REVERSEC

"Open, Sesame!"

unlocking Bluetooth padlocks with polite requests

Alex Pettifer
Miłosz Gaczkowski

Introductions

Introductions: Miłosz

- Miłosz Gaczkowski
  - /ˈmi.wɔʂ/
- Past life: University teaching
  - Computer science
  - Cybersecurity
- Current life: Mobile Security Lead at Reversec
- Dabble in IoT and thick client testing too
- Enjoys obscure power metal and the colour purple
  - Pink is ok too
- Twitter: @cyberMilosz
  - I think I'm on BlueSky too???

Introductions: Alex

- Alex Pettifer
- Cyber-consultant
  - Mobile & IoT stuff
- Likes locks
- Fan of rats
- Got rejected from Warwick because of my A-levels
  - Still salty

Why are we here?

Why are we here?

- Today's talk started as an intern project on smart padlocks
- Cross-section of physical and mobile app security
- Original goals:
  - Learn a little bit about Bluetooth Low Energy (BLE)
  - Build experience in mobile application reverse-engineering
- Got some interesting findings:
  - tl;dr: anyone can unlock any padlock by just asking nicely
- Our goals for today:
  - Entertainment
  - Technical understanding and fun findings
  - The process - so you can do similar things!

Key questions

Could a malicious user/device...

...listen in on and replicate the unlock signal?

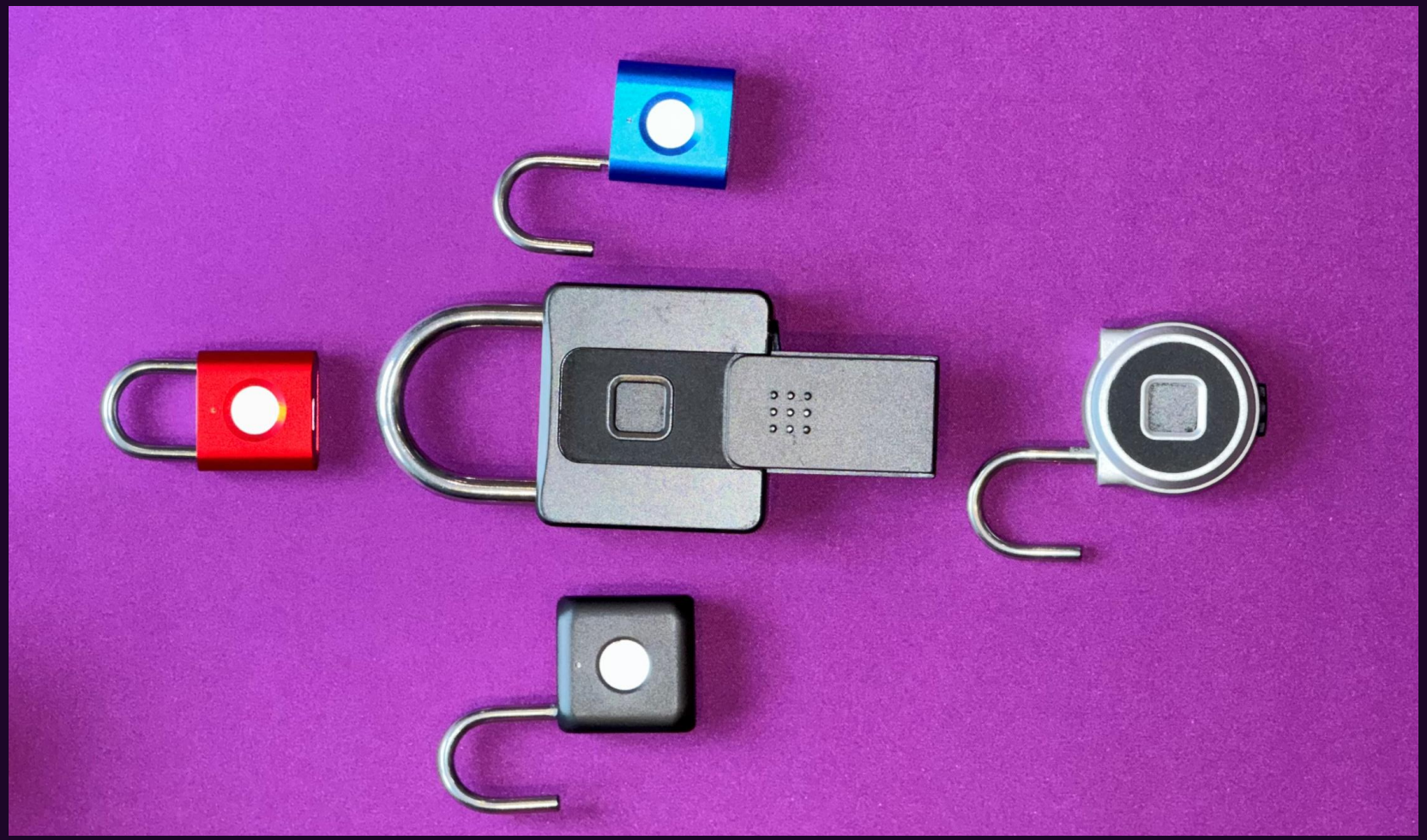
...tamper with the lock in other ways?

