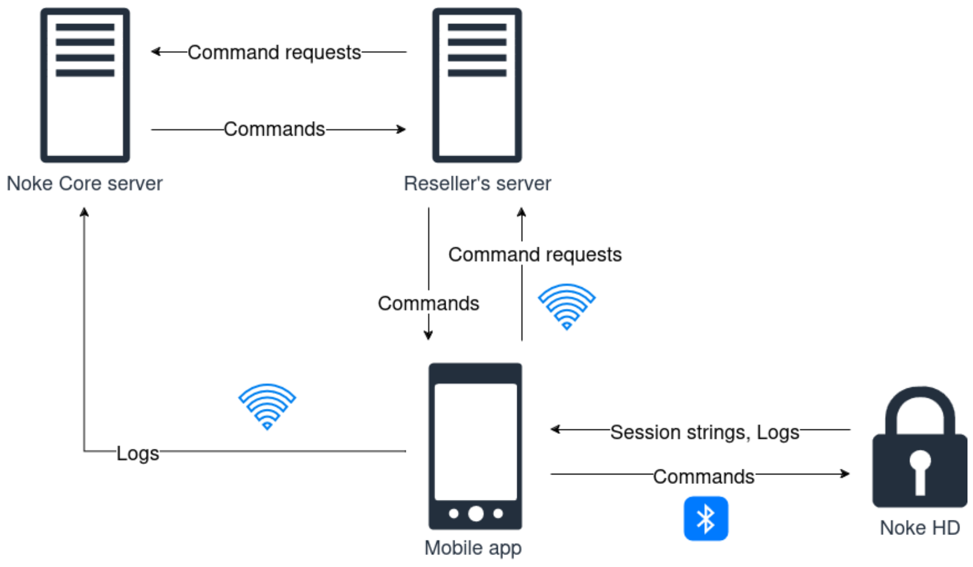

Figure 2.3: Actors of the platform and their communication

2.3 Overview of the unlocking protocol

The essential purpose of the unlocking protocol is to authenticate the actors mentioned above to each other and verify that the user is currently allowed to unlock the specific padlock. The process consists of multiple messages between the four actors. This section gives a high-level overview of the online and offline unlock process. This is the current level of knowledge about the protocol, and it will serve as a baseline for reverse-engineering the protocol in the Design analysis section.

2.3.1 Online unlock

The most common way of unlocking the padlock is online unlocking when the mobile app has access to an internet connection. The process consists of the following steps:

1. Turning on the padlock - The user needs to physically touch the touch sensor on the bottom of the padlock to be able to connect to it. In the meantime, the padlock is turned off to save the battery power.
2. Connecting to the padlock from the mobile app over Bluetooth - The user taps the padlock's icon in the mobile app to establish a Bluetooth connection between the phone and the padlock.
3. Reading a session string from the padlock - The mobile app reads a unique string called a session string from the padlock.
4. Requesting an unlock command from the server - The mobile app requests an unlock command by sending the session string and the padlock's MAC address to the reseller's server. The server forwards the session string to the Noke Core server for processing. The Noke Core server processes the session string and generates an unlock command. Finally, the command is sent to the mobile app via the reseller's backend.
5. Sending the unlock command to the lock - The mobile app sends the unlock command over Bluetooth connection to the padlock; the padlock verifies it and opens.
6. Receiving and sending logs - After successful unlock, the padlock sends logs to the mobile app over Bluetooth, and the app forwards them to the Noke Core server. This step is not necessary to unlock the padlock.

Dismounting of the padlock's shackle is similar to the unlocking process. The only difference is the API endpoint used to request the command. Since the API endpoint is called "unshackle", we will call this process unshackling. "Offline unshackle" is not possible.

2.3.2 Offline unlock

The offline unlock is the platform's second option for unlocking the padlock when there is no internet connection. The mobile app can request offline keys, store them in the mobile phone storage and use them later. The Geokey reseller does not provide this feature. The offline unlock protocol consists of the following steps:

- Requesting offline key and unlock command from the server - The mobile app requests an offline key and an unlock command by sending the MAC address of the padlock to the reseller's server. The reseller's server forwards the request to the Noke Core server, which generates the offline key and the unlock command. Both values are sent to the phone via the reseller's server and stored in the phone storage.
- Turning on the padlock - When the user wants to unlock the padlock, the padlock needs to be turned on, as in the online unlock process.
- Reading a session string from the padlock - The mobile app reads the session string from the padlock, as in the online unlock process.
- Encrypting unlock command - The mobile app encrypts the unlock command using a combination of the session string and the offline key.
- Sending the unlock command to the lock - The mobile app sends the unlock command to the padlock as in the online unlock process; the padlock verifies the command and opens.

2.4 Other features of the padlock

The padlock and the whole solution include other features besides locking and unlocking. We will also analyse these features in the design analysis. This section is a short overview of them:

- Log management - The user can see times of successful unlocks and unlock attempts on a web portal.
- Quick click codes - The Quick click codes feature allows the user to unlock the padlock without a smartphone with a series of long and short taps on the power button. The Geokey reseller currently does not allow setting a Quick click code.
- Firmware update - The padlock's firmware can be updated using a Noke Update app. New firmware versions are not publicly available but have to be requested directly from the Noke company. The Geokey reseller currently does not provide an option to update the padlock's firmware
- Noke Fob - Noke Fob is a separate device that can unlock the padlock without a mobile phone and internet connection. The user first needs to load offline keys inside the fob, and the fob can unlock the padlock on its own. Detailed analysis of the Noke Fob is out of the scope of this work.

3 Methodology

This section sets the theoretical foundations for the practical part - the design and security analysis.

3.1 Design analysis

In design analysis, we examine the architecture of the Noke HD padlock and the entire platform, focusing mainly on the unlocking protocol. We try to accomplish the following goals:

1. Reverse engineer details of the unlocking (unshackling) protocol.

2. Examine other features of the platform.

3. Analyse the protocol and other features. Discuss possible reasons for design choices.

The analysis focuses mainly on the part of the platform provided by the Noke company and covers only partially the parts specific to the resellers. We have access only to the platform provided by the Geokey reseller, so we cannot cover all the resellers' platforms.

3.1.1 Methods used for reverse engineering

For reverse engineering of the unlocking protocol (and other parts of the platform), we use the following methods:

- Examining documentation of the Noke Core API - The documentation for the API of the Noke backend is publicly available on GitHub [28]. We can gain valuable knowledge about the protocol and the entire platform by carefully reading the documentation.

- Examining Noke Mobile Library code - The Noke Mobile Library, which Noke offers to developers to implement communication with the Noke HD padlock, is publicly available on GitHub [9]. We can extract information about the app's functionality and the protocol from the source code.

- Reverse engineering of the mobile app - We use a Jadx decompiler [29] to decompile the mobile app. The decompiled source code can provide more information about the app's functionality.

- Logging Bluetooth communication of the phone - We use an Android developer tool called Bluetooth HCI (Host Controller Interface) snoop log [30]. Once enabled from the phone's developer menu, every Bluetooth packet sent or received by the phone is logged. The packets are captured without Bluetooth encryption. The logs can be retrieved from the phone in a bug report generated using Android Debug Bridge (ADB) [31] and examined on a computer in Wireshark [32]. We can reverse-engineer the Bluetooth communication between the padlock and the mobile app by running the HCI snoop log on the phone with the mobile app.

- Sniffing and decrypting HTTPS communication between the mobile app and the backend - We use an Android app PCAPdroid [33] to intercept and save HTTPS communication between the mobile app and the backend. To bypass certificate pinning, we patch the official mobile app. This can be achieved by decompiling the app using apktool [34], changing the security network configuration to accept the PCAPdroid certificate and recompiling the app again with apktool. PCAPdroid stores TLS keys used during the connection, allowing us to decrypt the traffic later in Wireshark. Decrypted HTTPS traffic can give us additional information about the Geokey server.

- Active man-in-the-middle - We can emulate the app's functionality with a Python script. Using the script, we can alter the data sent between the actors and understand the protocol better. (See Analysis setup for more information).

- Communicating with the API - We attempt to communicate with the Geokey API on non-standard endpoints and examine the responses. The responses can provide additional information about the Geokey API and the platform in general.

3.2 Security analysis

In the security analysis, we utilize the findings from the design analysis and analyse the platforms's security. Again, the main focus is on the unlocking protocol because unlocking is the platform's main feature. We have the following goals:

1. Assess the security of the protocol against common attacks on authentication protocols
2. Discuss the security of the platform in general
3. Discuss the impact of found weaknesses

3.2.1 Threat modelling

Threat modelling is a crucial part of designing a secure system. Threat modelling aims to identify what is to be protected (assets), who can attack the system (threat actors) and where the attacker can strike (attack surfaces). We create our own threat model for the Noke HD padlock. It can help us understand the designers' intentions for the platform and make the security analysis more consistent.

Assets

Assets are system parts that must be protected to ensure the system's confidentiality, integrity and availability. The Noke HD padlock's platform has the following assets:

1. Cryptographic keys - AES keys for encrypting the traffic to and from the padlock, a secret unlock command, an API key for communication with the Noke backend, Log key used to encrypt the logs;
2. Firmware of the padlock;
3. Backend servers;
4. Credentials - Users' credentials to log in to the mobile app and the web portal;
5. Event logs - Activity logs of the lock and the other components.

