No Key, No Problem: Vulnerabilities in Master Lock Smart Locks

Chengsong Diao, UC San Diego
Danielle Dang*, UC San Diego
Sierra Lira*, UC San Diego
Angela Tsai*, UC San Diego
Miro Haller, UC San Diego
Nadia Heninger, UC San Diego

Abstract

Smart locks are an increasingly popular and critical component of smart homes due to their convenience and efficiency compared to traditional locks. In this paper, we conduct an in-depth analysis of one smart lock product, the Master Lock Deadbolt D1000. We reverse engineer the Master Lock Vault Enterprise Android app, analyze their proprietary communication protocols, and discover several vulnerabilities: (1) Replay attacks can allow unauthenticated unlocking; (2) Former guests can continue unlocking the lock after their access should have expired; (3) Malicious users can arbitrarily extend temporary access and lock other users out; (4) Attackers can forge audit events and prevent authentic events from being uploaded to the telemetry servers; (5) Malformed Bluetooth Low Energy (BLE) messages can result in a Denial of Service (DoS) as well as memory leaks and corruptions. We developed an Android app implementing the communication protocols in order to demonstrate proof-of-concept exploits of these attacks. Finally, we propose countermeasures and discuss their broader implications for the security of smart locks and similar IoT devices.

1 Introduction

A smart lock allows users to unlock doors using a keypad, mobile app, or electronic key fob. The global smart lock market size was estimated to be $2.8 billion in 2024 [16]. Smart locks are not just popular, they are also security critical because they are designed to be integrated into doors as a trusted component for access control. Thus, attacks on smart locks have real-world impact on the physical safety of humans.

Smart locks are increasingly common for numerous use cases including accessing hotel rooms and vacation rentals, and allowing temporary access for pet sitters, house cleaners, or maintenance and repair personnel. In a common scenario, a smart lock owner will grant other users access to unlock the lock for a limited time period. A study by Mare et al. [13] found that smart locks were the second most common "smart" device found in Airbnb rentals. Hazazi and Shehab conducted a semi-structured interview study with 29 participants and found that popular smart lock features are "the ability to remotely control the lock", "keyless entry," and "the ease of giving others access" [5].

A typical app-controlled smart lock involves at least three systems who must communicate with each other to authenticate users and operate the lock: the physical lock itself, a user's mobile device, and a remote server controlled by the manufacturer [20]. Communication between the smart lock device and the user's mobile device is straightforward, usually via either Wi-Fi or Bluetooth Low Energy (BLE). Communication between the smart lock and the manufacturer's remote server is more complex. Some smart locks may include a built-in Wi-Fi modem so that the device can communicate directly over the Internet with the remote server. Ho et al. [6] analyzed one such example from Lockitron (since aquired and integrated into the myQ platform[1]). Connecting the lock to Wi-Fi is relatively computationally expensive for a resource-constrained device, and requires additional hardware components and energy consumption. Moreover, there are security implications to exposing a smart lock device to the Internet, such as vulnerabilities enabling remote attacks. An alternative choice is to design the smart lock so that it communicates only over BLE and relies on the user's mobile device to communicate with the manufacturer's remote server. This architecture may be referred to as a device-gateway architecture [6] or mobile-as-a-gateway IoT [22] because the phone serves as an internet gateway between the lock and the manufacturer's remote server, while the smart lock only communicates with the mobile phone via Bluetooth. Smart lock brands that use this architecture include Level, August, Yale, and Kevo [6, 22].

In this paper, we do an in-depth analysis of a smart lock product from Master Lock, a popular US manufacturer of smart locks with a revenue of $860 million in 2022 [17] that also uses this device-gateway architecture. As shown in

*equal contribution

[1]https://www.myq.com/products/smart-lock

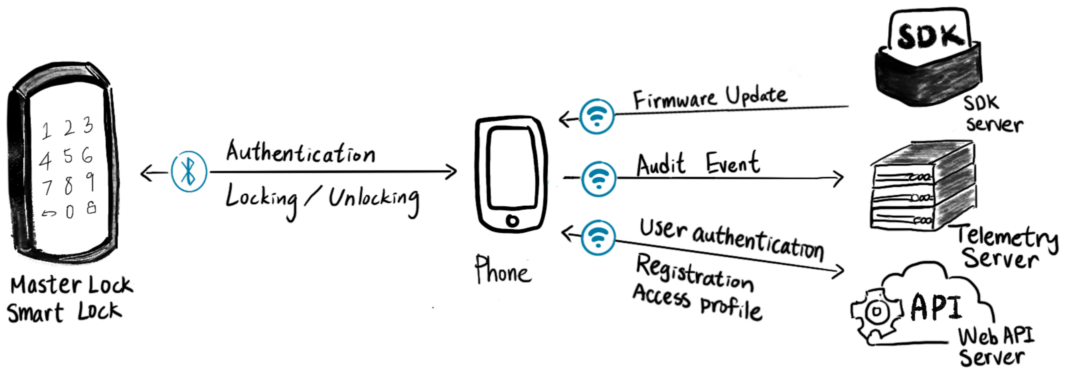

Figure 1: The Master Lock architecture includes the smart lock device, the user's mobile phone which acts as a gateway, and remote SDK, telemetry, and API servers operated by Master Lock.

Figure 1, they use three different servers to (1) fetch firmware updates from the SDK server, (2) upload audit events (such as when the lock was opened/closed) to the telemetry server, and (3) a general API server for user registration, authentication, and the issuance of access profiles. Using these access profiles, a user can later authenticate to the smart lock and perform privileged operations such as locking and unlocking.

Several prior works published between 2016 and 2019 [1,6,7,9,21] on the security of smart locks found widespread vulnerabilities across many vendors, including August, Dana, and Kevo. These results showed that security has been neglected in the early design of smart locks: four products sent passwords in plaintext over BLE, five were vulnerable to replay attacks, and five allowed guests with revoked access to unlock the door by setting their phone to airplane mode. Compared to their competitors, Master Lock fared better. In the analysis of Rose and Ramsey [1]—which analyzed 16 smart locks—Master Lock was one of the four locks for which they found no vulnerabilities. Later, Knight, Lord, and Arief [9] identified insufficiently secured API endpoints for Master Lock that allowed users to bypass access hour restrictions and open the lock after revocation. In their analysis, they found that the Bluetooth communication between the lock and phone was encrypted, and were not able to analyze it further. We give more details on these and other related works in Section 10.

Our Contributions. In the past six years, Master Lock appears to have patched the vulnerabilities reported in prior work and secured their API endpoints. Nevertheless, we carry out a more in-depth analysis of the mobile phone app, the lock's use of encryption, and the custom Bluetooth communication protocol, and demonstrate several novel vulnerabilities exploiting design flaws.

We give three attacks that allow an adversary in physical proximity of the lock to open it, forge entries in the audit log, and prevent the lock owner from operating the lock. Two other attacks allow a temporary guest to open the lock even after their access expired or was revoked. We describe our attacks and place them in the context of a threat model in Section 2, and give full technical details Section 6. We validated our findings by executing proof-of-concept attacks described in Section 7 with a custom Android app implementing the BLE protocols to communicate with the lock. To carry out our analysis, we had to overcome custom obfuscation methods and other obstacles added after the prior work described above using techniques described further in Section 4. The resulting reconstructed protocols are in Section 5. We disclosed our attacks to Master Lock in March 2025, and Master Lock plans to publish mitigations in June 2025. We describe these mitigations and tradeoffs in detail in Section 8.

Our work illustrates the evolving security landscape for smart lock security. We demonstrate attacks against smart locks that are just as strong as previous attacks demonstrated in 2016, but require increased sophistication on the part of the attacker. Rather than generic attack techniques that apply across many smart locks, a deeper analysis of the protocol is necessary to find more subtle flaws in the design.

2 Threat Models and Attack Overview

In this paper, we analyze the Master Lock Bluetooth Deadbolt D1000. We reverse-engineered the Master Lock Vault Enterprise Android app in order to study the lock-to-phone Bluetooth communication protocol and phone-to-server communication protocols and their security properties.

Before outlining our attacks, we describe our threat model as a combination of adversary capabilities and security goals.

2.1 Threat Model

All of our adversaries have the expertise to reverse engineer the mobile app, e.g., to extract static secrets that are necessary to authenticate and use the Master Lock telemetry server. Additionally, we consider the following adversary capabilities:

A1 physical proximity to the lock with the ability to eavesdrop on the (encrypted) communication over BLE, and ability to inject arbitrary BLE packets sent to the lock,

A2 temporary legitimate user-level access to the lock (e.g., as a hotel guest) in the past, which was either actively revoked by the lock administrators or expired naturally.