How much information would you need?

The locks

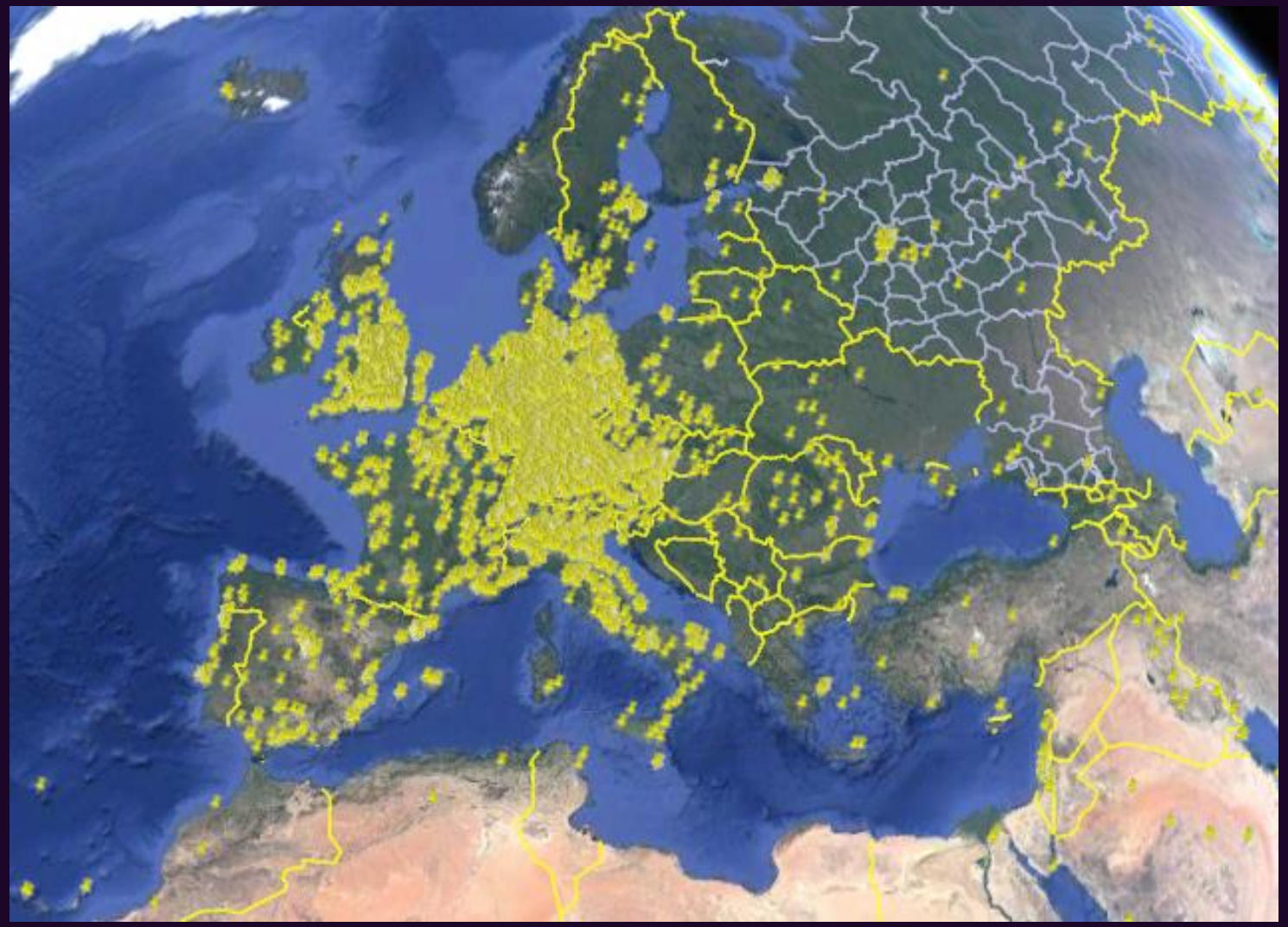
- Locks:
  - eLinkSmart range
  - Also known under other brands: Anweller, eseesmart, and others
- Rationale for specific lock choice:
  - Prominent on Amazon UK
  - Heavily advertised
  - Cheap == accessible
  - Seemingly also popular on other marketplaces, esp. Germany, Poland
- Functionality:
  - (Some) have keys
  - All have local fingerprint auth
  - Most have remote Bluetooth LE unlock
    - Supported by mobile app

