Lock Picking in the Era of Internet of Things

Edward Knight, Sam Lord, and Budi Arief

School of Computing, University of Kent, United Kingdom
edw@rdknig.ht, hello@samlord.me, b.arief@kent.ac.uk

***Abstract*—Smart locks are a recent development in the Internet of Things that aim to modernise traditional key-based padlock systems. They allow users to operate the lock with their smartphone instead of carrying around a physical key. Typically, smart locks have a cloud system for sharing access with other people, which makes them ideal for schemes such as communal lockers or bike sharing. One of the smart locks available on the market is that produced by Master Lock. They are an established brand, and unlike many of the single product companies that have provided insecure offerings, Master Lock have so far shown that their locks are reasonably secure and resistant to known attacks such as shimming, fuzzing, and replay attacks. This paper provides a security analysis of the Master Lock Bluetooth padlock. More importantly, it reveals that there were several security vulnerabilities, including a serious one in the Application Programming Interface used by Master Lock to provide a crucial feature for managing access. We carried out a responsible disclosure exercise to Master Lock, but communication proved to be quite a challenge. In the end we managed to establish contact, and as a result the most serious vulnerabilities have now been patched. This indicates that responsible disclosure is a valuable exercise, but we still need better report-and-response mechanisms.**

Keywords: security, IoT, smart locks, API vulnerabilities, responsible disclosure.

I. Introduction

Internet of Things (IoT) devices are gaining popularity and becoming more ubiquitous in our lives. There has been a significant growth of IoT devices over the past few years [1], but this new market has gained a reputation for churning out insecure systems. This has led to security and privacy concerns (such as [2]–[4]) and even a large-scale Distributed Denial of Service attack using the Mirai botnet of compromised IoT devices [5].

Smart locks provide a modern alternative to the traditional key or combination lock, and access is generally managed by an online service. Typically the locks themselves do not connect to the internet directly, but communicate with a smartphone app, which in turn talks to the online service. This separation allows for locks to work offline, an important feature if the owner is without an internet connection or the online service becomes unavailable, but also increases the attack space. The smartphone can be thought of as the "key", allowing its user to lock, unlock and even share the smart lock with other users. Furthermore, these locks typically provide an *override mechanism* to allow their user to unlock it should the associated smartphone become unavailable (e.g. due to the smartphone being lost or stolen, or if the smartphone's battery has run out). In a sense, these override mechanisms provide a way to open the lock with "something the user knows", akin to a combination lock or a password.

Locks have traditionally been judged on their physical security; how well they stand up to picking, shimming and cutting into. Smart locks need to be physically secure as well as have a secure connection to a secure management system. The market of smart locks is mostly made up of small startup companies manufacturing a single product, and many of them have been proven to be insecure (for example [6] and [7]). The work presented in this paper looks at the security of a smart lock built by an established lock manufacturer (Master Lock, https://www.masterlock.com/) to see if they have better security than other (relatively new) companies.

The key contributions of this paper are: (i) a thorough and systematic security analysis of the Master Lock Bluetooth padlock, leading to (ii) the revelation of several vulnerabilities, including a serious security issue with the Application Programming Interface (API) used, and (iii) an impactful responsible disclosure exercise, which ended up with the manufacturer of Master Lock patching their system to address the vulnerabilities we uncovered.

Section II provides an overview of related work in investigating security issues in smart locks. Section III introduces the Master Lock Bluetooth padlock, including its system and communication architecture. Section IV outlines our methodology for finding potential security vulnerabilities in Master Lock, along with an attacker model. Section V presents and discusses the results of our investigation, as well as our responsible disclosure. Section VI concludes our paper and provides some ideas for future work.

II. Related Work

Limited prior research has been undertaken regarding the security of smart locks. Nevertheless, there is a wide range of potential attack vectors. These include physical attack, wireless communication sniffing and spoofing, as well as exploiting the supporting back end services that enable many of the "smart" features of smart locks.

There is a wealth of related work in IoT security, some of which has informed our research here. An early paper by Suo et al. [8] provides an overview of four key layers of typical IoT architecture (perceptual, network, support and application layers). Of particular interest to us is the support layer, in which centralised control functionality (such as access management) is implemented. The challenge here is how to detect malicious activity, such as commands that should not be executed due to insufficient privileges, but may be performed anyway due to improperly implemented access control.

Wurm et al. [9] performed security analyses on consumer and industrial IoT devices. Some of the techniques used (hardware and network communication analysis, reverse engineering) gave us some initial direction when looking at Master Lock's offering.