These adversaries aim to compromise the following security goals of the smart lock:

G1 only authenticated users can perform privileged operations including locking and unlocking the smart lock,

G2 access control is enforced as soon as the next legitimate user opened the lock, including preventing access after revocation or expiration,

G3 audit log integrity,

G4 memory safety of the smart lock,

G5 availability of the smart lock to authenticated users.

We extracted these security goals from a combination of advertised features (**G1–3**) and common expectations on IoT devices (**G4–5**). For instance, Master Lock states that "manual access codes that user may have seen will remain viable until an administrator changes the codes and has someone perform a Bluetooth unlock to overwrite the old codes on the lock" [15] (cf. **G2**). Furthermore, **G3** is informed by their statement that Master Lock "records every time a lock is unlocked/locked, activated/deactivated, or its temporary code is viewed", which implies the audit log is complete and correct.

We consider threat models as the combination of adversary capabilities and violated security goals. E.g., (**A2**, **G2**) considers an adversary with temporary user access to the lock who preserves their access even after it was revoked or expired.

We discovered the following attacks against the Master Lock Bluetooth Deadbolt D1000 smart lock by reverse engineering the companion Android app.

- **Attack 1 (session replay):** Adversary **A1** can record the BLE communication of a whole session and replay it to repeat all executed commands, including unlocking the lock (which violates **G1**), without decrypting the communication or possessing a valid access profile. (Section 6.1.)
- **Attack 2 (exceeding access):** Adversary **A2** with temporary access can retain the access profile and continue operating the lock until the original expiration, even if the administrators actively revoke the access before expiration, violating **G2**. (Section 6.2.)
- **Attack 3 (clock tampering):** With temporary access, adversary **A2** can adjust the clock time of the smart lock arbitrarily, extending their own access past expiration (violating **G2**) or locking out all legitimate users (violating **G5**). (Section 6.3.)
- **Attack 4 (audit log tampering):** Adversary **A1** without access to the lock can upload arbitrary audit events to the telemetry server, and prevent legitimate audit events from being uploaded, compromising **G3**. (Section 6.4.)
- **Attack 5 (malformed messages):** Without valid access, adversary **A1** can send malformed BLE messages to the lock, making it unresponsive to BLE communication and the touchpad (violating **G5**) or corrupting memory content (violating **G4**). With valid access, adversary **A2** can even perform unauthorized reads to smart lock memory, further violating **G4**. (Section 6.5.)

3 Background

3.1 Notation

We use the following conventions. Encryption of message m with AES in CTR mode with key k and nonce n is denoted by AES-CTR.Enc($\mathsf{k},\mathsf{n},\mathsf{m}$), and decryption of ciphertext c as AES-CTR.Dec($\mathsf{k},\mathsf{n},\mathsf{c}$). Similarly, AES-CBC.Enc and AES-CBC.Dec denote AES-CBC encryption/decryption. Concatentation of byte strings is denoted by $\|$ (e.g., $a \| b$ for byte strings a and b). Furthermore, we denote bytes in hexadecimal notation, omitting concatenation where it is clear from context, e.g., `25CAFE` for bytes 25 (integer 37), *CA* (202), and *FE* (254). The length in number of bytes of a byte string a is denoted by $|a|$. We use $|a|_2$ to denote the big-endian encoding of a two-byte value with length less than 2^{16}.

3.2 Master Lock Bluetooth Products

The Master Lock company sells both traditional locks as well as a line of Bluetooth and Electronic Locks that are claimed to "withstand physical attacks and hacks, protecting your property with military-grade encryption and high-security features." [14] These products include Bluetooth padlocks, lock boxes, and door hardware with a door controller and deadbolt. Users can use the Master Lock Vault Enterprise or Home edition mobile apps [14] to interact with the locks. The product marketing emphasizes usability features such as replacing traditional keys with user mobile phones, security claims including that "encrypted digital keys can't be duplicated, so you control who has access", and the ability of users to manage and monitor locks remotely, with access to detailed history logs that track when, where, and by whom the lock was accessed. The Enterprise app supports user management, which allows designated administrators to add and remove regular users and modify their access to smart locks.

In this paper, we analyze the Master Lock D1000 Bluetooth deadbolt lock. Previous work identified security vulnerabilities in Master Lock Bluetooth padlocks [9], but we are unaware of prior or more recent analysis of this deadbolt product, which would be more likely to be used to secure residential or commercial property.

3.3 Bluetooth Low Energy Protocols

The communication between the mobile phones and lock that we analyze is based on Bluetooth Low Energy (BLE). To reduce battery usage and extend battery life, the lock is inactive most of the time and does not send any advertising packets. This means that phones cannot discover locks by scanning. Locks start advertising only when they are woken up by tapping their buttons or touchpad. At that point, a phone can discover the lock, establish a connection, and communicate. This communication is defined by the generic attribute profile (GATT) and includes two important concepts: services and characteristics. A service is identified by a service universally unique identifier (UUID) and consists of a logical group of characteristics. Every characteristic (identified by a characteristic UUID) represents a data point and can be written or read. Communication happens over a client-server architecture. The locks are GATT servers, which store data in the form of characteristics. Phones are GATT clients that read and write characteristics to communicate with locks.

4 Reverse Engineering

To analyze the security of the protocols designed by Master Lock, we reverse engineered their Master Lock Vault Enterprise Android app, version `2.28.0.1` published on April 10, 2024. A second consumer-focused app, called Master Lock Vault Home, uses the same BLE protocols for phone-to-lock communication. The only difference is in the server API endpoints. The enterprise app is the focus of our analysis because it provides more functionality.

Android APKs contain Dalvik EXecutable (DEX) files, which store executable code for applications and are compiled from Java or Kotlin code. Multiple tools can disassemble APKs and generate intermediate code for DEX files in different formats, such as Smali, Jasmin, and Jimple [19]. From these intermediate formats, decompilers can generate higher-level Java code. Normally, if an APK is released as-is after compilation, the decompiled code is fairly readable. Apart from some missing local variable names and occasionally misleading compiler-generated (synthesized) code, reverse engineering such an APK is not much more challenging than a normal code review; original class names and class or instance variable names, as well as most code blocks, control flows, and constants are reconstructed automatically. However, the Master Lock uses several techniques to obfuscate and protect its code in the hope of deterring reverse engineering.

Although obfuscation and protection make it significantly more time consuming to analyze program behaviors, they do not protect the apps against motivated adversaries. This section discusses a selection of the custom obfuscation techniques used in the Master Lock Vault Enterprise app, their goal, and how we bypassed them. We focus on a static analysis of the APK and only selectively execute parts of the code dynamically. This has lower setup overhead than a fully dynamic analysis such as emulating the application or attaching a debugger. It is also more generic since, e.g., debugging does not help with code that is decrypted and executed at runtime.

4.1 Renaming

Technique. Packages, classes, annotations, class variables, and instance variables are renamed to random sequences of characters. A variant of this technique misleadingly renames objects to share the same name as common library functions.

Objective. Obfuscation by renaming strips compiled code of the semantic information added by descriptive names. The resulting code is more time-consuming to understand, as all this information needs to be reconstructed from context and by inspecting all occurrences of the object in the program. This makes it more challenging to identify the subset of code relevant to our analysis. Intentionally confusing variable names can lead to incorrect guesses for the semantics of variables, classes, and methods, which diverts the focus of analysis.

Bypass. During reverse engineering, we replace random names with more meaningful ones, for example, with a guess at capturing the semantics of that object. In some cases, it can already help to make names unique across different scopes. For instance, the same random names might be used for multiple classes in different packages, or variables across classes. Even if we cannot guess the semantics, assigning them generic unique names (e.g., `integer_1` or `class_A`) can make searching for references more efficient.

4.2 Encrypting Constants

Technique. Security-critical constant integers, floating point numbers, and strings are encoded and encrypted, and only decoded and decrypted at runtime. The decoding code and locally stored decryption keys are heavily obfuscated (e.g., by other techniques in this section) to hinder static analysis.

Objective. The encryption of primitives and strings makes it difficult to find relevant code by searching key words. It also creates challenges in the analysis of algorithms and protocols, such as the binary formats of commands and messages.

Bypass. The code to decrypt and decode constants is often run in the static initialization of classes to minimize the runtime overhead added by this obfuscation. Hence, we can read the deobfuscated constants by using dynamic partial evaluation to load the classes and read the decrypted constants. However, this requires direct access to the classes. Although decompilers may not be able to reconstruct the Java source code of obfuscated classes, tools including dex2jar[2] can often transform Smali to Java bytecode, where only few classes require manual bytecode corrections. Given the bytecode as JAR file, we can load it as a Java library and write Java code to access fields of classes that contain the decrypted and decoded constants after the class was initialized. Unfortunately, for the Master Lock Vault Enterprise app, the static initialization code for decryption contains calls to the Android platform libraries which prevents us from loading the JAR. We discuss this obfuscation technique next.