Threat actors

A threat actor is an abstract model of an adversary from which the system has to be protected. The model limits their access and capabilities. Based on [35], these are:

1. Remote attacker - The actor has no access into the system. They intend to unlock the padlock without authorisation. An example of this threat actor is a thief trying to break into a building.
2. Network attacker - The actor has some access into the system. They are trying to gain access to other parts of the system. The example is a user of the padlock who is trying to unlock another user's padlock. A specific example is an attacker trying to reverse engineer the system and trying to use the padlock without using the backend of the Noke company.
3. Insider attacker - The actor can access the system's internal parts (for example, administrator or developer). They are trying to attack the system's users or create a backdoor to the system for future use.
4. Hardware attacker - The actor is trying to attack the hardware of the padlock using conventional techniques such as pliers or grinders. This threat actor is out of the scope of this work.

Attack surfaces

Attack surfaces are the places of the system where the adversary can attempt to attack the system. These are the attack surfaces of this system and examples of attacks that use them:

1. Bluetooth communication between the padlock and smartphone - eavesdropping, manipulating messages;
2. HTTPS communication between the smartphone and the Geokey server - eavesdropping, manipulating messages;
3. HTTPS communication between the Geokey server and the Noke Core server - eavesdropping, manipulating messages;

4. Hardware of the padlock - physical attacks, side-channel attacks;
5. The mobile app - extracting personal data, unauthorised access;
6. The backend servers - denial of service attack, unauthorised access.

Scope of the security analysis

Due to time and expertise constraints, we have to limit the scope of the security analysis. We chose the scope based on the typical security risks of smart locks (from Section Common security weaknesses). This is how we define the scope:

- Assets - The main focus is on the cryptographic keys and other digital secrets.
- Threat actors - The primary ones are the remote and network attackers. Insider attacker is considered only partially.
- Attack surfaces - The main focus is Bluetooth and HTTPS communication. Analysis of HTTPS communication between backend servers is limited because this attack surface is not directly accessible.

3.2.2 Attacks on authentication protocols

As the protocol used by the Noke HD platform is essentially an authentication protocol, we are considering typical attacks on authentication protocols. This section gives an overview of common attacks on authentication protocols using an overview provided by Oorschot [36]. Later, we choose relevant attacks from this list and use them for the security analysis.

- Replay attack - A replay attack is *An attack in which the attacker is able to replay previously captured messages (between a legitimate claimant and a verifier) to masquerade as that claimant to the verifier or vice versa* [37]. This attack targets weaknesses such as the static unlock command described in the Common security weaknesses section.

- Interleaving attack - Interleaving attacks are attacks that target the protocol using *messages or parts from one or more previous protocol runs, or currently on-going protocol runs (in parallel sessions), possibly including attacker-originated protocol runs* [36].

- Reflection attack - A reflection attack can be understood as a specific example of an interleaving attack. The basic idea is that the attacker tricks the target into providing an answer to its own challenge. It uses two parallel connections, A and B. The attacker captures a challenge from the target over connection A. Then he opens second connection B, sends the same challenge to the target and receives the response. He can then send the response back to the target and successfully authenticate.

- Man-in-the-middle attack (MITM attack) - MITM attack have an active version *where the adversary positions himself in between the user and the system so that he can intercept and alter data travelling between them* [38] and a passive version where the attacker only intercepts the data.

- Relay attack - A relay attack can be understood as a version of the man-in-the-middle attack with two men in the middle communicating with each other. For example, attacker A can be close to the smart lock and attacker B can be close to the lock's owner. Attacker A will capture packets sent by the lock and forward them to attacker B, who will send them to the owner and vice versa.

- Brute-force attack - A brute-force attack is *an attack that involves trying all possible combinations to find a match* [39]. The target of brute-forcing can be passwords, cryptographic keys or PIN codes. A dictionary attack is a more sophisticated version of the brute-force attack where the attacker is using a chosen subset of possible values. An example is a dictionary of the most common passwords.

3.2.3 Methods used for the security analysis

In the security analysis, we assess the protocol's security against some of the attacks on authentication protocols. We are excluding interleaving and reflection attacks. The interleaving attack is too broad and does not provide any method for testing. The reflection attack needs a protocol which uses the same method for authenticating both parties, which is not the case with the examined lock's protocol. The methods for the chosen attacks are described in this section.
We also assess the security of the platform and the protocol in general based on the findings from the design analysis. We mention notable weaknesses out of the scope of the security analysis.

Replay attack

We attempt to perform a replay attack by replaying an old unlock command to the padlock and requesting an unlock command with an old session string from the backend. We can see the results from the padlock logs and the responses from the backend server.
We do not attempt to sniff Bluetooth or HTTPS traffic from the perspective of a remote attacker. However, we consider such sniffing to be possible.

Brute-force attack

We do not need to perform the brute-force attack practically because we can assess the possibility of the attack based on the bits of security the secret has. The US National Institute of Standards and Technology recommends a security strength of at least 112 bits [40]. Attacks on secrets with a higher security strength are considered to be unfeasible.

MITM attack and relay attack

Based on the findings from the design analysis, we examine if MITM or a relay attack on the protocol is possible. We discuss if the platform's behaviour during the reverse engineering using an active MITM attack does not pose a security risk. We do not attempt to perform a practical attack from the perspective of a remote attacker. This would mean capturing, manipulating and forwarding the traffic.

3.3 Analysis setup

We implemented a Python script that emulates the app during the unlocking process and used it during the design and security analysis. The script can read the session string from the padlock, send it to the backend, obtain the unlock command, send it to the padlock and read the padlock's notifications. Using this script, we can perform replay and active MITM attacks and reverse engineer the protocol. An overview of the setup is in Figure 3.1. The script is available in Appendix A.
The Python script runs on a laptop connected to the padlock over Bluetooth and to the internet (and the backend server) over WiFi.

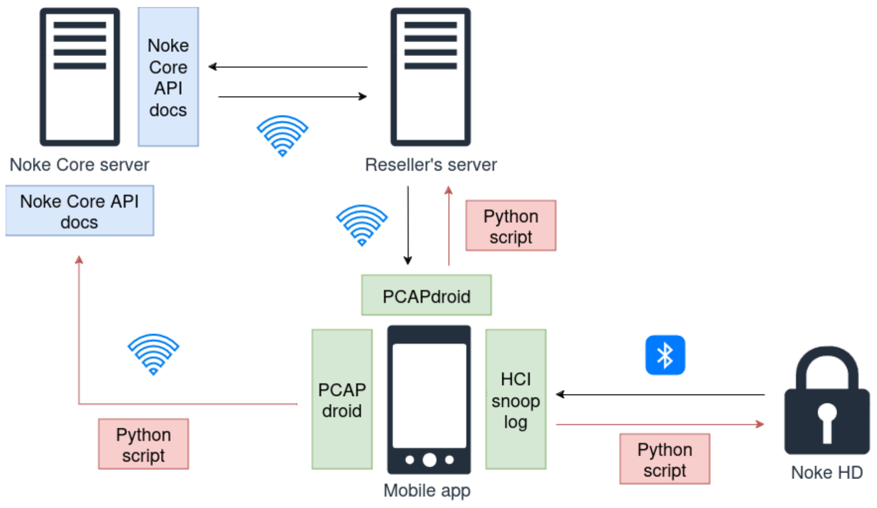

Figure 3.1: Overview of the setup. Blue parts indicate that we only have information about the communication structure; green parts are the surfaces where we eavesdrop on the communication, and in red parts, we can actively modify the communication.

4 Design analysis

4.1 Reverse engineering and analysis of the unlocking protocol

In this section, we use the methods from Methods used for reverse engineering section to enrich the bare description of the protocol with additional details. We also discuss possible reasons for the design choices. We are not analysing online and offline unlock separately because they share many steps. Usually, by "unlock", we mean "unshackle" as well.

4.1.1 Turning on the padlock

This step does not need any reverse engineering. We only used a stopwatch to measure the time, after which the padlock turns automatically off. The time is 30 seconds. There are two possible reasons for this choice. First, the padlock saves battery by only being active for short periods. Second, it improves security as an attacker has to touch the padlock to interact with it.

4.1.2 Connecting to the padlock from the mobile app

We gained additional information about the connection process from the HCI snoop log. The app connects to the padlock over Bluetooth Low Energy (BLE). The padlock is a GATT server, and the mobile app is a GATT client. GATT is a Generic ATTribute profile and defines how BLE devices transfer data. The server offers a table of GATT characteristics, which the client can read. The server is usually the peripheral. [41]
After the connection is established, the mobile app subscribes to notifications of one of the characteristics. These notifications enable the padlock to send messages without the app explicitly reading them. They are later used to send padlocks's logs. Connecting multiple clients to the GATT server on the padlock is not possible.
BLE is a standard communication channel for IoT devices because of

its low battery consumption and server-client architecture, which is common in IoT.

4.1.3 Reading the session string from the padlock

We examined the HCI snoop log and found that the session string is a 20-byte-long value stored in one of the GATT characteristics. From the code of the Noke Mobile Library, we were able to reconstruct the structure of the session string partially. The structure is shown in Figure 4.1.