We found that most of the investigation into breaking smart lock security has come from hacking communities, with scarce academic papers looking into the potential security vulnerabilities of smart locks. One example of the latter is a paper on the Nokē smart lock [10]. The researchers focused on the manual override feature of the Nokē lock and studied human-generated unlock codes. They showed that the human-generated codes were of low enough entropy to pose a serious vulnerability. A mechanical brute-force tool was built to demonstrate this.

A DEF CON presentation on hacking Bluetooth locks [6] specifically stated that two locks (from Nokē and Master Lock) had not been cracked, showing that neither padlock had trivially broken security. This provided a challenge and a list of common attacks on smart locks to focus on:

- *Plain-text passwords*: Bluetooth transmission of passwords without encryption
- *Replay attacks*: where a recording of the communication can be replayed to unlock the device
- *Fuzzing*: sending invalid data can cause the device to enter an error state and open
- *Device spoofing*: using a computer to pretend to be the lock, we can listen to commands sent by a phone and use them to unlock the real device
- *Hard-coded secrets*: some lock manufacturers include hard-coded secrets in the smartphone application, which can be retrieved by decompiling it

Another presentation from DEF CON included the full methodology used to hack an August Smart Lock installed in a front door [7]. This was of particular interest since the methods used attacked both the Bluetooth communication layer and the supporting cloud service, which is similar in architecture to the Master Lock service. They looked at:

- *Investigating the cloud service API*: checking if users can get privileged data or perform privileged actions from a guest account

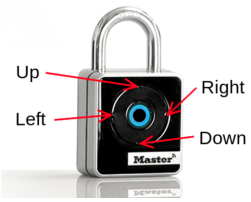

Figure 1. Master Lock and its override buttons

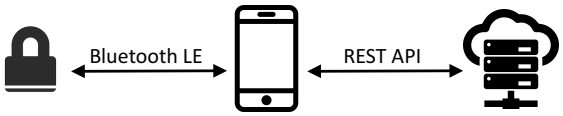

Figure 2. Master Lock components and their communication

- *Man-in-the-Middle*: by hosting a service which sits between the phone and the real cloud service, it is possible to modify requests and responses
- *Reading the firmware*: downloading the firmware for the device from the cloud service and looking for any hard-coded secrets

Additionally, after mapping out the API, the author made a Software Development Kit (SDK) for accessing the API. This SDK made it easier for them to investigate the methods provided by the API and to search for potential vulnerabilities.

Master Lock have registered patents for the lock itself [11], managing the authentication keys over wireless communication [12] and for wireless firmware updates [13]. This indicated to us that Master Lock has put some effort into designing these systems, and also gave us a starting point for areas to investigate.

III. System of Interest: Master Lock

We chose to focus on Master Lock as it is an established brand specialising in various lock and padlock systems. We had hoped that this industry experience would result in a more secure system than their competitors.

A. System architecture

The Master Lock Bluetooth padlock is a Bluetooth enabled padlock which comes in two form factors for indoor and outdoor usage. It has the look and feel of an ordinary padlock but instead of a keyway, it has a 4-directional pad that serves as the input device for the override mechanism (see Figure 1).

The overall Master Lock system has three key components (Figure 2): the padlock, the smartphone application, and the cloud service. These components communicate to each other by using two protocols: Bluetooth Low Energy (BLE) and HTTP, using the smartphone application as the interlinking component.

The padlock has the Slave Role, and communicates with the smartphone, which gets the Master Role [14, Vol 6, Part B, Section 1.1, p. 2553]. This means that the padlock

advertises itself when it turns on, on one of the three advertising channels. The phone will periodically scan for Bluetooth devices, and initiate a connection with the lock when it is available.

The Master Lock Vault eLocks application runs as a service in the background on the phone, and sends encrypted commands to unlock the padlock when connected. It has to query the cloud service to discover the locks which the user has access to. In the application interface, a user will either see the current valid temporary code if they are a guest, or the primary code if they own the lock. Furthermore, the owner is able to share future temporary codes which they can request from the cloud service (see Section III-C).

B. Unlocking the padlock

Before using the padlock, a user must install the Master Lock Vault eLocks app on their phone. They can then add the device to their Master Lock account by entering the code which is printed on a sticker attached to the packaging. This makes this user the owner of the padlock and gives their account full access to control the device.

The Master Lock Bluetooth padlock has two methods to unlock: using the smartphone eLocks app with Bluetooth LE communication, or entering a combination on the directional pad (the latter as an override mechanism).