4.3 Calling Native Functions

Technique. Some constant primitives, especially integers, are replaced with calls to native functions in the Android platform libraries. A later fix-up calculation ensures that all constants are identical in the end to the pre-obfuscation values so that the code behavior is unchanged.

Objective. The additional function calls increase code complexity and obscure the original simple constant values, which impedes static analysis. Moreover, they create challenges when executing code on non-Android platforms, as discussed in Section 4.2.

Bypass. We replace native platform library functions with local ones that return placeholder values. The return values can be an arbitrary value from the range defined in the Android documentation or by common sense. As the obfuscation needs to guarantee deterministic behavior on all Android platforms, returning any reasonable value must result in the deobfuscated constant after the fix-up calculation. A possible alternative is to replace each function call (rather then the called function) with a placeholder value, but that is less efficient and prone to errors especially when dealing with obfuscated code.

4.4 Using Reflection

Technique. Security-critical function call targets are obscured by Reflection, where the call target—stored as a string and passed as an argument to call the function—is obfuscated by techniques listed above. Reflection is mainly used for calling or accessing dynamic targets, because the targets are identified by string arguments to Reflection functions.

[2]https://github.com/pxb1988/dex2jar

Objective. Without explicit call targets as in normal function calls, it is challenging to find references to functions and trace function calls. Without this information, we do not have the context in which a function is used and cannot access interactions between different parts of the code.

Bypass. For deobfuscation, we search the invocation targets of relevant Reflections for function calls or field accesses. These must be present as strings. If the strings are encrypted, we can decrypt them with dynamic partial evaluation (see Section 4.2). Alternatively, if the Smali code is too complicated, we create breakpoints inside the Reflection related library functions[3] and print the argument values upon breakpoint hits. Next, we can read the invocation targets from standard output.

4.5 Loading DEX Dynamically

Technique. Critical parts of the code are stored in separate DEX files as encrypted assets inside APKs, which are decrypted and loaded dynamically during run-time.

Objective. The classes stored in encrypted assets are not available if only the DEX files are disassembled. Additional steps are required to extract code from assets for static analysis and to load the contained classes for relevant dynamic evaluation to work correctly.

Bypass. There is no way to directly load DEX files as Java classes as they contain Dalvik bytecode, which is different from Java bytecode and cannot be interpreted by JVM. To get around this, we pull the source code related to loading DEX files, mainly from the Android library `dalvik.system` package, and make some modifications. To start, we write DEX files to disk when they are loaded. Then, we convert the DEX files to JAR files and load them as runtime libraries in JVM. This modified DEX class loader will always try to load requested classes from runtime libraries instead of reading the real DEX files. The app code will perceive no difference from a real DEX loader after all DEX files are converted and properly loaded in the JVM runtime.

4.6 Obfuscating Control Flows

Technique. Control flow obfuscation aims to hide execution paths by adding decoy paths through redundant conditions, branches, or exceptions. The additional branches contain similar but incorrect code compared to the original. Some bytecode may be rearranged to confuse decompilers and make them unable to reason about code logic and prevent them from restructuring code to more common control flow patterns.

[3]Good candidate functions for breakpoints are `Class.forName`, `Class.getMethod`, and `Class.getField`.

Objective. Excessive branches and exception handlers make decompilers fail and manual reading of scrambled Smali instructions very challenging. If there are slightly different branches depending on some non-trivial conditions, it is hard to statically determine the correct branch, impeding analysis.

Bypass. It is challenging to provide deobfuscation that works well for all types of obfuscation and does not accidentally change program behaviors, especially when the obfuscation is aggressive. For example, there are thousands of lines of bytecode for loading the core cryptographic functions for the BLE communication from external encrypted DEX files. They are strongly obfuscated, which makes decompilers fail and manual inspection of bytecode too time consuming. Hence, in our reverse engineering process, we use breakpoints inside the JVM library functions to infer functions of some code by inspecting the arguments, return values, and call stacks. Although this method may not be accurate as we miss detailed code logic inside obfuscated functions, the correctness can be verified by checking output. For example, we can compare the results from reverse engineered source code to the ones of the original bytecode. Moreover, we confirm our understanding of the code by testing our attacks against the real product.

Tradeoffs. We can try additional techniques to deobfuscate control flow, such as removing all exception throwing and catching, skipping the code restructuring process of the decompiler to tolerate abnormal control flows (although this will make the code less understandable), and inserting fake line number labels among bytecode instructions to allow breakpoints and inspection of variable values in debuggers. However, there is a tradeoff between coverage and correctness: more aggressive deobfuscation might produce more structured bytecode, but it might also introduce more errors, which are time-consuming to catch. Thus, we avoided overly aggressive control flow deobfuscation when possible, especially for complicated functions.

5 Communication Protocols

This section describes the Master Lock app and smart lock communication protocols and cryptography. We extracted them using the techniques discussed in Section 4. These communication protocols form the basis of our security analysis in Section 6.

Recall from Figure 1 that the phone communicates with the API servers over the Internet (discussed in Section 5.1) while the smart lock is not connected to Wi-Fi and only communicates with the phone over BLE (discussed in Section 5.2), possibly having the phone relay messages to the API server.

5.1 Phone-to-API Server Communication

Master Lock BLE devices use three different API servers: an SDK server for firmware operations, a telemetry server to collect audit logs from locks, and an enterprise API server for user and access management. There are numerous API endpoints defined in the APK file, but we focus on a small security-critical subset. Among these, the factory reset, firmware update, and latest audit event index API endpoints are on the SDK server; the audit trail events upload API endpoints are on the telemetry server; and all other API endpoints are on the enterprise API server. All communications use HTTPS with certificate pinning to prevent machine-in-the-middle (MITM) attacks.

Authentication on the SDK and telemetry servers. The SDK server and the telemetry server do not require any user-specific credentials. Each only requires two strings, a license and a password, which are embedded in the app but are encrypted with 2048-bit RSA. The RSA private key to decrypt these two strings is encoded in the app and is XOR-encrypted with a custom-generated pad. Different pairs of licenses and passwords are used for the SDK server and the two telemetry servers (located in North America and Europe). After successful authentication with the license and password, the server returns a bearer token that is valid for one hour.

Factory reset and firmware update. The app can initiate factory reset or firmware update requests by sending the device identifier, firmware version of the lock, and a valid bearer token to the SDK server. The server then returns a sequence of encrypted commands, which will be encapsulated in the `FirmwareUpdate` command (as the $\mathsf{cmd}_{\mathsf{firmware}}$ part) and be relayed to the locks one by one.

Fetching latest audit event index. Before the app tries to upload audit events recorded on the lock, it queries the latest audit event index, providing the lock's device identifier and a valid bearer token to ensure only new events are uploaded. After getting this index, the app can increment it and use it as the start index for the `ReadAuditTrail` command sent to the lock. Combined with the end index returned by the lock in the `ReadAuditTrailEventIndex` commands, the app can get the new audit event entries since the last upload. This process may be repeated multiple times until all new events are uploaded[4] because the number of new events can be too large to fit in one BLE message.

Audit event uploads. There are three types of audit events that the app uploads to the telemetry server: encounter events, device events, and virtual events. Encounter events happen

[4]When there are no new audit events, the lock returns the error code 4.

when the app connects to a lock, which contains the lock's battery level, device identifier, encounter time, and location (latitude and longitude provided by the phone). Device events are specific to the lock. These include firmware updates, system clock time changes, invalid access attempts, and unlock events. Each event contains an event type, event data and length, event index (a counter value in the lock), and a firmware counter. The third type of event, virtual events, are only defined and not used in the app. Their definition contains the event type, event data, and time, where the data types are the same as in device events, but event types are likely different.

The telemetry server has three separate API endpoints corresponding to these three types of events. Uploading audit events does not require user, phone, or smart lock authentication. It only requires a valid bearer token and the device identifier of the smart lock.

Authentication on the enterprise API server. The enterprise API server requires user authentication to access device information including profiles and keys. A user needs to provide their organization ID, email, and password. The app sends these along with a license and a password, which are encoded and encrypted in the app but different from the pairs for the SDK and telemetry servers. For the enterprise API servers in North America and Europe, the same pair of license and password are used. If the authentication succeeds, the app receives a bearer token that is valid for 1.5 hours and other user information and settings.

Device information. The app can query information for all locks for which the current user has access with the bearer token, including device identifiers, product models, firmware versions, available updates, last known locations, and battery levels. The data includes device settings like passcode nicknames, but not passcodes or session keys for BLE communication with locks.