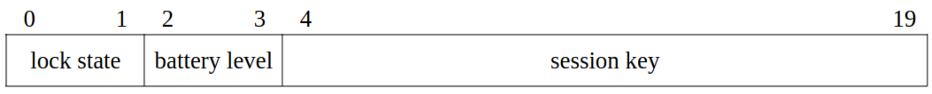

Figure 4.1: Structure of the session string

According to the Noke Mobile Library, the lock state is a variable that identifies if the padlock is currently unlocked, locked, unshackled, etc. The Table 4.1 shows all the states and their values.

Table 4.1: Lock states

Lock State	Value	Lock State	Value
UNKNOWN	-1	UNSHACKLING	4
UNLOCKED	0	UNLOCKING	5
UNSHACKLED	2	LOCKED_NO_MAGNET	7
LOCKED	3		

We do not know what the state "LOCKED_NO_MAGNET" means. We attempted to determine the location of the lock state in the first two bytes. We put the padlock into the states we can reproduce (unlocked, unshackled, locked) and read the session string. We observed that in practice, the first two bytes contained the same value every time (0xe427), meaning the lock state might be static or missing. If the lock state was implemented, the backend server could tell if the lock is opened or closed. The server might use this information to issue

commands for closed locks only.
The current battery level is a number between 0 and 3000, representing the current voltage of the padlock's battery in millivolts. The information is used to display the current battery level of the padlock on the web portal.
The session key is a seemingly random value. The Noke Mobile Libary documentation states that it is used to encrypt the unlock command for the padlock. Noke Core API's documentation says that the session string is unique and that the padlock generates it. We attempted to discover when the session key (and the whole session string) is generated and how often it refreshes. The results are shown in Table 4.2.

Table 4.2: Testing the session string generation

No.	Scenario	Result
1	Reading session strings A and B over the same Bluetooth connection	A == B
2	Reading session strings A and B over two separate succeeding Bluetooth connections	A != B
3	Reading a session string A, waiting for the padlock to turn off, reading a session string B	A != B

After a further examination of scenario 1, we found that the session string has the same value during the whole Bluetooth connection. The padlock thus generates the session string when the mobile app connects to it.
The main reason for the changing session string is to provide freshness to the unlock commands. An element of freshness is implemented in protocols to prevent replay attacks. In the security analysis, we examine if the protocol is vulnerable to replay attacks.

4.1.4 Receiving and sending logs

Even though receiving and sending logs is the last step of the unlocking process, we mention it now because the logs help us reverse engineer other steps of the unlocking process.

The padlock sends logs as BLE notifications. The mobile app subscribed to these after connecting to the padlock. We reconstructed the structure of the log packets from the HCI snoop log and the documentation. Figure 4.2 shows the structure.

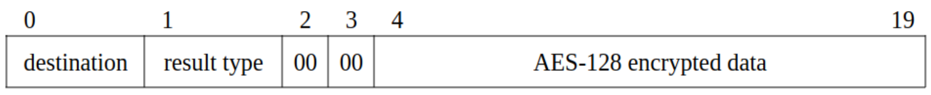

Figure 4.2: Structure of the log packet

The destination byte distinguishes two types of logs: logs intended to be processed by the mobile app and logs to be sent to the backend. The documentation calls them App and Server packets.
The result type specifies the result type of the log in the encrypted body. The result types found in the Noke Mobile Library source code are shown in Table 4.3.

Table 4.3: Log result types

Result type	Value	Result type	Value
SUCCESS	0x60	INVALIDDATA	0x65
INVALIDKEY	0x61	FAILEDTOLOCK	0x68
INVALIDCMD	0x62	FAILEDTOUNLOCK	0x69
INVALIDPERMISSION	0x63	FAILEDTOUNSHACKLE	0x6A
SHUTDOWN	0x64	INVALID	0xFF

The body of the log packets is encrypted with a Log key shared between the padlock and the Noke Core backend server. The user does not have access to this key. Encryption provides confidentiality to these potentially sensitive data and protects them against modification by a network attacker. Table 4.4 contains log events uncovered by the Noke Core API documentation and their possible meaning.

The app sends the Server packets, with the session string, MAC address and current time, directly to the Noke Core backend to endpoint `/upload/`. The mobile app uses a Mobile API key for authentication. The mobile app does not process the App Packets.

Table 4.4: Logged events

Event	Possible meaning
alarm_triggered	Noke U-Lock has a built-in alarm.
button_touched	Touch sensor is touched.
failed_unlock	Unlock was attempted but failed.
locked	The padlock was locked.
proximity_stats	Unknown
setup_unlocked	Unknown
start_up	The padlock turned on.
unknown	Unknown
unlock_via_access_code	Unlock using offline unlock.
unlocked	Online unlock.
unlocked_via_quick_click	Unlock using the Quick click code.
wake_stats	Unknown

We captured logs from multiple successful and unsuccessful online unlocks. After a successful unlock, the padlock sends one App packet and between 4 and 8 Server packets. We were not able to discover what causes this variation. After an unsuccessful unlock, only one Server packet and one App packet are sent. In all cases, the encrypted part of App packets was always empty (only zeros).
We attempted to discover how the Noke Core server processes the logs. We manipulated the encrypted body log packets or sent logs from a different session, but the API responded with a success code every time. This behaviour likely means the endpoint does not inform the user about internal errors, meaning we could not gain more information about the processing.
Noke Core API provides an endpoint `/activity/` to display the logs. However, the Geokey web portal does not use the data from this endpoint. It provides information about unlock times, but the source of this information is the requests to the Geokey server. We verified this by unlocking the lock without sending the logs to the Noke Core server. The web portal still showed a new unlock event. If the Geokey portal communicated with the `/activity/` endpoint, the user could see the lock's activity in more detail.

4.1.5 Communicating with the backend servers

In the case of the Geokey reseller, the app communicates primarily with the Geokey server, which moderates the communication with the Noke Core API. The only difference is the padlock logs that are sent directly to the Noke Core API.
From the PCAPdroid capture, we discovered that the requests to the Geokey API are encrypted with TLSv1.3 and that JSON Web Token (JWT) is used for authentication. The JWT contains UserID, expiration time and RefreshToken. It is valid for 15 minutes, and once it expires, it needs to be refreshed by sending the RefreshToken to `/api/RefreshToken`. This mechanism ensures that if an attacker captures the token, it can only be used within the next 15 minutes. The app requests an unlock (unshackle) command with an HTTP POST request to the Geokey server at endpoint `/api/unlock/` (`/api/unshackle/`). The request's body contains the session string and the padlock's MAC address. The request format for the Noke Core API is available in the documentation. A Private API key authenticates the request. The reseller obtains the Private API key directly from the Noke company. Table 4.5 shows different endpoints, their functions, and request content. All the requests are HTTP POST requests.

Table 4.5: Noke Core API endpoints

Endpoint	Function	Request content
`/unlock/`	Request unlock command	Session string, MAC address, (optional tracking key)
`/unshackle/`	Request unshackle command	Session string, MAC address, (optional tracking key)
`/keys/`	Issue, revoke and display offline keys	MAC address, (optional tracking key)

As the documentation explains, the tracking key is *an optional string used to associate to lock activity* [28, Section POST /unlock/]. The administrator can use the tracking key to filter logs requested using the API.

Vulnerability in the Geokey API

We attempted to discover if the Geokey API provides endpoints other than those we found in the captured traffic. The Geokey API endpoints follow the same naming convention by adding the `/api/` in front of the Noke Core API endpoint (as in the `/api/unlock/` and `/api/unshackle` endpoints).
We tried communicating with the `/api/keys/` endpoint. The `/keys/` endpoint of the Noke Core API serves to issue, display and revoke offline keys. The `/api/keys/` should not exist on the Geokey API, as Geokey does not provide the offline unlock feature.
However, the endpoint responded to an HTTP POST request containing an empty JSON in the body and returned offline keys, unlock commands, their revocation status and MAC addresses of 1352 padlocks. The same request is used at the /keys/ endpoint of Noke Core API to display offline keys for all user's padlocks.
This behaviour poses a severe security risk, which we discuss in the Elevation of privilege attack on the Geokey API Section of the security analysis. This section only uses this list of offline keys and unlock commands to reverse-engineer the protocol. We discuss possible explanation in the Notes to the /api/keys/ endpoint Section.
We did not find any other hidden endpoints using the technique.

4.1.6 Verifying the session string

Before the session string is used to encrypt the unlock command, the actor performing the encryption might verify its integrity and validity. We know from the Noke Mobile Library code that during an offline unlock, the library does not verify the session string. However, during the online unlock, the Noke Core server might perform additional verification steps. No information about the existence or nature of these steps is available.
We attempted to discover the session string verification process by sending modified session strings to the Noke Core backend (via the Geokey backend). Session strings were sent to both endpoints (`/api/unlock/` and `/api/unshackle/`). The results are in Table 4.6.

Table 4.6: Testing session string verification. ✓ - the command was successfully returned; SS - session string; SK - session key.