To use a smartphone to unlock the padlock, the user can simply press any direction on the directional pad while the Master Lock Vault eLocks application is open. This sometimes works when the application is not open, but seems to be dependent on the smartphone keeping the Bluetooth radio active.

To unlock manually, the user needs to enter a combination on the directional pad on the front of the lock (which is similar to a password). If the combination is correct, it will unlock as soon as the last input is entered. Otherwise, the device will continue listening to inputs for a few seconds before the LED turns red, indicating an invalid code.

There are three different types of combination:

- Primary code (the owner of the device sets this)
- Backup Master code (the owner of the device knows this, and it cannot be changed)
- Temporary code (used for time-limited guest or shared access)

C. Sharing access

Sharing lock access is a big feature for smart locks, since a selling point is that you can grant temporary access without giving someone a key, and can revoke access at any time. This means you do not have to trust the guest to give the key back, and there's no risk of them making a physical copy.

To share the device with other users, one can either share a temporary code for a specific time slot, or add a guest user to the lock. When sharing a temporary code, the Master Lock Vault eLocks application uses the default sharing system of the smartphone's operating system, so it can share the code through any messaging app of the user's choice. Adding a guest user requires them to have a Master Lock Vault account.

Temporary codes are valid for 8 hours, and are generated every 4 hours at 12am, 4am, 8am, 12pm, 4pm, and 8pm. This means that there are two valid temporary codes at any time. The padlock does not have to be connected to the smartphone app to know which temporary codes are valid, but will fall out of sync if the battery is removed. When the owner generates a temporary code, they can either choose to get the current code, or a code valid at a specified time in the future.

The owner can limit the guest's access to day-time (7am-7pm) or night-time (7pm-7am). The guest will only be able to receive temporary codes during those hours, but the validity of the codes is still the same. This means that a guest with access limited to day-time will be able to get the code generated at 4pm, which will be valid until midnight. Similarly a guest with access at night-time will be able to get the code generated at 4am, which will be valid until midday.

IV. Methodology

When we started our research, we spent some time investigating traditional locks and methods of picking them. While a "smart lock" does not have a keyway, it still has many of the physical components of a traditional padlock, meaning that it shares some of the same vulnerabilities. Some padlocks made by startup companies suffer from technical weaknesses, and some had poor build quality or were not resistant to simple physical attacks [6]. When we experimented with lock picking, we found that cheaper Master Lock padlocks were very easy to pick—even for novices. We theorised that vulnerabilities may translate to their smart padlock security as well.

Others have carried out tests regarding the physical robustness of the Master Lock Bluetooth padlock. For example, [15] shows that striking the lock with a hammer can cause it to fall apart. We are not interested to repeat this investigation, so we looked at other ways to tamper with the lock and gain illicit access.

This section outlines various methods we explored in order to find security vulnerabilities in Master Lock. These techniques are also generally applicable to other IoT devices, and may serve as a template methodology for investigating potential vulnerabilities in other IoT systems with a similar architecture.

- We looked at a possibility of *physical tampering* by disrupting the power source to see if this would cause inconsistency in the internal clock.
- We investigated the likelihood of a *brute force attack* to guess the override pattern to unlock the padlock.
- We used a *Bluetooth packet sniffer* to investigate the communication between the phone and the padlock.
- We *decompiled the Android app* to check for hard-coded encryption keys and secrets, and to find out how it communicated with the padlock and the cloud.
- We investigated the *cloud service's REST API* to see if we could extract information about a lock and its owner.

A. RTC manipulation

The padlock has a Real-Time Clock (RTC) which it uses to validate temporary codes. Based on information from a teardown video [16], we saw that the padlock uses a Texas Instruments MSP430 chip [17]. There is no other hardware that could keep track of real time, so we conclude that the padlock uses the RTC function on this chip.

There is only one power source in the padlock—the coin cell battery. So the first thing we could do was to remove the battery and see how that affected the operation of the lock.

B. Brute-force attack

There are possible brute-force attacks based on the directional-pad input. For example, it is possible to build a rig to attach a mechanical brute-force device to the lock, similar to the work presented by [10]. This brute-forcer can be programmed to enter multitudes of codes to open the lock using the manual override.

There are three types of code, each of different length, with the temporary code always starting with *right*, while the primary and master backup codes start with *up*.

When a human is inputting codes, it is possible to enter 3 codes in roughly 20 seconds, after which the device will "lock-out" and not accept any more inputs for 60 seconds. This time does not vary by much when entering different length codes, as the majority of the time is spent waiting for the directional pad to re-activate after each incorrect code. This means that on average we can enter one code every 26.7 seconds. It is possible that a rig would enter codes faster than this, but this is adequate for a benchmark.