Session keys and access profiles. After receiving a device identifier, the app can use the bearer token to request an access profile for the lock. The access profile has the following format, concatenating the base64 encoded values of the session key k, profile P, and the index idx of the last uploaded audit trail event:[5]

$$\text{access profile} = \text{base64}(\mathsf{k}) \,\|\, \text{base64}(\mathsf{P}) \,\|\, \text{base64}(\mathsf{idx})$$

This information is used to establish a secure channel between the phone and smart lock. As we describe further in Section 5.2, the phone sends the profile P with the `StartSession` command to the lock. This profile includes an encryption of the session key k. After this exchange, the phone and lock can encrypt data with the shared key k. The API server response also includes the validity start and end time for the profile.

[5]After base64 decoding, the session key is 32 bytes long (which corresponds to the 256-bit AES key), the length of profile P is 70 bytes, and the index is four bytes using little-endian byte order.

Binary structures. There is no information about the byte format of the firmware update command $\mathsf{cmd}_{\mathsf{firmware}}$ and profile P because they are not interpreted by the app. Instead, the phone directly relays these bytes to the lock by encapsulating them in the corresponding commands. Appendix A.2 provides additional information that we have inferred about their structure, which is not required knowledge for our attacks.

5.2 Phone-to-Lock Communication

The interaction between phone and smart lock consists of three stages: scan, connect, and communicate.

Scan. The smart lock sends BLE advertisements after it is woken up via the touchpad. The phone scans for BLE devices and the app parses scan records to get advertisement data, which contain advertisement types and corresponding advertisement records. To find a lock, the app checks that the 128-bit UUID advertisement record exists and matches one of two constant UUIDs: $\mathtt{UUID}_{\mathtt{device}}$ for a normal device state and $\mathtt{UUID}_{\mathtt{boot}}$ for a bootloader state. For completeness, these UUIDs are listed in Table 3, Appendix A.1. Then, the app parses the manufacturer-specific data in the advertisement record, which contains the company ID, SKU (stock keeping unit, a product identifier), device ID, and firmware version. If the device ID matches a list of known devices to which the user has access, the app proceeds to the connection stage.

Connection. The phone connects to the GATT server hosted by the lock. Once connected, the app starts service discovery by matching against one of the same service UUIDs as in the scan stage. Then, the app gets the characteristic with UUID $\mathtt{UUID}_{\mathtt{conn\text{-}char}}$ and the corresponding descriptor with UUID $\mathtt{UUID}_{\mathtt{conn\text{-}desc}}$ that contains additional configurations for the characteristic. The app sets the write type of the characteristic to default and enables notifications for the characteristic, so that it can receive responses from the lock. The app as the BLE client can write to the characteristic to send messages to the lock, and receive responses through notifications of BLE server initiated updates from the lock.

Communication. The app authenticates to the lock with the `StartSession` command, which also exchanges a session key k as part of the profile P that is sent to the lock. For invalid or expired profiles, the smart lock refuses to start a session which prevents the user from executing commands such as unlocking the door. All following commands are sent over an authenticated and encrypted channel, since the phone encrypts them with the session key before transmission. The lock responses are all encrypted in the same way, except for

Algorithm 1 CmdTag$(\mathsf{k},\mathsf{n},\mathsf{cmd})$

```
Input: Key k, nonce n, command cmd; $|\mathsf{k}| = 32, |\mathsf{n}| = 13$
    $\mathsf{cmd}_{\mathsf{encoded}} \leftarrow 19 \,\|\, \mathsf{n} \,\|\, |\mathsf{cmd}|_2 \,\|\, \mathsf{cmd} \,\|\, 00\ldots0$
                                    $\triangleright$ s.t. $|\mathsf{cmd}_{\mathsf{encoded}}| \bmod 16 = 0$
    $\mathsf{x} \,\|\, \mathsf{t} \,\|\, \mathsf{y} \leftarrow$ AES-CBC.Enc$(\mathsf{k}, 00\ldots0, \mathsf{cmd}_{\mathsf{encoded}})$
                                    $\triangleright$ s.t. $|\mathsf{t}| = |\mathsf{y}| = 8$
Output: Tag t
```

Algorithm 2 CmdEnc$(\mathsf{k},\mathsf{n},\mathsf{cmd})$

```
Input: Key k, nonce n, command cmd; $|\mathsf{k}| = 32, |\mathsf{n}| = 13$
    $\mathsf{c}_{\mathsf{cmd}} \leftarrow$ AES-CTR.Enc$(\mathsf{k}, 01 \,\|\, \mathsf{n} \,\|\, 0001, \mathsf{cmd})$
    $\mathsf{t} \leftarrow$ CmdTag$(\mathsf{k},\mathsf{n},\mathsf{cmd})$
    $\mathsf{C}_{\mathsf{t}} \leftarrow$ AES-CTR.Enc$(\mathsf{k}, 01 \,\|\, \mathsf{n} \,\|\, 0000, \mathsf{t})$
    $\mathsf{C} \leftarrow |\mathsf{cmd}|_2 \,\|\, \mathsf{c}_{\mathsf{cmd}} \,\|\, \mathsf{C}_{\mathsf{t}}$
Output: Ciphertext C
```

Algorithm 3 CmdDec$(\mathsf{k},\mathsf{n},\mathsf{C})$

```
Input: Key k, nonce n, ciphertext C; $|\mathsf{k}| = 32, |\mathsf{n}| = 13$
    $|\mathsf{cmd}|_2 \,\|\, \mathsf{c}_{\mathsf{cmd}} \,\|\, \mathsf{C}_{\mathsf{t}} \leftarrow \mathsf{C}$    $\triangleright$ s.t. $|\mathsf{c}_{\mathsf{cmd}}| = |\mathsf{cmd}|_2, |\mathsf{C}_{\mathsf{t}}| = 8$
    $\mathsf{cmd} \leftarrow$ AES-CTR.Dec$(\mathsf{k}, 01 \,\|\, \mathsf{n} \,\|\, 0001, \mathsf{c}_{\mathsf{cmd}})$
    $\mathsf{t}' \leftarrow$ CmdTag$(\mathsf{k},\mathsf{n},\mathsf{cmd})$
    $\mathsf{t} \leftarrow$ AES-CTR.Dec$(\mathsf{k}, 01 \,\|\, \mathsf{n} \,\|\, 0000, \mathsf{C}_{\mathsf{t}})$
    If $\mathsf{t}' \neq \mathsf{t}$ then $\mathsf{cmd} \leftarrow \perp$
Output: Command cmd or $\perp$
```

the `StartSession` command, which includes an initialization vector (IV) before the ciphertext. Next, we describe the encryption and decryption schemes, and the format of commands and responses.

Encryption and Decryption. The core encryption and decryption functions are more protected than other code in the app. They are inside an external encrypted DEX file, which is decrypted and loaded dynamically by heavily obfuscated loader code with thousands of Smali instructions. The scheme consists of an authenticated block cipher built from AES in CCM mode with a custom embedding function. CCM is a mode of operation standardized in NIST SP 800-38C [2] for block ciphers that combines a CBC-MAC authentication tag with a counter mode stream cipher. The encoding function for commands supports bounded-length commands of less than 2^{16} bytes. They use a 13-byte nonce[6] to build the IV for AES-CTR encryption. Due to careful IV construction, they avoid reusing IVs to encrypt the tag as well as command encryption, and the size of the counter matches the maximum command length. Algorithms 1 to 3 detail the customized encryption, decryption, and authentication tag algorithms, respectively.

Establishing a communication session. If the profile P sent in `StartSession` is valid, then the lock responds with a shortened nonce n_s and the encryption of an error code that should always be zero. The app uses the latter to run the decryption function and to confirm that the same session key k is used on both sides. For command encryption with session key k, the 13-byte nonce n is constructed by prepending seven zero bytes to the six-byte nonce n_s. After this, the nonce increments for every encryption (when the app sends commands to the lock) and decryption (when the app receives responses from the lock). The increments happen on the six-byte n_s, and wraps around after 2^{48} increments.

[6]For communication sessions, this nonce is $\mathsf{n} = 00\ldots00 \,\|\, \mathsf{n}_s$ for a six byte random value n_s, i.e., the nonce only has six bytes of entropy.

Table 1: Command response error codes for BLE messages.

Error Code e	Inferred Meaning
0	`Ok` (no errors)
1	`Invalid Operation`
2	`Invalid Time`
3	`Not Permitted`
4	`Data Not Available`

Command and Response Encryption. For all commands sent inside a session (i.e., after `StartSession`) except those reading data from the lock, the response from the lock is encrypted using the format $00 \,\|\,$ CmdEnc$(\mathsf{k},\mathsf{n},e)$, where e is a one-byte error code described in Table 1. Similarly, all commands (except `StartSession`) sent from the phone to the lock are encrypted with the format $01 \,\|\,$ CmdEnc$(\mathsf{k},\mathsf{n},\mathsf{cmd})$, where cmd is the byte-encoded command. A list of the critical commands, their binary formats, and the corresponding response formats from locks (if different from a single error code) is shown in Table 2.