No.	Scenario	Response from /api/unlock	Response from /api/unshackle
1	Invalid length of SS	Error code 400 (Cannot interpret input)	Error code 500 (Internal server error)
2	1st byte of SS set to 0x00	✓	✓
3	2nd byte of SS set to 0x00	✓	✓
4	Battery level set to 0x00	✓	✓
5	Battery level set to invalid value (0xffff)	✓	✓
6	Modified SK	Error code 500 (Internal server error)	✓
7	SK replaced by an old SK from the same padlock	✓	✓
8	SK replaced by an old SK from a different padlock of the same user	Error code 500 (Internal server error)	✓
9	Modified MAC address	Error code 400 (Lock not found)	Error code 400 (Lock not found)
10	MAC address of a different padlock of the same user	Error code 500 (Internal server error)	✓

From the responses, we discovered that the server first [1] searches for the padlock in the database of the user's locks using the lock's MAC address (scenario 9). Then, it checks the length of the session string (scenario 1). The first four bytes are not checked (scenarios 2 - 5).
At last, the server verifies the authenticity of the session key. The session key is not a random value but is bound to the MAC address of the specific padlock (scenarios 6 - 8). A possible implementation is that the session key can contain, apart from a random value, a message authentication code (MAC) binding it to a specific padlock. This mechanism ensures that the session key comes from the respective padlock.
According to the results, the endpoint `/api/unshackle` does not verify the authenticity of the session key. This missing authentication check is likely an implementation fault and not an intentional feature of the unshackle process. We discuss the security implications in the MITM and relay attack Section.

4.1.7 Encrypting the unlock command

The unlock command is encrypted during both the online and offline unlocking process. The unlock command is a 20-byte-long secret value shared between the padlock and the Noke Core server, which is used to open the padlock. We will distinguish it from the unlock packet, which consists of an encrypted unlock command and a plaintext header. The Noke Core server encrypts the command during the online unlock process and the mobile app (using the Noke Mobile Library) during the offline unlock process.

Offline unlock

The library needs an unlock command, a offline key and the session string from the padlock for the encryption. The unlock command and offline key can be obtained from the Noke Core server. We describe how we obtained our padlock's unlock command and offline key in

1. When the server receives a request with a modified MAC address and invalid session string length, the error is "Lock not found", which means the search for the padlock happens first.

Section Communicating with the backend servers. The maximum number of offline keys for one padlock is 100. The documentation briefly suggests that this is the number of offline keys stored in the padlock [28, Section Fobs]. The keys can be assigned to different users and be revoked individually. Offline unshackling is not possible.
The `offlineUnlock()` function from the `NokeDevice` class performs the encryption. We can emulate the process and manipulate different values using the obtained unlock command, offline key, and the Noke Mobile Library to test the server-side functionality.
The first 4 bytes of the unlock command are used as a header of the unlock packet. Then, the current timestamp is stored in the unlock command. A checksum of the unlock command is then calculated and stored in its last byte.
The library creates a preSessionKey by XORing the session string and the offline key. The session string limits the validity of the unlock packet to the current session only. There is a potential mistake in the code performing the XOR operation. The expected behaviour would be to XOR the session key (the last 16 bytes of the session string) with the offline key. However, the first 16 bytes of the session string are used. This mistake is probably caused by the different lengths of the session string (20 bytes) and the offline key (16 bytes). PreSessionKey is thus constructed using the predictable header of the session string, and the last 4 bytes of the session string are not used. We discuss possible security implications in the MITM and relay attack Section of the security analysis.
Finally, the body of the unlock packet is encrypted using the AES-128 algorithm with the preSessionKey as a key. The resulting structure of the unlock packet is in Figure 4.3.

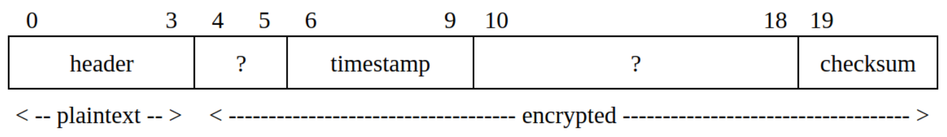
0
3
4
5
6
9
10
18
19
header
?
timestamp
?
checksum
< -- plaintext -- >
< ---------------------------------- encrypted ---------------------------------- >

Figure 4.3: Structure of the offline unlock packet. The question marks label bytes which are not modified during the offline encryption process, and their content is unknown.

Structure of the unlock command

We used the offline keys and unlock commands obtained in the Communicating with the backend servers section to reconstruct the unlock command's structure. The first two bytes likely do not have any meaning because they are equal to zero in all captured unlock commands. The third byte has one of the values 0x04, 0x44, 0x84 or 0xc4. This means that the third byte of the unlock command has a binary value of **000100. We do not know what is the meaning of the first two bits. All captured online unlock commands have this byte equal to 0x44 and unshackle commands to 0x42.
The fourth bytes of the obtained unlock commands contain all numbers between 0 and 100. These 100 values correspond to the 100 available unlock commands. The fourth byte is thus likely to be a sequential number of the unlock command - offline key pair.
All remaining bytes of the unlock command are equal to zero except for bytes 6 and 20, which are equal to 0xa2. However, only the first of them stays in the final packet. The checksum overrides the second one, so the value is not important. An overview of the unlock command structure is shown in Figure 4.4.

0	1	2	3	4	5	6	18	19
00	00	command type	sequence number	00	a2	00		a2

Figure 4.4: Structure of the unlock command

Online unlock

Online unlock might use the same encryption process. This approach would simplify the whole protocol and reduce the amount of code in the padlock's firmware used to handle the packets. We attempted to verify this hypothesis.
We reconstructed the structure of the packets from multiple unlock and unshackle packets captured using the HCI snoop log. As shown in Figure 4.5, the structure is similar to the offline unlock command.

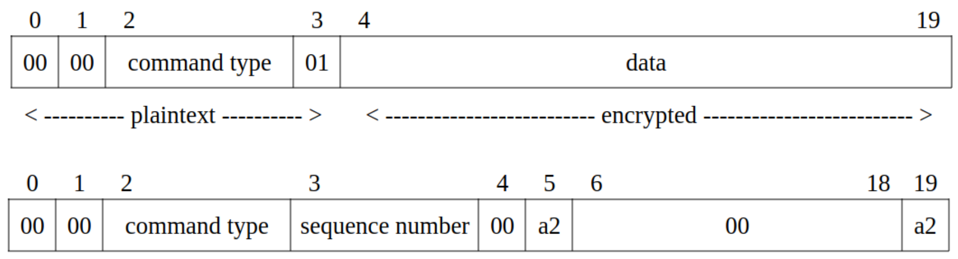
0
1
2
3
4
19
00
00
command type
01
data
< ---------- plaintext ---------- >
< ------------------------- encrypted ------------------------- >
0
1
2
3
4
5
6
18
19
00
00
command type
sequence number
00
a2
00
a2

Figure 4.5: Comparison of online unlock packet (top) and offline unlock command structure (bottom)

Next, we attempted to determine if the Noke Core server makes the same mistake while encrypting the unlock command as the Noke Mobile Library. This would mean they share part of the source code, and the entire encryption process is likely the same. The missing authentication check on the `/api/unshackle/` endpoint allowed us to request an unlock packet with a modified session string. We changed the last four bytes of the session string to a random value, requested an unshackle packet and successfully unshackled the lock using this packet. This means that the server likely uses the `offlineUnlock()` function from the Noke Mobile Library, and the encryption process is the same for offline and online unlock. The difference is the actor who performs the encryption and the fact that the authenticity of the session key is checked during an online unlock.
Online encryption likely uses the same set of keys and unlock commands as offline encryption. If two sets of keys were used, the padlock would need to distinguish between offline and online packets before decryption. However, the header contains no such information. The first two bytes are equal to zero in both offline and online packets. The third byte is 0x44 in all online and some offline unlock packets. The fourth contains only the sequence number of the key - command pair. The unshackle command likely has only one key, different from the keys for unlock commands. The only reason for 100 keys for unlock commands is that they can be distributed among users as offline keys and individually revoked. As offline unshackle does not exist, only one key is needed.

4.1.8 Verifying the unlock command by the padlock

The mobile app writes the encrypted unlock packet to a GATT characteristic of the padlock over the existing Bluetooth connection.
The documentation does not describe how the padlock verifies the unlock packet. The padlock can be expected to implement the following steps: verification of the structure and header of the unlock packet, decryption of the unlock packet, and validation of the unlock command. As the encryption process is likely the same for online and offline unlock, the padlock verification process will also be the same. We attempted to reconstruct the details of these hypothetical steps.
To obtain information on how the structure of the unlock packet is checked, we modified the header of the encrypted unlock command and sent it to the padlock. Then, we observed the logs received from the padlock. The results are in Table 4.7. However, an alternative explanation is that the padlock's firmware might not implement the verification phase, and the logs might result from failed decryption or validation.

Table 4.7: Testing verification of unlock packet

No.	Scenario	Result
1	Invalid command type (0xff)	Alternating between result type 0x60 (Success) and result type 0x6d (unknown). The padlock does not unlock.
2	Sequence number out of range (0x64)	Result type 0x61 (Invalid key)

An interesting finding is the behaviour of the padlock when it receives an invalid command type. The padlock seemingly enters an undefined state. We examine this potential weakness in the MITM and relay attack Section of the security analysis.
The firmware then selects the key for decryption based on the sequence number from the packet header. PreSessionKey is created using the current session string and the key, and the packet is decrypted.
After the decryption, the resulting unlock command is checked. We attempted to discover which parts of the unlock command are checked

in the firmware by encrypting an unlock command with a modified body and sending it to the padlock. Table 4.8 shows the results.