C. Bluetooth security

We briefly looked at the security of the Bluetooth communication between the Master Lock and the smartphone companion app. We recorded the communication between the phone and the lock using Wireshark [18] and a Bluetooth packet sniffer. Recording this was tricky because the smartphone sometimes fails to detect the lock and connect properly.

D. Application security

Master Lock have made apps for Android and iOS. We tested the Android version[1], as it is easier to decompile.

1) Temporary codes

We found that when sharing a temporary code, the interface advertises that the codes will be valid for 4 hours. However, the message generated to actually share the code says it expires in 8 hours. This peculiar behaviour showed that there was some amount of complexity in the temporary code system, making it a good candidate for vulnerabilities. To find out more about the generation of temporary codes, we needed to know more about how the application works.

2) Decompiling

The Android Package Kit (APK) of the app had not been obfuscated, so decompiling produced mostly easy-to-read Java source code. Apktool [19] was used to extract metadata and the bytecode in DEX format. dex2jar [20] converted the bytecode to JAR format, and JD-GUI [21] decompiled this to source code.

Having access to the source code of the app allowed us to look at the communication protocols between both the padlock and the phone, and the phone and the server.

3) Dumping app data

With a rooted phone, we managed to dump the app's data store. In the shared preferences folder, we found the file `com.masterlock.ble.app.xml`. This contains two pieces of sensitive data: "authToken" and "dbToken". The "authToken" is used along with the username to access the API. The "dbToken" is the password for an encrypted database. We also found this same password in the decompiled APK source code.

In the databases folder there is `masterlock.db`, an SQLite database encrypted by SQLCipher [22]. Using the "dbToken", we decrypted the database and found that it stores all of the data that the app uses. Looking at the database access in the source code, it seems like most – if not all – of this data is just cached API queries.

E. API security

The Android app uses the Retrofit library [23] for communicating with the server's REST API. The API calls all start with an explicit version number. Of the API calls listed in the app source code, there are 44 v4 calls and 14 v5 calls. The app only communicates with the API over HTTPS, however, the API itself is accessible over plain HTTP when accessed through other means. This means that the passwords and keys are being transmitted

[1] https://play.google.com/store/apps/details?id=com.masterlock.ble.app version 2.1.2.4

in plaintext when HTTP is used instead of HTTPS. We built a Python script to aid our testing, which is available on Github[2].

1) Generating an API key

In most of the requests with the API, the server authenticates a user by both username and API key. If a user does not already have an API key (this can be found in the Android app's shared preferences folder), the first call they need to make is to "v4/account/authenticate/". This request requires an API key as parameter, but one can use "androidble" or "iosble" in place of an actual key. The body of the request is a JSON object containing the username and password. This is the only API call that requires a password – the API key is used like a password in all other requests. Sample Python code for performing this request can be seen below:

```
def get_api_key(username, password):
    """API call to get an API key to use in future requests."""
    method = "POST"
    url = BASE_URL + "v4/account/authenticate/"
    parameters = {"apikey": "androidble"}
    body = {
        "username": username,
        "password": password
    }
    response_json = call_api(method, url, parameters, body)
    return response_json["Token"]
```

2) Getting padlock details

We have found one API call in particular that returns a lot of information, from which we can extract some interesting values:

```
def get_products(username, api_key):
    """API call to get a list of products registered with the account."""
    method = "GET"
    url = BASE_URL + "v4/product?complex=true"
    parameters = {
        "username": username,
        "apikey": api_key
    }
    response_json = call_api(method, url, parameters)

    products = []
    for product in response_json:
        products.append({
            "name": product["Name"],
            "product_id": product["Id"],
            "device_id": product["KMSDevice"]["DeviceId"],
            "KMS_id": product["KMSDevice"]["Id"],
            "latitude": product["KMSDevice"]["Location"]["Latitude"],
            "longitude": product["KMSDevice"]["Location"]["Longitude"],
            "primary_code": product["KMSDevice"]["PrimaryCode"],
            "model_id": product["Model"]["Id"],
            "model_name": product["Model"]["Name"],
            "model_number": product["Model"]["ModelNumber"],
            "model_SKU": product["Model"]["SKU"]
        })
    return products
```

This single API call gives us all the information about all products registered to the account. Further API requests about individual padlocks require the "KMS id". We get the GPS coordinates of the last known location of the lock in this request, as well as the user-set primary code that was discussed earlier. Even a guest user can get the primary code using this API call, which should only be accessible to the owner.