The locks



The locks – where are they?

- here



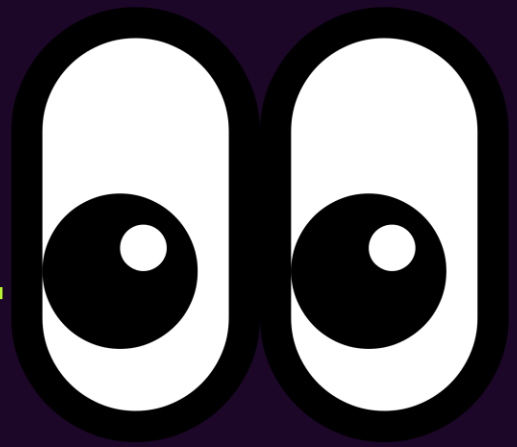
The locks – where are they?

- and here

The locks – where are they?

- and definitely here

Epic foreshadowing



Tooling, approach, and process

Methodology

Intercept and understand BLE communications

Tools used: Wireshark and nRF Sniffer, or a mobile phone

01

Decompile and reverse-engineer the application

Tools used: Frida, jadx-gui, and ADB

02

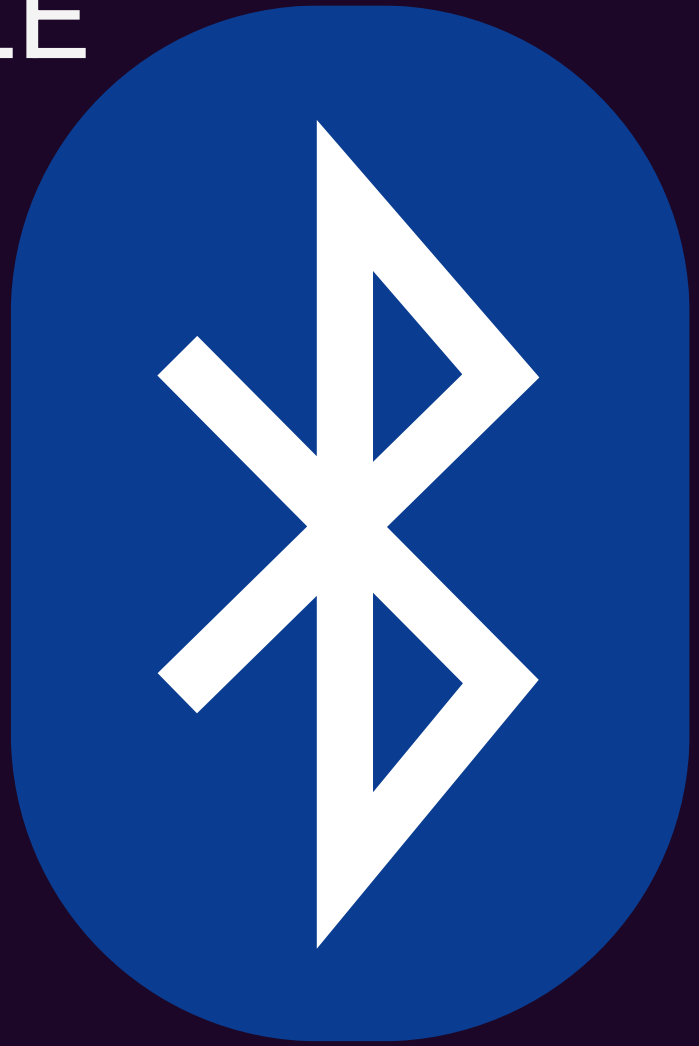
Inspect HTTPS communications

Tool used: Burp Suite

03

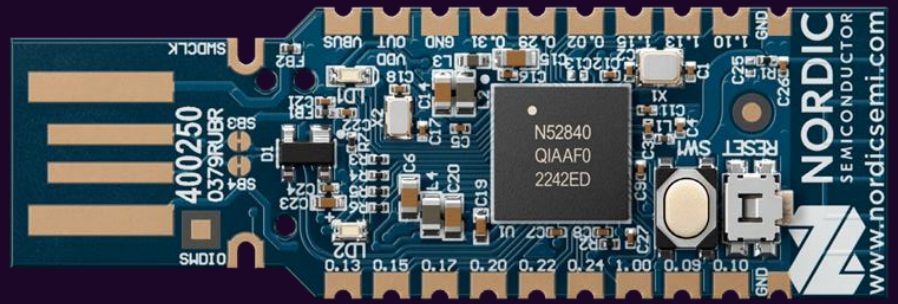
A quick primer on Bluetooth LE

- Short range communication
- Over radio
- Embedded encryption is possible
  - But not always used
- Sources and destinations identified by MAC addresses
  - This is public information – think IP addresses
- Otherwise – it's just standard I/O



Intercepting BLE

- We decided to use an external BLE sniffing device, as opposed to HCI dumping on the device.
  - This was to model and understand what was possible from an external perspective
- For this we used the nRF52840, with the nRF sniffer software, both available from Nordic Semiconductor
- From here the intercepted BLE communications were displayed in Wireshark