6 Attacks

This section presents five attacks on the Master Lock Bluetooth Deadbolt D1000 smart lock, discussing the exploited vulnerability and impact for each. We refer to the specific attacker capabilities and security goals outlined in Section 2.

6.1 Attack 1: Session Replay

Vulnerability. We find that sessions for the same user use the same access profile (unless it expires) and hence the same session key. For every session, the smart lock picks the same nonce (`000000000001`, the six-byte big-endian integer one) in its response to the `StartSession` command. This allows a passive physically proximate adversary who merely records Bluetooth communications to replay all packets sent by the smart phone of the legitimate user at a later time. As long as the adversary preserves the order and was able to record all packets, the lock will execute the same commands again. The recorded session and the corresponding BLE connection

Table 2: Relevant commands in the Master Lock phone-to-smartlock communication via BLE.

Command Type	**Command Format**	**Response Format**
`StartSession`	$00 \parallel \mathsf{P}$	$00 \parallel \mathsf{n}_s \parallel \mathrm{CmdEnc}(\mathsf{k}, \mathsf{n}, e)$ OR 01
`KeepAlive`	$\mathsf{cmd} = 01$	default
`WriteTime`	$\mathsf{cmd} = \mathtt{0A} \parallel \mathsf{time}$	default
`ReadAuditTrail`	$\mathsf{cmd} = \mathtt{0CF001} \parallel \mathsf{idx}$	$00 \parallel \mathrm{CmdEnc}(\mathsf{k}, \mathsf{n}, e \parallel \mathsf{idx} \parallel \mathsf{event}_1 \parallel \mathsf{event}_2 \parallel \ldots)$
`ReadAuditTrailEventIndex`	$\mathsf{cmd} = \mathtt{0D}$	$00 \parallel \mathrm{CmdEnc}(\mathsf{k}, \mathsf{n}, e \parallel \mathsf{idx})$
`FirmwareUpdate`	$\mathsf{cmd} = 14 \parallel \mathsf{cmd}_{\mathsf{firmware}}$	default

are likely to have already been terminated before the adversary replays the messages, but this does not affect the attack because it starts a new session with the same key and nonce.

Impact. This attack allows a physically proximate adversary **A1** recording a benign unlocking event to violate goal **G1** and later replay the recorded session to gain unauthorized access to the property protected by the smart lock, such as a house or hotel room.

6.2 Attack 2: Exceeding Access

Vulnerability. We noticed that every access profile is valid for nine months when inspecting the metadata sent by the API server when the profile is requested, which includes the expiration date. Revocations only happen at the application layer, which prevents guests with revoked access from using the official Master Lock Vault Enterprise app to open the smart lock. However, on the underlying protocol layer, the access profile remains valid until its original expiration date.

Impact. An adversary **A2** with temporary access to the lock and the technical expertise to reverse engineer the protocol as we did in this work can extract their own access profile while they have legitimate access. After their access is revoked, they can use the access profile to authenticate to the lock and send any commands until the profile expires. Hence, in practice, it takes up to nine months for a revocation to take effect, violating **G2**.

6.3 Attack 3: Clock Tampering

Vulnerability. Although commands sent to the smart lock are authenticated and encrypted, this does not prevent a malicious user with temporary access from tampering with the lock state. We find that the `WriteTime` command, which is intended to synchronize the clock of the smart lock with the phone time, can be misused. As we learned during disclosure (see Section 8), this command should only be allowed for admins.[7] However, an adversary without admin privileges can bypass this restriction by implementing their own Bluetooth GATT client with the protocols required to send the `WriteTime` command directly to the lock and set the clock in the lock to an arbitrary time of their choosing.

[7]Admins can trivially tamper with the clock of the smart lock by adjusting the system time on their phone.

Impact. An adversary **A2** with temporary access can set the smart lock clock to a date in the past to maintain access beyond the expiration time of their access profile, violating **G2**. Alternatively, they can set the clock time to a future time, which expires valid access profiles. As a result, authentication requests with the `StartSession` command fail, resulting in a denial-of-service attack that locks out valid users, violating the goal of availability **G5**.

6.4 Attack 4: Audit Log Tampering

Vulnerability. The API endpoints for the telemetry server are only authenticated with a static license and password, which are both obfuscated and embedded in the app. Using the reverse engineering technique described in Section 4.2, an adversary can recover these values from the APK and authenticate to the telemetry server. Once authenticated, it can upload audit event data, which is neither encrypted nor authenticated. We inferred the binary format for audit events by observing the format of existing ones using the `ReadAuditTrail` command. Forging events for a specific smart lock requires knowledge of that device's identifier, which is conveniently advertised over BLE and can be recorded by any adversary in physical proximity to the lock. Furthermore, audit events are associated with a monotonically increasing index. If an event for a given index already exists, then the server does not accept any new entries. On the one hand, this means that the adversary can only forge entries after the last benign event that was uploaded. On the other hand, benign events that are uploaded after forged ones are also rejected. Adversaries can get the current event index from the SDK API server, which features the same flawed authentication as the telemetry server.

Impact. This attack allows an adversary **A1** that is in the physical vicinity a single time to sniff the device identifier and forge audit events for security sensitive operations, such as locking and unlocking, changing the clock time, and invalid access attempts. Moreover, the adversary can also hide real audit events by preemptively uploading seemingly benign audit events to increase the event index and prevent the lock from reporting a suspicious pattern of locking and unlocking events, clearly violating log integrity **G3**.

6.5 Attack 5: Malformed Messages

Vulnerability. We discovered that the adversary can send messages with mismatched encoded length and actual length and cause the smart lock to crash. Specifically, when the encoded length is between 610 and 6561, the lock becomes unresponsive and cannot be activated through the touchscreen or BLE. These malformed messages affect both the unauthenticated `StartSession` command as well as later commands that are authenticated and encrypted. Furthermore, large encoded lengths that do not crash the lock (smaller than 610) overwrite other memory. In particular, this allows an adversary to overwrite the audit event index, which causes the lock to return arbitrary memory as event data in responses to `ReadAuditTrail` commands. After the audit events are uploaded to the telemetry server, the memory data can be exposed to the user/adversary as part of the audit log if the memory bytes can be parsed as valid events. If the memory cannot be parsed as valid events, the lock will label them as "unknown" type events, which are not shown in the official app and the web portal. However, these invalid events can still occupy the indices for authentic valid events, causing these valid events to be dropped.

Impact. Sending malformed `StartSession` commands allows any adversary **A1** with physical proximity to perform a denial-of-service (DoS) attack, violating the availability goal **G5**. Moreover, an adversary **A2** with temporary access could, in addition to running the same DoS attack, attempt to exploit the memory leak and write capabilities (which violate the memory safety goal **G4**) to leak secret keys embedded in the lock or gain privileged access to the lock. However, without access to the firmware, such exploitation is challenging and we did not further investigate this attack vector. We tried extracting the firmware or the embedded secret keys by measuring signals between pins on the circuit board, but it failed because the lock seems to be using integrated memory and flash together inside the CPU unit. We did not try side channel or fault injection attacks for firmware extraction. Noise from BLE communication might be an obstacle for the former, but the latter is a promising avenue for future analysis.

6.6 Further Security Issues

Command Fingerprinting. One issue with the use of a stream cipher for command encryption is that the ciphertext lengths directly correspond to the plaintext lengths of the encrypted command. Hence, despite the encryption, an eavesdropper can infer that the encrypted command belongs to the subset of commands with a given length. While this may not directly lead to a compromise of the security goals, it does degrade the confidentiality guarantees of the encrypted communication channel between the lock and the phone. This could strengthen other attacks. For instance, since the smart lock reuses the same nonce for every connection of the same user (cf. Section 6.1), an adversary can swap commands from different connections that were sent at the same position (e.g., the fourth command of two different exchanges). This allows the attacker to customize the series of executed commands, which may be useful for a more sophisticated exploit chain.

Custom CCM Encryption Scheme. Although CCM encryption is standardized [2] and has a security proof by Jonsson [8], we have some concerns with Master Lock's custom variant described in Section 5.2. Master Lock's nonce only contains six random bytes (see Section 5.2), which is less than the seven bytes needed for Jonsson's proof and, crucially, smart lock D1000 does not guarantee nonce freshness. Indeed, our Attack 1 (Section 6.1) exploited the fact that the same nonces are used with the same key for different connections. While the two-byte counter is somewhat small, the maximum transmission unit for BLE (at most 512 bytes), as well as the limit on command lengths, guarantee that none of the encrypted plaintexts are too long.