[2]https://github.com/Edward-Knight/master_lock_api

3) Generating temporary codes

The Android app does not contain the code to generate temporary codes, so it has to ask the server to generate them. Temporary codes can be generated for any time in the future in 4 hour intervals. The latest code that can be generated is in the year 9999, before the year rolls over to 5 digits.

```
def generate_temporary_code(username, api_key, kms_id, access_time=None):
    """
    API call to generate a temporary code. If access_time is None,
    gets a currently active code.
    """
    method = "GET"
    url = BASE_URL + "v4/kmsdevice/" + kms_id + "/servicecode/"
    parameters = {
        "username": username,
        "apikey": api_key
    }
    if access_time is not None:
        parameters["accessTime"] = access_time
    response_json = call_api(method, url, parameters)
    return response_json["ServiceCode"]
```

As with a lot of "KMS id" specific API calls, the account username must be registered with the product. For this API call, one can be registered as a guest with the padlock.

4) Sending password reset emails

Another API request of interest is to send password reset emails. In the app, this is triggered when the user selects "forgot password" when authenticating. There is a similar API call to send an email reminding the user what their username is.

```
def forgot_password(email):
    """
    API call that sends a password reset email to the specified address.
    Returns True if successful and False otherwise.

    Call this in a loop for a Denial of Service attack.
    """
    method = "POST"
    url = BASE_URL + "v4/account/resetpassword"
    parameters = {
        "apikey": "androidble"
    }
    body = {"email": email}
    try:
        response_json = call_api(method, url, parameters, body)
        return isinstance(response_json, dict) \
            and "ServiceResult" in response_json \
            and response_json["ServiceResult"] == 1
    except MasterLockError:
        return False
```

Of note is the fact that one does not need to be authenticated to use this API call—they only need to supply the user's email address. If the email is recognised, Master Lock will send an email to that address. Otherwise the request will return an error code. This also means that this method can be used to check if an email address is associated with a Master Lock account.

F. Attacker model

There are a couple of potential scenarios where an attacker may be able to compromise the security of a Master Lock device:

- The simplest scenario of attack involves a guest user exploiting the security vulnerability in the REST API service to generate many temporary codes that will be valid into the future. This means that the attacker will still be able to unlock the padlock, even after their access has been revoked.

Table I
STATISTICAL ANALYSIS OF THE FREQUENCY OF BUTTON PRESSES BASED ON THE HARVESTED TEMPORARY CODES

Direction	Count	Percentage
Up	21073	25.3%
Down	20816	25.0%
Left	20848	25.0%
Right	20633	24.7%

- For a potential attacker who has not been added as a guest user of the lock, the likely scenario is for them to carry out a brute-force attack on the override mechanism using the 4-key directional pad.

As this kind of lock is typically used in sharing schemes, the vulnerabilities uncovered through our research pose a real threat to the security of anyone using Master Lock, as discussed in the next section.

V. RESULTS AND DISCUSSION

Using the techniques and approaches outlined in Section IV, we gained valuable insights into how Master Lock works, and we uncovered several security vulnerabilities with varying levels of severity.

A. RTC manipulation result and discussion

We found that when the battery is removed, the RTC is effectively "paused", meaning it falls behind the real time. When the battery is re-inserted, the RTC resumes from where it left off. The RTC is corrected when it communicates with a phone again. Of course, this "correction" process just uses the phone's internal clock. After testing, we discovered that the lock's RTC is updated to whatever the time on the phone is. This updating only happens when communicating with the owner's phone, not a guest's.

We have managed to successfully exploit this behaviour to get the padlock to validate expired temporary codes. However, either the padlock has to be unlocked (in order to access the battery compartment) or the attacker needs to use the owner's phone, so this is not a feasible attack.

B. Brute-force attack result and discussion

Sampling temporary codes from the API shows that they all start with a *right* press – no other patterns were found. Out of 11,910 temporary codes, the expected occurrence of each direction (not including the initial right press) is $(11910 * 7)/4 = 20842.5$. Our statistical analysis (Table I) shows that the temporary codes are uniformly distributed.

Brute-forcing the backup master code would be the most valuable, as the user cannot change it. However it is also the most complex, with an attack space of 4^{10} codes (4 directions, code of length 11 but always starts with *up*). This would take almost a year to attack, which is not practical.

Brute-forcing the primary code has an attack space of 4^6 codes (4 directions, code length of 7 but starts with *up*). This would take a maximum of 30 hours to brute-force, which is much more reasonable.

Finally, brute-forcing the temporary codes has an attack space of $(4^7)/2$ (4 directions, code length of 8 but starting with *right*, but with two codes active at any time). This would take a maximum of 60 hours, but is not feasible as each temporary code is only valid for 8 hours.