Table 4.8: Testing verification of the unlock command. ✓- successful unlock.

No.	Scenario	Result
1	Modified 1. byte (0xff)	Result type 0x61 (Invalid key)
2	Modified 2. byte (0xff)	Result type 0x61 (Invalid key)
3	Modified 5. byte (0xff)	✓
4	Modified 6. byte (0xff)	Result type 0x63 (Invalid permission)
5	Timestamp set to zero	✓
6	Modified 11. - 19. byte	✓
7	Wrong checksum	✓

The results show that the padlock checks only the first four (which are sent in plaintext) and the sixth byte (which is static in all obtained unlock commands) of the unlock command. This behaviour raises questions about security, which we answer in the MITM and relay attack Section of the security analysis. The purpose of the timestamp might be to add randomness to the unlock packet, but the session string used for encryption already ensures that. The checksum does not serve any purpose as it is not checked. The sixth byte could contain the command's operation code (indicating what action the padlock should perform). We base this hypothesis on the Invalid permission response but cannot verify this hypothesis.
If the check is successful, the padlock opens itself. The documentation states that a synchronisation phase follows a successful online unlock. The padlock synchronises information about issuing and revoking Quick click codes and offline keys with the backend server. We could not capture any messages from the mobile app to the padlock after successful unlock, which could be part of the synchronisation phase. This might be caused by the fact that the Geokey reseller does not implement any features that would need synchronisation or does not implement the synchronisation phase altogether. After the synchronisation phase, the padlock immediately turns off.

4.1.9 Notes to the /api/keys/ endpoint

We obtained a list of many offline keys and unlock commands in the Communicating with the backend servers section. This section discusses possible explanations why such a list exists.
The list could be used in cases when the customer requests offline keys. Another explanation is that the Geokey company wants an option to unlock the padlocks without interacting with the Noke Core API. Another hypothesis is that Geokey uses offline instead of online unlock. When a new lock is registered, the Geokey server might request new offline keys from the Noke Core API. When the user wants to unlock the lock, the Geokey server would generate the unlock packet without communicating with the Noke Core API. If Noke charged for every request to the API, this would reduce the cost. However, the commands obtained from the `/api/keys/` endpoint have a different header than those obtained from the `/api/unlock/` endpoint and changing the header invalidates the command, which rules out this hypothesis.

4.1.10 Summary of the protocol

This section briefly summarises the findings from reverse-engineering the unlocking protocol. It does not discuss all the details of the protocol. The padlock and the backend share the following secrets:

- 100 keys (also called offline keys) used to encrypt the unlock commands
- Unknown number of unlock commands
- Log key used to encrypt the logs
- One key - command pair for unshackling the padlock
- Possibly a secret used to authenticate the session keys

This is the sequence of actions when the user unlocks (unshackles) the padlock using an online unlock. The offline unlock shares similar steps. The only difference is that the key - unlock command pair is stored on the mobile phone, and that the mobile app performs the encryption. The backend is a general term for the Noke Core and reseller servers. Figure 4.6 shows a visual overview of the protocol.

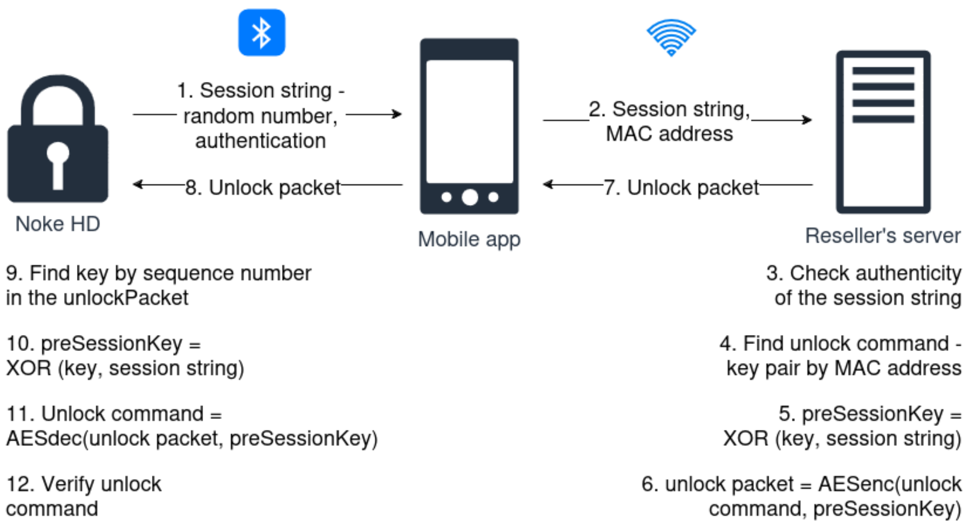
1. Session string - random number, authentication
8. Unlock packet
Noke HD
Mobile app
2. Session string, MAC address
7. Unlock packet
Reseller's server
9. Find key by sequence number in the unlockPacket
10. preSessionKey = XOR (key, session string)
11. Unlock command = AESdec(unlock packet, preSessionKey)
12. Verify unlock command
3. Check authenticity of the session string
4. Find unlock command - key pair by MAC address
5. preSessionKey = XOR (key, session string)
6. unlock packet = AESenc(unlock command, preSessionKey)

Figure 4.6: Overview of the protocol

1. The user turns on the padlock and connects to it over Bluetooth using the mobile app. The padlock generates a session string consisting of padlock's authentication and random number. The mobile app reads the session string over Bluetooth.
2. The app sends the session string and padlock's MAC address to the backend.
3. The backend verifies the authenticity of the session string.
4. The backend searches for an unlock command - key pair using the padlock's MAC address.
5. The backend calculates a preSessionKey by XORing the session string with the key.
6. The backend encrypts the unlock command with AES-128 using the preSessionKey, forming an unlock packet.
7. The backend sends the unlock packet to the mobile app
8. The mobile app writes the unlock packet to the padlock.
9. The padlock chooses the decryption key using a sequence number in the header of the unlock packet.
10. The padlock calculates a preSessionKey by XORing the session string with the key.
11. The padlock decrypts the unlock packet with AES-128 using the preSessionKey, obtaining the unlock command.
12. The padlock verifies the unlock command and opens.

4.2 Reverse engineering and analysis of other features of the platform

The Geokey reseller does not provide these features, so the possibilities to reverse engineer them are limited to studying the documentation.

4.2.1 Quick click codes

The user creates Quick Click codes using Noke Core API endpoint `/qc/issue/`. The request contains the MAC address of the padlock and the code encoded as a series of 1's and 0's. The code must be between 3 and 24 taps long. The Quick Click code is automatically loaded to the padlock after the next successful online unlock (in the synchronisation phase). A single padlock can hold up to 100 Quick Click codes. Each code of the padlock has to be unique.
Quick Click codes can be revoked using the Noke Core API, but the change takes effect only after the next synchronisation phase.
There is no information about this feature's implementation, but the solution can be simple. The backend has to have a database of the padlocks and Quick click codes. This database is synchronised with the padlock after every successful online unlock.

4.2.2 Firmware update

The documentation describes that uploading new firmware to the padlock consists of several steps.
First, the padlock needs to be switched to a Firmware Update Mode. This is not Noke's design choice but a step required by the microchip. The Nordic Semiconductors documentation calls it Device Firmware Update mode (DFU) [42]. The command for turning on the DFU mode can be requested from the Noke Core API at endpoint `/fwupdate/`. The request contains the padlock's MAC address and session string. The session string serves as a replay protection, as it does in the unlocking process.
Once the padlock reboots, it is in DFU mode, making it ready to receive new firmware. This firmware file is loaded via Bluetooth from the Noke Update app to the padlock and installed.

5 Security analysis

The padlock datasheet mentions the product's security and states that *the system has passed rigorous third-party testing* [43]. However, no report from this testing is publicly available. Other main selling points from the security perspective mentioned in the datasheet are 128-bit AES encryption, Bluetooth 4.0 and cloud-based key storage.
We use the attacks and methods described in the theoretical part and the findings from the design analysis. The analysis is limited to the scope defined in the Scope of the security analysis section.

5.1 Security features of the unlocking protocol

The protocol described in the design analysis implements several main building blocks to ensure the security of the unlocking process.

- Session string - A unique session string generated after every connection to the lock is used to encrypt commands sent to the lock. This binds the commands to the specific session and ensures their freshness. It aims to prevent replay attacks.
- AES encryption - Commands sent to the lock and the logs generated by the lock are encrypted using the AES-128 algorithm. This ensures the confidentiality of the commands and the logs.
- Authentication of the session string - All session strings generated by one lock are linked to the lock's MAC. This mechanism authenticates the lock to the backend server.
- TLS encryption of the HTTP traffic - All HTTP traffic between the backend servers and the mobile app is encrypted using TLSv1.3. This ensures the confidentiality of the messages.
- Authentication of the requests to the Geokey API - Each request to the Geokey API is authenticated using JWT. This authenticates the user to the API.
- Authentication of the requests to the Noke Core API - Each request to the Noke Core API is authenticated using an API key. This authenticates the Geokey server to the Noke Core API.