7 Proof of Concept

To evaluate the correctness of our analysis and the effectiveness of the attacks in Section 6, we wrote an Android app implementing the BLE protocols we reverse engineered. The firmware version of the Master Lock Bluetooth Deadbolt D1000 used in our experiments was `1679338484`, which was the latest at the time of testing (November 5, 2024). We suspect that this number represents the firmware build time or release time (parsed at a timestamp, it corresponds to March 20, 2023, at 11:54:44), which supports our hypothesis that the BLE protocols do not change frequently.

7.1 Communication Protocol Sanity Checking

We first simulated benign BLE communications with the lock following the protocols we reverse engineered to verify the correctness of our inferred protocols and commands. We used a valid profile P received from the enterprise API server to authenticate the `StartSession` command. Then we tried all supported commands we discovered from the app on the lock, including the `KeepAlive` command, but excluding the `FirmwareUpdate` command. All our commands functioned as expected. We could unlock, relock, read and write passcodes, read and write clock time, read audit logs, read battery levels and temperatures, read and write other configurations, and so on. The connections were maintained if we sent the `KeepAlive` commands every five seconds when there were no other ongoing communications. If we did not send the `KeepAlive` commands when idle, the connections were closed by the lock after nine seconds. These results confirm the correctness of our reverse engineering.

Furthermore, we authenticated to API endpoints using the reverse engineered license and password, and checked that we received valid responses.

7.2 Attack Verification

We confirmed that an adversary can reuse the same access profile to authenticate across different sessions and replay all commands in recorded sessions (Attack 1, Section 6.1). Moreover, we verified that access profiles remain valid after revocation (Attack 2, Section 6.2). The `WriteTime` command allowed us to set an arbitrary clock time for the smart lock, into the past or the future (Attack 3, Section 6.3). To test the forgery of audit events (Attack 4, Section 6.4), we uploaded two events for locking and unlocking the smart lock (`Open` respectively `Close`), each associated with a timestamp. We confirmed that the upload succeeded by observing the forged events in the list of events on the enterprise web application page. The latest audit event index returned by the SDK server also increased by two.

To test sending malformed messages (Attack 5, Section 6.5), we enumerated all possibilities for the encoded message length. We found that if the encoded length is less than 610 and does not match the actual length, the lock closes the connection immediately and generates an audit event of invalid wireless access, indicating replays. If the encoded length is greater than 6561, the lock immediately reboots. For the memory corruption, we observed that the audit event index was overwritten when we set the encoded length to a large non-crashing value (e.g., 609), which causes the `ReadAuditTrail` to return random event data. We did not have access to the smart lock firmware to confirm that this was a buffer overflow vulnerability, but after disclosure Master Lock confirmed it.

8 Disclosure and Mitigation

We reported our findings to Master Lock on March 10, 2025. The security team of Fortune Brands Connected Products—the company owning Master Lock as well as Yale and August smart locks—responded on March 14. We had an in depth meeting with their team in which they acknowledged our findings in detail, provided context on the origin of these issues, insights into their design decisions, and updates on the mitigation progress. We summarize this discussion below.

Session replay (Section 6.1). Sessions can be replayed because they reuse the same key and starting nonce, which makes messages sent by a legitimate user in a previous session valid in a later session. To mitigate this, the lock should choose a fresh random nonce from a cryptographically large space for every session. This will cause replayed packets to fail the authenticity check during decryption with CmdDec with high probability because the packet nonce will not match the session nonce. Master Lock informed us that among their products, only the D1000 firmware resets the nonce when starting connections. They will release a firmware update for the D1000 in June 2025 that randomizes the nonce.

Exceeding Access (Section 6.2). Mitigating this vulnerability faces a challenging tradeoff between functionality and security. A simple solution would be to require the lock to communicate with one of the API servers over a secure channel (e.g., authenticated by the shared symmetric secret used for firmware updates) and verify that an access profile is still valid. However, this requires an Internet connection, and the lock relies on the phone to relay packets to communicate with API servers. Master Lock explained to us that some of their products are used in remote locations with sparse Internet access (e.g., a cabin in the forest). Users still expect such locks to work with the phone,[8] which prompted Master Lock to issue access profiles that are valid for a fixed period. A mobile phone storing such a profile can unlock the smart lock without Internet access until the profile expires. Of course, this trades off security for functionality, because the same time period also defines how long it takes for revocation to take effect.

Master Lock products usually allow access profiles to be valid for one week. Interestingly, the D1000 profiles that we found and exploit in Section 6.2 are an exception with the much longer validity period of nine months. Master Lock explained that this exception was due to a firmware bug that previously caused the real time clock of the D1000 lock to jump forward in time. To avoid locking users out, they increased the validity period to nine months.

This firmware bug is already fixed in the D1000 firmware that we targeted. Master Lock planned to push out firmware updates to D1000 locks more aggressively, and then reduce the validity period of access profiles to seven days, as in their other products. They later shared that their remediation is in progress and should be completed by June 2025. This mitigation was chosen to still support the offline use case. Our attack still applies, but the attack window is shorter.

Clock Tampering (Section 6.3). This attack shows that the clock time of the smart lock is security critical to enable the lock to accurately assess the validity of access profiles. Master Lock explained that a lock always fetches the time from a trusted server if the phone offers a connection. We suggest that these time updates should be authenticated to prevent the

[8]Some locks, including the Deadbolt D1000, have a physical numeric pad, where the user can still enter their access code manually.

phone from tampering with the time when relaying packets. For this, they could use the lock-specific symmetric key that is used to protect for firmware updates and access profiles to compute a message authentication code (MAC) for the time update. Such authenticated time updates should be protected against replay. They also rely on honest users to relay them to the lock since malicious users can choose to drop them.

However, for the case of using the smart lock offline, clock drift still needs to be corrected to avoid locking out users. Thus, Master lock continues to allow *trusted* users to use the `WriteTime` command to set the clock of the lock to the user's phone time. Their app only allows this operation if a user has admin privileges and the phone is offline at that moment. Unfortunately, the access profiles for non-admins also have the permission set that allows them to perform this operation. Our attack circumvented their app checking logic by implementing the communication protocols ourselves and sending the `WriteTime` command directly to the lock.

Audit Log Tampering (Section 6.4). We recommend that audit events are generated on the lock and authenticated with a MAC using the lock-specific symmetric key that is shared with the API servers. The adversary does not know this key, so they can no longer forge audit events or modify them when they are relayed over the phone. Master Lock first expressed concern about the complexity of the change, as it requires changes in the backend and the smart lock firmware. They also argued that audit logs are already imperfect: it is possible that many events happen before the lock can send them to the server (e.g., in the offline use case). Since the lock has limited resources, it uses a ring buffer to store events and may overwrite older entries before they can be stored on the server. From the start, Master Lock said they will address the lack of authentication for the audit event API endpoint, and only allow users with permission to access an audit log to upload new events for it. However, this API access restriction still allows existing users (e.g., with temporary access) to upload bogus audit events, which could be avoided by adding MACs. Master Lock later shared that they decided to fully mitigate our attack after all, but did not share more technical information.

Malformed Messages (Section 6.5). This attack highlights that testing with invalid inputs is especially important when communicating over untrusted channels or with untrusted entities. Master Lock confirmed that this vulnerability is due to a buffer overflow in the firmware. There are many tools to detect and avoid such memory safety vulnerabilities, such as AddressSanitizer [18], fuzz testing, or the use of memory-safe programming languages like Rust. Master Lock shared that they included boundary checks with C assertions in their builds to protect against such overflows but realized that those assertions were removed for production builds. The D1000 firmware update of June 2025 that mitigates Attack 1 will also patch this buffer overflow and undergo more extensive testing.

Ethical Considerations. We made efforts to carry out our research responsibly: we minimized our interaction with Master Lock web servers and only ever interacted with test accounts for users and smart locks we owned.

We believe that this type of product analysis and the lessons learned help to improve the security of Master Lock products for their end users. Our work also highlights the need for further protocol analysis of other smart locks so that vulnerabilities can be discovered and patched before malicious actors find and exploit them in the wild and cause real harm.

9 Discussion

Despite Master Lock's efforts to secure their product by authenticating APIs, encrypting communication, and obfuscating their mobile app, our research shows that more sophisticated protocol vulnerabilities remain and still compromise various security goals.