C. Bluetooth sniffing result and discussion

Analysing the handshake between the lock and the phone, we saw some information transmitted in plaintext (e.g. the lock's firmware version). After a handshake is completed, the smartphone and padlock seem to move to encrypted communication.

We continued investigating the Bluetooth security at a low priority, as we had already seen that a wide-range of attacks had already been attempted on the Master Lock [6] without success. We have not found this aspect of the system to be vulnerable, but a more detailed analysis is required before we would be confident in calling it secure.

Details about the padlock which cannot be seen on the device were communicated in plaintext, including the firmware version.

D. Application and API dissecting result and discussion

Analysing the decompiled Android app, we were able to find the REST interface for the cloud service and the Bluetooth LE interface for the padlock. We also found the encryption key used for application's database, which we used to decrypt a dumped database. This contained some secrets about the lock, and some personal information.

We created a Python program to interact with the REST API, which is able to authenticate with a username and password and query the details for the account which the padlock is registered to. Accounts with guest association can access privileged information, such as the device location and the primary code. Guests can also generate temporary codes for times which should be restricted.

We have demonstrated the specific attacks outlined below.

1) Key invalidation and rate limiting

The API keys generated seem to never expire. This means that leaking an API key is as bad as leaking the user's password. However, we found out that a password change will invalidate the API keys.

In our testing, we also found that the server does not rate-limit calls, or blacklist IP addresses. On its own, this is not a security flaw, but it can be exploited in combination with other flaws to make certain attacks viable.

2) Guest access hours

Guests can be allowed to unlock the padlock at certain time intervals. They can do this through Bluetooth LE with their phone, or by a temporary code which rolls over every 8 hours. The app therefore has to be able to perform the "generate temporary code" API call with guest-level credentials. However, the guest is only restricted to the time when they can *call* the API. Because one can specify the time a temporary code should be active for, it is possible to generate temporary codes that can be used later. This effectively means the entire "access hour" restriction can be bypassed by pre-generating temporary codes far into the future.

We created temporary access codes for one of our locks that will be valid for a 10-year span from 2017-12-07 until 2027-12-05. A selection of these codes can be seen in Appendix A. We tested these codes for the appropriate time slots "in the future", and they worked as expected.

What we did not expect was that the temporary codes were still valid after the Master Lock device was unregistered from the original owner and registered with a new owner. We would have expected that this process would have invalidated all future temporary codes, but our experiment showed that they were still valid. This means that it is possible for anyone who previously had guest access to keep being able to unlock the device even after their access is revoked, which is a serious vulnerability.

3) Guest access level

All padlock-specific API calls require the account to be associated with the device in Master Lock's system. This association can be at two levels, "guest" and "owner". Some API calls will therefore return different data, or not work at all, if the user is without proper association. However, this is not correctly configured for "get product details". Even if one's account only has guest access for the padlock, this API call will return the primary code. This is the owner-set directional code used to unlock the padlock. Even if the owner revokes guest access and resets their encryption keys, as suggested on the Master Lock website[3], the primary code will stay the same. As an unusual password, the primary code is unlikely to ever be changed by the owner, and likely to be the same on all the locks they own.

E. Responsible disclosure

We carried out a responsible disclosure exercise by submitting our findings to Master Lock to give them time to fix their systems before making the vulnerabilities public.

We found it challenging to find an appropriate method to contact Master Lock. We tried the following recommended methods to no avail:

- The Forum of Incident Response and Security Teams (FIRST) list[4]
- The Task Force on Computer Security Incident Response Teams (TF-CSIRT) Trusted Introducer list[5]
- Common Vulnerabilities and Exposures (CVE) CVE Numbering Authorities (CNA) list[6]
- A security.txt file[7]
- HackerOne's directory[8]
- Bugcrowd's bug bounty list[9]
- RFC 2142 email addresses [24]

The only official contact methods we found were contact forms on the main Master Lock website[10] and on the Master Lock Vault website[11]. As these were both customer contact methods, we were not hopeful of our disclosure document getting to the right place.

As a last resort, we managed to find the "Global Information Security Manager" for Master Lock through LinkedIn. We reached out to their personal email address to ask for a work contact, stating our intentions. They replied quickly (same day on 19 March 2018) and promised to analyse our disclosure document, but we did not hear anything further. While writing up this paper, we decided to try contacting Master Lock again. This happened on 12–13 March 2019 and this time we got a much more comprehensive response. They confirmed our findings, and worked quickly to resolve the issues. As a result, several key vulnerabilities have now been patched:

- The most serious vulnerability in the API (which would allow a guest user to obtain the primary code and future temporary codes) has been fixed, by restricting guest access permissions
- Insecure HTTP access to their API has been disabled
- The API now uses a rate-limiting mechanism to prevent an attacker from spamming email addresses

There are still some less severe vulnerabilities that need to be addressed, but we feel confident that the company is taking security seriously, and that they will fix the other vulnerabilities in due course.