5.2 Replay attack

The design analysis found that the platform implements an element of freshness in the form of the session string. This type of protocol should not be vulnerable to replay attacks. However, even a good architecture still leaves space for a bad implementation.
Two packets can be replayed: a command to the padlock (unlock or unshackle) and a session string to the backend. A successful replay attack on the unlock command would mean the padlock can be opened with an old unlock command. A successful replay attack on the session string would allow an attacker to request a new unlock command with an old session string.
We tested replaying old unlock and unshackle commands over new Bluetooth connections and got the expected results. The padlock does not unlock (unshackle) and responds with logs containing result type 0x61 (Invalid key). This means that the session string does provide freshness and replay protection. The command can be successfully decrypted and verified only if encrypted by the session string of the current Bluetooth connection. The fact that the padlock immediately turns off after it is locked further secures it against replay attacks.
Replaying the session string from the same padlock to the backend servers was successful. The unlock (unshackle) command was obtained, but it failed to unlock (unshackle) the padlock. This is caused by the session string (and thus the unlock command) being valid only for the Bluetooth session. A partial replay attack on the session string is possible but poses no security risk.
We verified the assumption from the design analysis that the session string protects the padlock from replay attacks of this kind.

5.3 Brute-force attack

The primary value to brute-force is the unlock packet. Successfully guessing the unlock packet would allow a remote attacker to open the lock without access to the backend server or the app. The unlock packet is 20 bytes long. The first two bytes are static, the command type byte has six static bits, and the sequence number byte uses only seven bits (because the range is 0 - 99). This gives the unlock packet

131 bits of security, which is over the recommended 112 bits.
The session string and offline key both offer sufficient 128 bits of security.
We did not discover any weak secrets, and we do not consider the protocol vulnerable to brute-force attacks.

5.4 MITM and relay attack

The active testing performed in the design analysis is an active MITM attack. We modified values sent during the unlocking process on both the HTTP and Bluetooth channels. No vulnerabilities were found, only four weaknesses, which appear to be mistakes in the implementation of the protocol:

- The padlock does not check the checksum of the unlock command - Even though Noke Mobile Library adds a checksum to the unlock command, the firmware does not check it. If the attacker gains access to the plaintext unlock command, they can change the content of the command, and the changes will not be detected using the checksum. However, we have not found any advantage the attacker might gain by doing this.

- Authenticity of session key not checked on `/unshackle/` endpoint - The `/unlock/` endpoint does check the authenticity of the session key. An attacker cannot obtain an unlock command using a modified session key. However, the `/unshackle/` does not check the authenticity of the session key. This allows the attacker to generate an unshackle command with an arbitrary session string. However, the padlock still checks the command using the valid session string. We used this weakness during reverse engineering to submit arbitrary session strings to the backend and discover details about the unlock command generation.

- Wrong alignment of the session string and offline key during encryption - The offline key is not XORed with the session key (the last 16 bytes of the session string) but with the first 16 bytes of the session string. This reduces the security level of the constructed preSessionKey to 112 bits, as it depends on static data. However, the `/unlock/` endpoint checks the authenticity of the whole session key. The `/unshackle/` endpoint allows generating an unshackle command using a session string with an arbitrary last four bytes.

- Undefined behaviour when submitting invalid command type in unlock packet - Invalid command type in the unlock packets caused the padlock to enter an undefined state. The padlock was returning success logs after invalid commands. We tried sending these invalid commands at a high rate but did not gain any advantage.

We did not find a possibility of a relay attack. One side of the attack is possible - even a remote attacker can read the session string from the padlock. However, the second attacker would need to control the user's app and request the unlock command from the backend without the user's knowledge. Attacking the app is out of the scope of this analysis.

5.5 Other attacks

Even though the main focus was on the security analysis of the protocol using methods defined in the theoretical part, we also found other vulnerabilities. They are out of the scope of this work, but we mention them because they are not to be neglected.

5.5.1 Brute-force attack on Quick click codes

The sequence of Quick click code is required to be between 3 and 24 symbols (taps) long. This creates a password with 3 to 24 bits of security, which opens the possibility of a brute-force attack.
McBride and col. [44] demonstrated an exploit of this feature on the previous padlock by the Noke, Inc. company (the lock is called just Noke Padlock). This padlock offers the same feature - only the sequence has to be between 8 - 16 symbols long. The results showed that the passwords created by the users have low entropy and that 87 % choose password lengths between 8 - 12. The researchers even

created a physical device to brute-force the Quick-click codes. They were able to crack 40 % of passwords in under 10 minutes and 93 % in under two hours.
The previous padlock has a feature to slow down the brute-force attacks - it does not accept any more tries for one minute after a certain amount of wrong attempts. This would be a good solution to improve the security of the Noke HD padlock, but there is no mention of such a countermeasure for this lock.

5.5.2 Attack on firmware

The padlock uses a Nordic Semiconductor nRF52840 chip, which is vulnerable to a fault injection (as well as the whole nRF52 series) [45]. This vulnerability allows the attacker to put the microchip into an error state, which turns off the chip's read-and-write protection, meaning that reading the firmware (firmware dump) is possible. The firmware of the padlock is a protected asset in the threat model, and the attacker can obtain valuable information from it (possibly even some hard-coded keys).
However, the attacker would need physical access to the internals of the padlock (either by breaking in by force or legitimately by unshackling and disassembling the padlock) and non-trivial equipment, which is needed for this kind of attack. These conditions rule out most of the attackers.

5.5.3 Elevation of privilege attack on the Geokey API

In the Communicating with the backend servers Section, we discovered a severe vulnerability in the Geokey API. An authenticated Geokey user can use the `/api/keys/` endpoint to request MAC addresses, unlock commands and offline keys for the padlocks of 1352 other users. Each offline key - unlock command pair has an assigned revocation status. The endpoint reacts similarly to the `/keys/` endpoint of the Noke Core API, which is used to display, issue and revoke offline keys. By sending a request with an empty JSON in the body, the user can obtain all offline keys issued to his account. Sending a similar request to the Geokey's `/api/keys/` endpoint leaks this list of

MAC addresses, unlock commands and offline keys. We tested the `/api/keys/` endpoint to revoke the offline keys of one of our padlocks, but the attempt was not successful. The offline key's revocation status in the list did not change, and we could still unlock the padlock with the offline key. The attempt to issue new offline keys was also unsuccessful.
The attack on the vulnerability can be classified as an elevation of privilege because the attacker is accessing assets requiring higher privileges.
A network attacker can unlock any padlock contained in the leaked data. The padlock can be recognised by its MAC address. When near the padlock, the attacker can read the session string from the padlock, use the Noke Mobile Library to encrypt the unlock command using the offline key, and successfully unlock the padlock. The network attacker may also store his own unlock command and offline unlock and use the padlock without depending on the Geokey and Noke backend servers and the Geokey app.
An insider attacker with access to the Geokey server can access the database of the keys and the commands and unlock any lock from the database. We are discussing the responsible disclosure process in the Responsible disclosure section.
To mitigate the vulnerability, Geokey can either close the `/api/keys/` endpoint or implement authorisation (return the offline keys and unlock commands only for the padlocks the user owns).

5.6 Responsible disclosure

We informed the Geokey company about the vulnerable API endpoint `/api/keys/` on May 17. We summarised the findings, the impact, and the potential mitigation of the vulnerability. The company responded the same day, saying the endpoint had been fixed. After the fix, the endpoint does not return a list of all padlocks and their offline keys and unlock commands. It responds with an error 400 (Mac address not found). However, it is still possible to request the data individually for each padlock by specifying the MAC address of the padlock in the request.
The Geokey company was notified about the remaining vulnerability,

and it was fixed on May 20. Now, the endpoint responds with an error 403 (You are not authorized to generate offline keys) when requesting other user's data.

5.7 Conclusion of the security analysis

The unlocking protocol has a good architecture that protects it against basic attacks on authentication protocols and smart locks. The crucial part of the solution is that cryptographic keys used to open the padlock are stored in the backend servers. This means that controlling the app does not bring the attacker any advantage. Another essential part of the solution is the session string, which authenticates the padlock to the backend servers and binds commands to specific sessions.

However, the user has to trust the reseller and the Noke company. The reseller can open any customers' padlocks with the offline unlock feature. Once the reseller has the offline keys, Noke Core servers are no longer needed to open the padlocks. The Noke company can naturally do the same.

The last note is that good architecture still leaves space for faulty implementation on the platform. We have found a severe vulnerability in the API in the Elevation of privilege attack on the Geokey API section. Several weaknesses were found in the MITM and relay attack, also caused by a faulty implementation. These weaknesses, in particular, raise questions about the "rigorous third-party testing" advertised by the Noke company.

6 Conclusion

The thesis analysed the design and security of the Noke HD smart padlock and made the information available to the public.
In the design analysis, we were able to reverse-engineer most of the unlocking protocol, which does not have publicly available documentation. We described how the unlock command is generated and encrypted in the backend servers and how the padlock verifies its authenticity. We discussed the possible reasons for the design choices. We were not able to reverse engineer the generation process of the session string, which is generated by the padlock for each Bluetooth connection. Apart from the unlocking, other features of the platform were also analysed.
The security analysis assessed the security of the protocol against common attacks on authentication protocols - replay, brute-force, MITM and relay attacks as well as the security of the whole platform against other specific attacks.
The padlock was considered secure against trivial replay and brute-force attacks. One severe vulnerability was found in the API of the company that resells the padlock. Missing authorisation checks allowed an authenticated user to request credentials for many other users' locks. The company was notified, and currently, the vulnerability is fixed. We also found several minor weaknesses in the padlock firmware and the backend servers' code, which were likely caused by faulty implementation. We did not find any possible attack exploiting these weaknesses.
The thesis brings more open-source knowledge for the Noke HD padlock and its platform. It can serve as a material for customers considering purchase of the padlock. It is a contribution to the general topic of smart locks as well.