Insecure Protocol Design. The presence of encryption in the BLE communication does not suffice to create a secure protocol. Master Lock claims to use "military-grade authentication and encryption mechanisms built upon proven, NIST recommended and FIPS approved algorithms to deter sniffing, replay and manipulation attempts" [15]. This encryption was enough for prior work to conclude that there are no replay attacks [1], and to deter further analysis of the Bluetooth protocol [9]. Nonetheless, our Attack 1 (Section 6.1) achieves exactly such a replay attack. It turns out that Master Lock is only secure against replaying single BLE messages inside the same session. However, their smart lock reuses the same nonce at the start of every session, which enables an adversary to record and replay full sessions. This shows that secure cryptographic protocol design requires cryptographic building blocks to be composed and used correctly.

Secure Lock-to-API Server Channel. The Master Lock smart lock can only communicate with the API server through the user's smart phone. While previous work [6, 22] studied this "gateway architecture" as a new deployment model that has advantages and disadvantages over directly connecting the lock to Wi-Fi, we observe that Master Lock did not design a protocol with an end-to-end secure channel between the lock and the API server. This problem is the root cause of Attack 3 described in Section 6.4: if audit events had been sent from the lock to the telemetry server over an authenticated channel, it would not have been possible to forge them.

However, establishing such a channel is challenging. While BLE offers pairing and bonding to establish a secure channel between the phone and lock, this does not prevent the phone from injecting malicious time updates. Moreover, the API authentication with user credentials and secret material embedded in the app does not prevent a malicious user from uploading forged audit events. Instead, Master Lock faced the problem of establishing a secure channel across two protocols: IP (server-to-phone) and BLE (phone-to-lock). They appear to have successfully created such a secure channel for firmware updates, which uses a static symmetric secret key embedded in the lock, but not for other communication between the phone and the server.

Another challenge is that the API server may not be reachable when a user unlocks the lock, and vendors face a difficult choice between security and limiting functionality. For example, if time updates were only done when the phone has an Internet connection, then they could be authenticated by the server, and a malicious guest would not be able to control the lock's clock time (Attack 2, Section 6.3). However, smart lock clocks in remote areas could drift over time, which could lead to the lock rejecting valid access profiles when its internal time is set to a future date.

A significant effort to secure IoT deployments produced Matter [10], a standard that focuses on interoperable and secure integration of IoT devices from various manufacturers and over different protocols. To bridge protocols, Matter requires trusted components (called controller and commissioner) to provision devices and issue certificates. However, in a smart lock deployment, the user's phone cannot always be trusted. Therefore, an interesting question for future work is how one can extend Matter or design another framework for IoT devices in gateway deployments to support secure protocol development, taking into account that (1) the communication must run over multiple protocols, (2) the IoT device only has limited energy and computational resources and might run on custom hardware and (3) the IoT device may be offline at the time of interaction.

10 Related Work

Closest to our work is an earlier analysis of the Master Lock Bluetooth padlock by Knight, Lord, and Arief [9] in 2019. They report that the Master Lock APK was unobfuscated and the API endpoints had weak authentication using API keys that could be bypassed using static, hard-coded credentials. They also discovered that the Real Time Clock (RTC) of the padlock is paused when the battery is removed, and resynchronized with the clock time of the padlock owner's phone. They conclude that this attack is not feasible in practice, as the padlock needs to be open to remove the battery and the attack requires access to the owner's phone. In comparison, our Attack 3 in Section 6.3 exploits our understanding of Master Lock's protocol to send the underlying command used for clock synchronization in cases where the official app would not send it, making clock tampering practical even on newer Master Lock products that include mitigations against these historical vulnerabilities. Knight, Lord, and Arief attempted to sniff the Bluetooth communication between the lock and the phone, but were not able to find any vulnerabilities. We reverse engineered the encryption protocol (Section 5.2) and implemented our own client, which allowed us to directly interact with the lock. We did not further investigate an access control feature allowing guests to open the lock only during certain hours by generating new temporary codes directly with the webserver API, which Knight, Lord, and Arief were able to bypass. Finally, Knight, Lord, and Arief found a misconfigured API point that leaks a primary code to guest users, which allows them to unlock the padlock even after their access was revoked. They reported their findings, and Master Lock fixed the API server misconfiguration and authentication.

Two security analyses of the August smart lock [7, 21] found multiple attacks either due to the lack of security measures or a very strong adversary model. This lock works similarly to the Master Lock we analyzed: the lock also only communicates through the user's phone with a web server hosted by August. The attack of [7] also bypasses certificate pinning and elevates guest access to admin privileges by simply replacing the user type field in the API request. Both research teams find that the August smart phone app stores critical information in plaintext, including the handshake key (which allows an adversary to open the door) and the firmware encryption key. However, retrieving it requires privileged access on the owner's phone. Ye, Jiang, Yang and Yan [21] also find a DoS attack whose root cause is that the August smart lock can only handle one request at a time and insufficiently restricts access to such operations against unauthorized users. In comparison, our attacks achieve unlocking and other security violations with a weaker adversary that does not require access to the lock owner's phone.

Rose and Ramsay [1] analyzed 16 Bluetooth locks and presented vulnerabilities against 12 of them at DEF CON 24. The Master Lock padlock was one of the four smart locks for which they were not able to find any vulnerabilities. They found passwords exchanged in plaintext over BLE in four locks (two Quicklock models, iBluLock, and Plantraco). Furthermore, they compromised five locks (Ceomate, two Elecycle models, Vians, and Lagute) with replay attacks, where they unlocked the locks by replaying a recorded (and possibly encrypted) exchange of the lock and an authorized user. Using fuzzing, they discovered an error in the Okidokeys Smart Doorlock, where the lock would open when a particular byte of the key is set to zero. They decompiled the APK of the Android app for the Danalock Doorlock and found a hard-coded password that was used to secure the communication. In a more sophisticated attack on the Mesh Motion Bitlock Padlock, they were able to exploit a predictable nonce to impersonate the smart lock towards the user and steal their password.

Ho et al. [6] investigate the presence of unauthorized unlocking (**G1**) and state inconsistencies in five smart locks (August, Danalock, Kevo, Okidokeys, and Lockitron). They introduce the term "Device-Gateway-Cloud (DGC) architecture" for the deployment model that Master Lock also follows, where the smart lock connects to a web server of the vendor using the owner's phone as a gateway. They consider four types of adversaries: physically present (corresponds to **A1**), a "revoked adversary" (**A2**), and two additional ones not considered in our work called "relay attacker" and "thief". The latter steals the owners authorized device. They find generic revocation evasion attacks against all smart locks using the DGC architecture by simply putting the phone into airplane mode to prevent it from fetching updated revocation information. Our Attack 2 (Section 6.2), which also allows an adversary to exceed access, works even if the smart lock fetches updates because it exploits a disconnect between logical revocation and the expiration of the underlying key material used by the protocol. Ho et al. also cut Internet access on the gateway to cause state inconsistencies by preventing audit events from being uploaded. Our Attack 4 (Section 6.4) not only prevents the creation of audit events but allows an adversary to forge them. In summary, their paper explores generic attacks against locks with the DGC architecture while we explore targeted attacks against the Master Lock which require a deeper understanding of the protocol and exploit lower-level assumptions.

The analysis of the DGC architecture was more recently extended by Zhou et al. [22] to the concept of Mobile-as-a-Gateway (MaaG) IoT. They analyze ten different smart locks (from Level, August, Yale, Ultraloq, Kwikset Aura, Honeywell, Schlage, Geonfino, Tile, Chipolo). They find different vulnerabilities related to the maintenance and synchronization of access policies, identifying inconsistencies in all analyzed smart locks including static key material that is not rotated on revocation. Our work shows that such attacks are still present in other smart lock models, e.g., our Attack 2 (Section 6.2) is similar to some of their findings. In addition, our work explores a broader range of attack vectors using a deeper analysis of a single smart lock protocol, showing that the ecosystem also suffers from regular security vulnerabilities independent of the MaaG architecture.

Other work, such as an analysis of SmartThings by Fernandes, Jung, and Prakash [3], analyzed a different deployment model. These architectures provide a platform for various different smart home applications, including locks that are directly connected to the Internet. This work is less closely related to ours and such architectures face other challenges.

Levi et al. [11] describe relay attacks against (non low-energy) Bluetooth. Rather than analyzing and breaking the authentication protocol, these attacks relay the physical Bluetooth signal between the IoT device and a legitimate user to trick the device into assuming the user is nearby and authenticating. Similar techniques were used to open car doors that use passive keyless entry systems by capturing the key's signal with an antenna and amplifying it [4].

Lonzetta et al. [12] survey various Bluetooth attacks against IoT devices. We already discussed attacks on BLE, and in particular for smart locks, above. The remaining attack on non-BLE Bluetooth include downgrade attacks, PIN cracking, and static link keys in older Bluetooth protocol versions. Most other attacks are against specific implementations and exploit unauthenticated messages, device impersonation via MAC spoofing and other techniques, and exploiting leaked hard-coded secret key material which may be extracted or stolen by malware on the device.