File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

(btatt.handle) && (btatt.value)

No.	Time	Source	Destination	Protocol	Length	Info
540	12.643767	Master_0x0a39c2a2	Slave_0x0a39c2a2	ATT	35	Sent Write Request, Han
550	12.823768	Master_0x0a39c2a2	Slave_0x0a39c2a2	ATT	51	Sent Write Command, Han
557	13.048997	Slave_0x0a39c2a2	Master_0x0a39c2a2	ATT	53	Rcvd Handle Value Notif
559	13.049673	Slave_0x0a39c2a2	Master_0x0a39c2a2	ATT	53	Rcvd Handle Value Notif
561	13.050349	Slave_0x0a39c2a2	Master_0x0a39c2a2	ATT	43	Rcvd Handle Value Notif
644	15.478773	Master_0x0a39c2a2	Slave_0x0a39c2a2	ATT	53	Sent Write Command, Han
646	15.479450	Master_0x0a39c2a2	Slave_0x0a39c2a2	ATT	47	Sent Write Command, Han
684	16.559006	Slave_0x0a39c2a2	Master_0x0a39c2a2	ATT	51	Rcvd Handle Value Notif

```
Frame 550: 51 bytes on wire (408 bits), 51 byte
nRF Sniffer for Bluetooth LE
Bluetooth Low Energy Link Layer
Bluetooth L2CAP Protocol
Bluetooth Attribute Protocol
  Opcode: Write Command (0x52)
  Handle: 0x0008 (Mesh Proxy Service: Unknown)
  Value: 10006981c012eeed33a97ca72b1769338cca
```

```
0000  00 2c 00 03 14 05 06 0a  03 24 19 6c 00 2d f5 d2
0010  04 a2 c2 39 0a 02 19 15  00 04 00 52 08 00 10 00
0020  69 81 c0 12 ee ed 33 a9  7c a7 2b 17 69 33 8c ca
0030  3d b4 a9
```