VI. Conclusion and Future Work

Previous research into smart lock security had found many locks to be insecure, but did not show any technical vulnerabilities in Master Lock's offering. We applied techniques from one researcher [7] which broaden the scope of attack to the entire system, and ultimately found

[3]https://www.masterlock.com/bluetoothlockbox/guest-access

[4]https://first.org/members/teams/

[5]https://www.trusted-introducer.org/directory/teams.html

[6]https://cve.mitre.org/cve/request_id.html#cna_participants

[7]https://securitytxt.org/

[8]https://hackerone.com/directory

[9]https://www.bugcrowd.com/bug-bounty-list/

[10]https://www.masterlock.eu/contact-us

[11]https://contact.masterlock.com/90/contactus.aspx

several significant vulnerabilities with the Master Lock Vault eLocks service.

We disclosed these vulnerabilities to Master Lock (initially on 19 March 2018, with a follow up a year later). After some initial difficulties in communication, we managed to get a positive response from Master Lock, who acknowledged our findings and moved quickly to implement patches to secure the padlock's ecosystem.

Future investigation could look closer at the Bluetooth LE protocol used between the smartphone app and the padlock. Other researchers seem to have found it not trivially insecure, but a more in-depth analysis is required to determine if there is any part of the communication which is vulnerable. Capturing the padlock firmware during an update would also be interesting. This can be analysed in a similar way to the Android application to see if it contains any hard-coded keys or secrets, or even to reverse-engineer the algorithm for generating temporary codes.

A mechanical brute-forcer tool could be built to allow a systematic approach to unlock the Master Lock by repeatedly guessing the override codes. This is an interesting engineering challenge, and we believe the low entropy of potential codes will make it feasible to find the correct code in a relatively short time.

From a more generic point of view, many IoT systems rely on back-end services to handle data and provide necessary features. We have shown in this paper that these back-end services can be the cause of serious vulnerabilities if they are not secured properly. A holistic perspective has to be taken with IoT security, requiring thorough examination of the devices, supporting cloud services, and interlinking components.

Finally, our experience in conducting a responsible disclosure exercise shows that it is a valuable process, but we are still some way from having an effective means to communicate our findings to device manufacturers.

VII. References

[1] Louis Columbus. 2017 Roundup Of Internet Of Things Forecasts. https://www.forbes.com/sites/louiscolumbus/2017/12/10/2017-roundup-of-internet-of-things-forecasts/, December 2017.

[2] Eyal Ronen and Adi Shamir. Extended functionality attacks on iot devices: The case of smart lights. In *2016 IEEE European Symposium on Security and Privacy*, pages 3–12, 2016.

[3] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 636–654, 2016.

[4] Kishore Angrishi. Turning internet of things (iot) into internet of vulnerabilities (iov): Iot botnets. *arXiv:1702.03681*, 2017.

[5] Manos Antonakakis, Tim April, and Michael Bailey, et al. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, 2017.

[6] Anthony Rose and Ben Ramsey. Picking Bluetooth Low Energy Locks from a Quarter Mile Away. Presented at DEF CON 24, August 2016. Recording available at https://www.youtube.com/watch?v=KrOReHwjCKI.

[7] Jmaxxz. Backdooring the Frontdoor. Presented at DEF CON 24, August 2016. Recording available at https://www.youtube.com/watch?v=MMB1CkZi6t4.

[8] Hui Suo, Jiafu Wan, Caifeng Zou, and Jianqi Liu. Security in the internet of things: a review. In *2012 international conference on computer science and electronics engineering*, volume 3, pages 648–651. IEEE, 2012.

[9] Jacob Wurm, Khoa Hoang, Orlando Arias, Ahmad-Reza Sadeghi, and Yier Jin. Security analysis on consumer and industrial iot devices. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 519–524. IEEE, 2016.

[10] Jack McBride, Julio Hernandez-Castro, and Budi Arief. Earworms Make Bad Passwords: An Analysis of the Noke Smart Lock Manual Override. In *Int'l Workshop on Secure Internet of Things 2017 (SIoT 2017)*, Sep 2017.