6.1 Future work

This work has a limited scope, as described in Section Scope of the security analysis, which leaves possibilities for future research.
The security of the Bluetooth protocol implementation in the padlock

can be tested. This includes the pairing process, the encryption on the Bluetooth layer, etc. The API (and the backend server itself) and the mobile app were also omitted in this work and are possible subjects for thorough security analysis. The API's authentication mechanisms, resistance against DoS attacks, and scanning can be tested. The decompiled code of the app can be examined in detail, as well as the log-in mechanism.

Research can also focus on the padlock's microchip. This means performing side-channel attacks to extract secrets or attempting to read the firmware of the padlock using the known vulnerability (see Attack on firmware Section).

Another possible direction is to test the padlock's physical security.

Bibliography

1. COOPER, Verena. *Top 10 Cyber Security Trends And Predictions For 2024* [online]. San Francisco, California: Splashtop Inc., 2024-04-01 [visited on 2024-05-05]. Available from: `https://www.splashtop.com/blog/cybersecurity-trends-and-predictions-2024`.
2. VASUDEVAN, Vinod; ZAKHOUR, Zeina; GOMES, Vasco. *Top 10 cybersecurity threats in 2024* [online]. Bezons, France: Eviden SAS, 2024-02-29 [visited on 2024-05-05]. Available from: `https://eviden.com/publications/digital-security-magazine/cybersecurity-predictions-2024/top-10-cybersecurity-threats`.
3. AGARWAL, Aparna. *The Top 5 Cybersecurity Threats and How to Defend Against Them* [online]. Schaumburg, Illinois: ISACA, 2024-02-27 [visited on 2024-05-05]. Available from: `https://www.isaca.org/resources/news-and-trends/industry-news/2024/the-top-5-cybersecurity-threats-and-how-to-defend-against-them`.
4. *Noke HD and HD+ Bluetooth Smart Padlock* [online]. Temple, Georgia: Janus International Group Inc., 2024 [visited on 2024-03-12]. Available from: `https://www.janusintl.com/hd-padlocks`.
5. SILVERIO-FERNÁNDEZ, Manuel. What is a smart device? - a conceptualisation within the paradigm of the internet of things. *Visualization in Engineering* [online]. 2018, vol. 6, no. 3 [visited on 2024-05-05]. ISSN 2213-7459. Available from: `https://viejournal.springeropen.com/articles/10.1186/s40327-018-0063-8`.
6. *Britannica Dictionary definition of LOCK* [online]. Chicago, Illinois: Encyclopædia Britannica, Inc., 2024 [visited on 2024-05-05]. Available from: `https://www.britannica.com/dictionary/lock`.
7. *Yale Doorman* [online]. Stockholm, Sweden: ASSA ABLOY, 2024 [visited on 2024-05-05]. Available from: `https://www.yalehome.com/hr/en/products/smart-locks/yale-doorman`.

8. *Glue Smart Door Lock Pro* [online]. London, England: Glue AB, 2021 [visited on 2024-05-05]. Available from: `https://www.gluehome.com/products/smart-pro`.

9. *Noke Mobile Library for Android* [online]. Lehi, Utah: Noke, Inc., 2018 [visited on 2024-03-14]. Available from: `https://github.com/noke-inc/noke-mobile-library-android`.

10. ALURI, Diamond Celestine. Smart Lock Systems: An Overview. *International Journal of Computer Applications* [online]. 2020, vol. 177, no. 37 [visited on 2024-05-05]. ISSN 0975-8887. Available from: `https://ijcaonline.org/archives/volume177/number37/aluri-2020-ijca-919882.pdf`.

11. *OWASP IoT Top 10 2018 Mapping Project* [online]. Wilmington, Delaware: OWASP Foundation, Inc., 2018 [visited on 2024-05-05]. Available from: `https://github.com/scriptingxss/OWASP-IoT-Top-10-2018-Mapping`.

12. LOUNIS, Karim; ZULKERNINE, Mohammad. Bluetooth Low Energy Makes "Just Works" Not Work. In: *Cyber Security in Networking Conference*. Quito, Ecuador, 2019. Available also from: `https://hal.science/hal-02528877`.

13. PASKNEL, Victor. *Hacking the Nokē Padlock* [online]. San Francisco, California: A Medium Corporation, 2017-04-13 [visited on 2024-05-05]. Available from: `https://medium.com/@pasknel/hacking-the-nok%C4%93-padlock-adfe7b1b5617`.

14. PASKNEL, Victor. *Totally Pwning the Tapplock Smart Lock* [online]. Buckingham, United Kingdom: Pen Test Partners, 2018-06-13 [visited on 2024-05-05]. Available from: `https://www.pentestpartners.com/security-blog/totally-pwning-the-tapplock-smart-lock/`.

15. ROSE, Anthony; RAMSEY, Ben. *Picking Bluetooth Low Energy Locks from a Quarter Mile Away* [online]. Las Vegas, Nevada: Merculite Security, 2016 [visited on 2024-05-05]. Available from: `https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Rose-Ramsey-Picking-Bluetooth-Low-Energy-Locks-UPDATED.pdf`.

16. CABALLERO-GIL, Cándido; ÁLVAREZ, Rafael; HERNÁNDEZ-GOYA, Candelaria; MOLINA-GIL, Jezabel. Research on smart-locks cybersecurity and vulnerabilities. *Wireless Networks*. 2023. ISSN 1572-8196. Available from DOI: `10.1007/s11276-023-03376-8`.

17. MUNRO, Ken. *Smart Locks: Dumb Security* [online]. Buckingham, United Kingdom: Pen Test Partners, 2018-08-31 [visited on 2024-05-05]. Available from: `https://www.pentestpartners.com/security-blog/smart-locks-dumb-security/`.

18. LODGE, David. *Hardware reverse engineering. A tale from the workbench* [online]. Buckingham, United Kingdom: Pen Test Partners, 2018-06-22 [visited on 2024-05-05]. Available from: `https://www.pentestpartners.com/security-blog/hardware-reverse-engineering-a-tale-from-the-workbench/`.

19. RAY. *Lockpicking in the IoT* [online]. Hamburg, Germany: Chaos Computer Club e.V, 2016-12-27 [visited on 2024-05-05]. Available from: `https://media.ccc.de/v/33c3-8019-lockpicking_in_the_iot#t=49`.

20. LODGE, David. *Different 'smart' lock, similar security issues* [online]. Buckingham, United Kingdom: Pen Test Partners, 2019-02-18 [visited on 2024-05-05]. Available from: `https://www.pentestpartners.com/security-blog/different-smart-lock-similar-security-issues/`.

21. LODGE, David. *Pwning the Nokelock API* [online]. Buckingham, United Kingdom: Pen Test Partners, 2019-05-24 [visited on 2024-05-05]. Available from: `https://www.pentestpartners.com/security-blog/pwning-the-nokelock-api/`.

22. KNIGHT, Edward; LORD, Sam; ARIEF, Budi. Lock Picking in the Era of Internet of Things. In: *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, pp. 835–842. Available from DOI: `10.1109/TrustCom/BigDataSE.2019.00121`.

23. *Geokey and Noke Integration* [online]. Fremont, Nebraska: Geokey, Inc., 2024 [visited on 2024-03-12]. Available from: `https://geokeyaccess.com/geokey-and-noke-hardware-solution/`.

24. *NoKe HD Padlocks* [online]. Kentwood, Michigan: Movatic, 2024 [visited on 2024-05-17]. Available from: `https://shop.movatic.co/products/noke-hd-padlocks-licenses`.

25. *Noke Heavy-Duty Padlock* [online]. Auckland, New Zealand: Astute Access Group Ltd, 2023 [visited on 2024-05-17]. Available from: `https://www.astutesmartlocks.com/noke-hd-padlock.html`.

26. *Alexela Smart Trailer* [online]. Tallinn, Estonia: AS Alexela, 2024 [visited on 2024-05-17]. Available from: `https://www.alexela.ee/en/nutihaagis`.

27. *EN 12320:2012 standard* [online]. Brussels, Belgium: European Committee for Standardization, 2012 [visited on 2024-05-05]. Available from: `https://standards.iteh.ai/catalog/standards/cen/191a7896-a71b-4816-a47d-1017ddfb9d1c/en-12320-2012`.

28. *Noke Core API Documentation* [online]. Lehi, Utah: Noke, Inc., 2018 [visited on 2024-03-14]. Available from: `https://github.com/noke-inc/noke-core-api-documentation`.

29. *JADX* [online]. skylot, 2024 [visited on 2024-05-05]. Available from: `https://github.com/skylot/jadx`.

30. *Verifying and Debugging - Debugging with logs* [online]. Mountain View, California: Google LLC, 2022-04-29 [visited on 2024-05-05]. Available from: `https://source.android.com/docs/core/connect/bluetooth/verifying_debugging#debugging-with-logs`.