11 Conclusion

We analyzed the Master Lock Deadbolt D1000 smart lock and found five attacks through reverse engineering their protocol: First, although prior work concluded that Master Lock was not vulnerable to replay attacks, we find that nonce reuse in their variant of AES-CCM still allows session replay including unauthorized unlocking. Second, underlying user profile key material is not revoked appropriately which allows former guests to still open the lock after their access ended. Third, we found that temporary guests can bypass application-level restrictions by directly talking to the lock and set the lock's clock to arbitrary values to avoid expiration or lock valid users out. Fourth, the telemetry server API has insufficient authentication and allows any adversary knowing the device identifier of a lock to forge entries or prevent legitimate once from being uploaded. Finally, malformed messages allow an adversary to overwrite and leak memory or make the lock inoperable until a reboot. While Attacks 1, 2, and 5 are specific to the Deadbolt D1000 product, Attacks 3 and 4 likely also apply to other Master Lock products.

The root causes for these attacks include that Master Lock does not properly establish a secure channel between the lock and their servers due to mistakes in the encryption scheme, reliance on code obfuscation to hide secrets, and insufficient identity and data authentication. We reported these vulnerabilities, and Master Lock has taken steps to mitigate them.

We encourage more analysis of the security of smart lock protocols, as well as efforts to build a secure framework that allows IoT vendors to securely establish channels to their deployed devices, including through paths involving multiple protocols and untrusted devices.

Acknowledgments

Danielle Dang, Sierra Lira, and Angela Tsai participated in this project as part of the Early Research Scholar Program (ERSP) at UCSD, run by Christine Alvarado, Ruanqianqian (Lisa) Huang, and Javahir Abbasova, which introduces second-year undergraduates to research through a year-long project. Miro Haller was supported by a Google PhD fellowship. We thank the Fortune Brands Connected Products security team for a positive disclosure experience and for sharing their detailed analysis and design decisions with us. Finally, we are grateful to Keegan Ryan for attempting to extract the smart lock firmware with us.

References

[1] Ben Ramsey Anthony Rose. Picking bluetooth low energy locks from a quarter mile away. https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Rose-Ramsey-Picking-Bluetooth-Low-Energy-Locks-UPDATED.pdf, 11 2016. Presentation at DEF CON 24.

[2] Morris Dworkin. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality, 7 2007.

[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] Aurélien Francillon, Boris Danev, and Srdjan Capkun. Relay attacks on passive keyless entry and start systems in modern cars. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[5] Hussein Hazazi and Mohamed Shehab. Exploring the usability, security, and privacy of smart locks from the perspective of the end user. In *Nineteenth Symposium on Usable Privacy and Security (SOUPS 2023)*, pages 559–577, Anaheim, CA, August 2023. USENIX Association.

[6] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 461–472, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Jmaxxz. Backdooring the frontdoor. https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Jmaxxz-Backdooring-the-Frontdoor-UPDATED.pdf, 11 2016. Presentation at DEF CON 24.

[8] Jakob Jonsson. On the security of CTR + CBC-MAC. In Kaisa Nyberg and Howard Heys, editors, *Selected Areas in Cryptography*, pages 76–93, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[9] Edward Knight, Sam Lord, and Budi Arief. 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)*, pages 835–842, 2019.

[10] Silicon Laboratories. Introduction to Matter. https://siliconlabs.github.io/matter/latest/general/FUNDAMENTALS_INTRO.html#why-matter, 2025. Accessed: 2025-03-11.

[11] Albert Levi, Erhan Çetintaş, Murat Aydos, Çetin Kaya Koç, and M. Ufuk Çağlayan. Relay attacks on bluetooth authentication and solutions. In Cevdet Aykanat, Tuğrul Dayar, and İbrahim Körpeoğlu, editors, *Computer and Information Sciences - ISCIS 2004*, pages 278–288, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[12] Angela M. Lonzetta, Peter Cope, Joseph Campbell, Bassam J. Mohd, and Thaier Hayajneh. Security vulnerabilities in Bluetooth technology as used in IoT. *Journal of Sensor and Actuator Networks*, 7(3), 2018.

[13] Shrirang Mare, Franziska Roesner, and Tadayoshi Kohno. Smart devices in airbnbs: Considering privacy and security for both guests and hosts. *Proceedings on Privacy Enhancing Technologies*, 2020.

[14] Master lock vault. https://www.masterlockvault.com/. Accessed: 2025-02-09.

[15] Master lock vault enterprise user guide. https://cdn.masterlock.com/electronic-products-support-documents/Vault-Enterprise-User-Guide.pdf. Accessed: 2025-02-09.

[16] Grand View Research. Smart lock market size and share, industry report, 2030. https://www.grandviewresearch.com/industry-analysis/smart-lock-market, 4 2025. Accessed: 2025-05-28.

[17] Michael Sainato. Master lock's milwaukee plant to close after 100 years and send jobs abroad. https://www.theguardian.com/us-news/2023/jun/29/master-lock-milwaukee-plant-closure-manufacturing-holdout, 6 2023. The Guardian.

[18] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[19] Girish Sharma, M Mabrishi, K Hiran, and Ruchi Doshi. Reverse engineering for potential malware detection: Android apk smali to java. *Journal of Information Assurance & Security*, 15(1):26–34, 2020.

[20] Chia-Sheng Tsai and Cheng-I Hung. An enhanced secure mechanism of access control. In *2010 Second International Conference on Communication Systems, Networks and Applications*, pages 119–122, 2010.

[21] Mengmei Ye, Nan Jiang, Hao Yang, and Qiben Yan. Security analysis of Internet-of-Things: A case study of August smart lock. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 499–504, 2017.

[22] Xin'an Zhou, Jiale Guan, Luyi Xing, and Zhiyun Qian. Perils and mitigation of security risks of cooperation in Mobile-as-a-Gateway IoT. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 3285–3299, New York, NY, USA, 2022. Association for Computing Machinery.

A Additional Reverse Engineering Results

This section collects additional insights from reverse engineering, which provide informative context but are not directly used in our attacks.

A.1 BLE UUID Characteristics

Table 3 shows the UUIDs for BLE communication that we extracted during reverse engineering (as described in Section 4).

A.2 Binary Structures Of Profile And Firmware Update Commands

As discussed in Section 5.1, the firmware update command $\mathsf{cmd}_{\mathsf{firmware}}$ and profile P bytes are directly relayed to the lock, without parsing them on the phone. Hence, we cannot find their structure in the mobile app source code. Nevertheless, there are patterns in the samples we collected from the enterprise API server. We inferred that the command $\mathsf{cmd}_{\mathsf{firmware}}$ and profile P have the same binary structures. They both start with six bytes of the shortened nonce n_s, followed by two bytes for the data length (in big-endian encoding), the data, and—finally—an eight-byte message authentication code (MAC). Excluding the six-byte shortened nonce at the beginning, the remaining parts have the same structure as the output of the encryption function CmdEnc. Thus, we suspect that the command $\mathsf{cmd}_{\mathsf{firmware}}$ and profile P are also produced using their custom AES-CCM scheme, with the first six bytes as the n_s and the rest as the ciphertext that can be decrypted and authenticated by the decryption function CmdDec. This would imply that the SDK server and the enterprise API server must, for every lock, know a symmetric secret key that is embedded into the lock. These keys must be fixed at least for the same firmware version and without factory resets for the same lock. We do not know whether the keys are different for different locks, because we only have one lock on which we can experiment.

We hypothesize that the profile P contains the following information: The length of the plaintext is 54 bytes according to our inference above, which should contain a 32-byte session key, four-byte access start time and four-byte access end time both encoded as seconds since "the epoch," four-byte user id, six-byte device id, and four-byte remaining unknown (but could be a session key identifier, some counters, or firmware version). We cannot verify these because the app never parses the profile P and we were unable to extract the firmware or the symmetric secret key from the lock.

Table 3: Relevant UUIDs for reverse engineering the Master Lock phone-to-smartlock communication via BLE.

Shorthand	**UUID**	**Description**
$\texttt{UUID}_{\texttt{device}}$	`94e00001-5d5b-11e4-846f-4437e6b36dfb`	Normal service UUID
$\texttt{UUID}_{\texttt{boot}}$	`94e00000-5d5b-11e4-846f-4437e6b36dfb`	Bootloader (firmware update) service UUID
$\texttt{UUID}_{\texttt{conn-char}}$	`94e00002-5d5b-11e4-846f-4437e6b36dfb`	Characteristic UUID
$\texttt{UUID}_{\texttt{conn-desc}}$	`00002902-0000-1000-8000-00805f9b34fb`	Descriptor UUID