[11] Glenn P. Meekma. Padlock. Patent US8453481B2, July 2010. Available at https://patents.google.com/patent/US8453481B2.

[12] Nathan Conrad, Yi Zhang, and Nemanja Stefanovic. Wireless key management for authentication. Patent US9600949B2, July 2014. Available at https://patents.google.com/patent/US9600949B2.

[13] Nathan Conrad. Wireless firmware updates. Patent WO2017066409A1, October 2015. Available at https://patents.google.com/patent/WO2017066409A1.

[14] Bluetooth Special Interest Group. *Bluetooth Core Specification*, core version 5.0 edition, December 2016.

[15] BosnianBill. (897) Review: Master Lock's Bluetooth Padlock. https://www.youtube.com/watch?v=YsKMsvx8vvo, August 2016.

[16] Dave Jones. EEVblog #1014 - Masterlock Bluetooth Padlock Teardown. https://www.youtube.com/watch?v=zRdovnGruqk, 2017.

[17] Texas Instruments. MSP430. http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview/overview.html.

[18] Gerald Combs. Wireshark. https://www.wireshark.org/, July 2016. version 1.12.13.

[19] Connor Tumbleson and Ryszard Wiśniewski. Apktool. https://ibotpeaches.github.io/Apktool/, September 2017. version 2.3.0.

[20] Bob Pan. dex2jar. https://github.com/pxb1988/dex2jar, June 2015. version 2.0.

[21] Emmanuel Dupuy. JD-GUI. http://jd.benow.ca/, August 2015. version 1.4.0.

[22] Zetetic LLC. SQLCipher. https://www.zetetic.net/sqlcipher/, November 2008.

[23] Square Inc. Retrofit. https://square.github.io/retrofit/, 2013.

[24] Dave Crocker. Mailbox names for common services, roles and functions. RFC 2142, Internet Mail Consortium, Santa Cruz, CA, May 1997. Available at https://www.ietf.org/rfc/rfc2142.txt.

Appendix

A. Samples of Generated Temporary Codes

Using the Python script outlined in Section IV-E3, we managed to create temporary access codes for one of our Master Locks, for every 4-hour slot from 7 December 2017 until 5 December 2027. A small selection of the generated temporary codes is provided below.

```
2017-12-07_00 RLLDLDRR      2019-03-15_00 RDRUUULL      2027-12-03_00 RUDLDDUL
2017-12-07_04 RUDULLRR      2019-03-15_04 RLDULLUD      2027-12-03_04 RRLLRDLL
2017-12-07_08 RRDDLLUL      2019-03-15_08 RRRDLRUL      2027-12-03_08 RRLDLRLR
2017-12-07_12 RRLLDRDL      2019-03-15_12 RURRDLDU      2027-12-03_12 RRRRLRDU
2017-12-07_16 RRRRDUDU      2019-03-15_16 RUDLDLDR      2027-12-03_16 RRULURUD
2017-12-07_20 RURULLLR      2019-03-15_20 RRDURLLR      2027-12-03_20 RRRURRRL
2017-12-08_00 RRDDLRUU      2019-03-16_00 RRLULRUR      2027-12-04_00 RLDULUDD
2017-12-08_04 RDDLDLLR      2019-03-16_04 RLLURDRU      2027-12-04_04 RURLDLLR
2017-12-08_08 RDRLUUDL      2019-03-16_08 RRLURRLD      2027-12-04_08 RRLRLDLR
2017-12-08_12 RURDURLD      2019-03-16_12 RULDUURL      2027-12-04_12 RRRLUULR
2017-12-08_16 RUDRRULL      2019-03-16_16 RULURLLU      2027-12-04_16 RDDDLDLD
2017-12-08_20 RRDLUDRD      2019-03-16_20 RRDUUDLU      2027-12-04_20 RDUUUULU
2017-12-09_00 RURLLDDR      2019-03-17_00 RDLDRRLL      2027-12-05_00 RULRUUUD
2017-12-09_04 RULRDDDU      2019-03-17_04 RDURUDRR      2027-12-05_04 RLDURDLD
2017-12-09_08 RRLDLRLR      2019-03-17_08 RRRDULLR      2027-12-05_08 RLULLDLL
2017-12-09_12 RLULRURR      2019-03-17_12 RDLUULUR      2027-12-05_12 RUDUDDRD
2017-12-09_16 RDLLDRLD      2019-03-17_16 RLUDDDUR      2027-12-05_16 RDLRDDLU
2017-12-09_20 RURDULDL      2019-03-17_20 RDDLULRL      2027-12-05_20 RRURDLDD
...                         ...
```