31. *Android Debug Bridge (adb)* [online]. Mountain View, California: Google LLC, 2022-02-09 [visited on 2024-05-05]. Available from: `https://developer.android.com/tools/adb`.

32. *Wireshark - Go Deep* [online]. Davis, California: Wireshark Foundation, 2024 [visited on 2024-05-05]. Available from: `https://www.wireshark.org/`.

33. FARANDA, Emanuele. *PCAPdroid* [online]. 2024. [visited on 2024-05-05]. Available from: `https://github.com/emanuele-f/PCAPdroid`.

34. WIŚNIEWSKI, Ryszard. *Apktool* [online]. 2024. [visited on 2024-05-05]. Available from: `https://ibotpeaches.github.io/Apktool/`.

35. MARISETTY, Suresh. *Five Steps to Successful Threat Modelling* [online]. Cambridge, United Kingdom: Arm Limited, 2019-01-10 [visited on 2024-05-05]. Available from: `https://community.arm.com/arm-community-blogs/b/internet-of-things-blog/posts/five-steps-to-successful-threat-modelling`.

36. OORSCHOT, Paul C. van. Authentication Protocols and Key Establishment. In: *Computer Security and the Internet*. Springer International Publishing, 2020, pp. 91–124. ISBN 9783030336493. ISSN 2197-845X. Available from DOI: `10.1007/978-3-030-33649-3_4`.

37. GRASSI, Paul A; GARCIA, Michael E; FENTON, James L. *Digital identity guidelines: revision 3*. 2017. Available from DOI: `10.6028/nist.sp.800-63-3`.

38. REGENSCHEID, Andrew; BEIER, Geoff. *Security best practices for the electronic transmission of election materials for UOCAVA voters*. 2011. Available from DOI: `10.6028/nist.ir.7711`.

39. GARFINKEL, Simson L. *De-identification of personal information*. 2015. Available from DOI: `10.6028/nist.ir.8053`.

40. BARKER, Elaine. *Recommendation for key management:: part 1 - general*. 2020. Available from DOI: `10.6028/nist.sp.800-57pt1r5`.

41. TOWNSEND, Kevin. *Introduction to Bluetooth Low Energy - GATT* [online]. New York, New York: Adafruit Industries, 2024-03-08 [visited on 2024-05-06]. Available from: `https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt`.

42. *Device Firmware Update process* [online]. Trondheim, Norway: Nordic Semiconductor ASA, 2019-04-08 [visited on 2024-05-06]. Available from: `https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v15.3.0/lib_bootloader_dfu_process.html`.

43. *HD and HD+ padlock sell sheet* [online]. Lehi, Utah: Noke, Inc., 2018 [visited on 2024-05-06]. Available from: `https://www.janusintl.com/hubfs/Nok%C4%93%20HD%20Padlocks/HD%20AND%20HD+%20padlock%20sales%20sheet.March%202020.pdf?hsLang=en`.

44. MCBRIDE, Jack; HERNANDEZ-CASTRO, Julio; ARIEF, Budi. Earworms Make Bad Passwords: An Analysis of the Nokē Smart Lock Manual Override. In: *2017 International Workshop on Secure Internet of Things (SIoT)*. 2017, pp. 30–39. Available from DOI: `10.1109/SIoT.2017.00009`.

45. *Information Notice IN-133 v1.0* [online]. Trondheim, Norway: Nordic Semiconductor ASA, 2020-06-12 [visited on 2024-05-06]. Available from: `https://infocenter.nordicsemi.com/pdf/in_133_v1.0.pdf`.

A Python script emulating the Geokey App functionality

```python
import asyncio
from bleak import BleakClient
import json
import requests
import jwt
import time
import os
from Crypto.Cipher import AES

"""
1. Get the session string from lock
2. Obtain the unlock command from the backend
3. Send the unlock command
"""

# padlock information
RX_CHAR_UUID = "1bc50002-0200-d29e-e511-446c609db825"  # for sending commands
TX_CHAR_UUID = "1bc50003-0200-d29e-e511-446c609db825"  # for receiving notifications
STATE_CHAR_UUID = "1bc50004-0200-d29e-e511-446c609db825" # for reading session string
PADLOCK_MAC = ""

# API urls
GEOKEY_API_URL = ""
NOKE_CORE_API_URL = "https://coreapi-sandbox.appspot.com"

# API endpoints
UNLOCK_URI = "/api/unlock/"
UNSHACKLE_URI = "/api/unshackle"
UPLOAD_URI = "/upload/"
REFRESH_TOKEN_URI = "/api/RefreshToken"

# authentication tokens
MOBILE_API_KEY = ""  # for Noke Core API
GEOKEY_TOKEN = ""  # for Geokey API


logs = []


def callback(sender, data):
    with open("unlock_commands.txt", "a") as f:
        f.write(f"{data.hex()} ")
    logs.append(data.hex())


def refresh_token(current_token):
    print("Refreshing token...")
    headers = {
        "Authorization": f"Bearer {current_token}",
        "Content-Type": "application/json; charset=UTF-8",
        "User-Agent": "okhttp/5.0.0-alpha.2",
        "Accept-Encoding": "gzip",
        "Connection": "Keep-Alive",
        "Host": GEOKEY_API_URL[8:],
    }

    refresh_token = jwt.decode(
        current_token,
        algorithms=['HS256'],
        options={"verify_signature": False})["RefreshToken"]
    refresh_json = {
        "refresh_token": refresh_token,
        "token": current_token
    }
    headers["Content-Length"] = str(len(json.dumps(refresh_json)))

    refresh_response = requests.post(
        GEOKEY_API_URL + REFRESH_TOKEN_URI,
        json=refresh_json,
        headers=headers)
    new_token = refresh_response.json()["data"]["access_token"]

    with open("token.txt", "w") as f:
        f.write(new_token)

    return new_token


def get_unlock_command(token, session_string):
    print("Requesting unlock command from backend...")
    headers = {
        "Authorization": f"Bearer {token}",
        "Content-Type": "application/json; charset=UTF-8",
        "User-Agent": "okhttp/5.0.0-alpha.2",
        "Accept-Encoding": "gzip",
        "Connection": "Keep-Alive",
        "Host": GEOKEY_BACKEND_URL[8:],
    }

    unlock_json = {
        "mac": PADLOCK_MAC,
        "session": session_string.upper(),
    }
    headers["Content-Length"] = str(len(json.dumps(unlock_json)))

    return requests.post(
        GEOKEY_BACKEND_URL + UNLOCK_URI,
        json=unlock_json,
        headers=headers)


def upload_logs(session_string, logs):
    print("Sending logs to backend...")

    headers = {
        "Content-Type": "application/json",
        "Connection": "close",
        "charset": "utf-8",
        "Authorization": f"Bearer {MOBILE_API_KEY}",
        "User-Agent": "Dalvik/2.1.0",
        "Host": NOKE_BACKEND_URL[8:],
        "Accept-Encoding": "gzip",
    }

    logs_json = {
        "logs": [
            {
                "session": session_string.upper(),
                "responses": [log.upper() for log in logs if log[:2] == "50"],
                "mac": PADLOCK_MAC,
                "received_time": int(time.time()),
            }
        ]
    }

    headers["Content-Length"] = str(len(json.dumps(logs_json, separators=(',', ':'))))
    return requests.post(
        NOKE_BACKEND_URL + UPLOAD_URI,
        json=logs_json,
        headers=headers)


async def main():
    client = BleakClient(PADLOCK_MAC)
    print("Connecting to the lock...")
    await client.connect()
    print("Getting session string...")
    session_string = await client.read_gatt_char(STATE_CHAR_UUID)
    if not session_string:
        exit(1)

    session_string = session_string.hex().upper()
    print(f"Session string: {session_string}")

    if not os.path.is_file("token.txt"):
        with open("token.txt", "w") as f:
            f.write(GEOKEY_TOKEN)

    with open("token.txt", "r") as f:
        cognito_token = f.readline().strip()

    unlock_response = get_unlock_command(cognito_token, session_string)

    if unlock_response.status_code != 200:
        if unlock_response.status_code == 401:
            print(unlock_response)
            cognito_token = refresh_token(cognito_token)
            unlock_response = get_unlock_command(cognito_token, session_string)
        else:
            print(unlock_response)
            exit(1)

    if unlock_response.status_code != 200:
        print(unlock_response.json())
        exit(1)
    try:
        unlock_command = unlock_response.json()["data"]["command"]
    except:
        print(unlock_response.json())
        exit(1)

    print(f"Unlock command: {unlock_command}")

    await client.write_gatt_char(RX_CHAR_UUID, bytes.fromhex(unlock_command))
    print("Unlock command sent to the lock")
    await asyncio.sleep(0.1)

    with open("unlock_commands.txt", "a") as f:
        f.write(f"{session_string.lower()},{unlock_command},")

    print("Reading notifications...")

    await client.start_notify(TX_CHAR_UUID, callback)
    await asyncio.sleep(5)
    await client.stop_notify(TX_CHAR_UUID)

    print("Disconnecting from the lock...")
    await client.disconnect()

    upload_logs(session_string, logs)

    with open("unlock_commands.txt", "a") as f:
        f.write("\n")


asyncio.run(main())
```