Value (btatt.value), 18 bytes — Packets: 688 · Displayed: 8 (1.2%) — Profile: Default

Reversing packets

```
Phone -> Smartlock: 1000c96e581aed958a5865a8b7ebabb45cc6
SmartLock -> Phone: 300058ab9ae5715e2f6b254f5da1ef8c86493a28
SmartLock -> Phone: 3cef5fb77eba952b25e76801ba4e4d8dd69e0975
SmartLock -> Phone: 0c1fdda8f325ac489a01
Phone -> Smartlock: 1000bb822881069dc139f95273b0f203e7b6
SmartLock -> Phone: 1000756178b35d6b4ed952a04392324ce616
```

- The messages were constructed such that long messages were split into multiple packets, with the first two bytes of the message being the length.
- The messages themselves all had two traits in common that strongly indicated encryption was being used:
  - Seemingly random
  - Every length was an exact multiple of 16 bytes, implying a block cipher
- Clearly some encryption was being performed by the application

Reverse-engineering the app

- Pulling the application and loading it into jadx revealed heavy obfuscation
- All classes, methods and variables were renamed to single characters
- However, a pattern was found. Custom log statements
- Most important methods had one or two log statements with a similar format `"ClassName – methodName – message"`
- From here deobfuscation was straightforward, if time consuming. Class and method names were now in plaintext, and most variables were named explicitly in the logs

Obfuscated

```
public static byte[] T(int i2, String str) {
     byte[] bArr = new byte[18];
     System.arraycopy(Packet.shortToByteArray_Little((short) 16), 0, bArr, 0, 2);
     System.arraycopy(Packet.shortToByteArray_Little((short) 18), 0, bArr, 2, 2);
     System.arraycopy(Packet.intToByteArray_Little(i2), 0, bArr, 4, 4);
     System.arraycopy(Packet.intToByteArray_Little((int) (c.g.a.a.s.h.x() / 1000)), 0, bArr, 8, 4);
     byte[] bytes = str.getBytes();
     System.arraycopy(bytes, 0, bArr, 12, bytes.length);
     c.n.a.i g2 = c.n.a.f.g("BleProtocolUtils");
     g2.j("--packageUnlockCloudPwd-- bUlkCloudPwd:" + c.g.a.a.s.a.c(bArr, ","));
     return p(bArr);
 }
```

Deobfuscated

```java
public static byte[] packageUnlockCloudPwd(int token, String password) {
    byte[] packet = new byte[18];
    System.arraycopy(Packet.shortToByteArray_Little((short) 16), 0, packet, 0, 2);
    System.arraycopy(Packet.shortToByteArray_Little((short) 18), 0, packet, 2, 2);
    System.arraycopy(Packet.intToByteArray_Little(token), 0, packet, 4, 4);
    System.arraycopy(Packet.intToByteArray_Little((int) (DateUtil.getTimeInMillis() / 1000)), 0, packet, 8, 4);
    byte[] bytes = password.getBytes();
    System.arraycopy(bytes, 0, packet, 12, bytes.length);
    Logger classLogger = CustomLogger.classLogger("BleProtocolUtils");
    classLogger.log("--packageUnlockCloudPwd-- bUlkCloudPwd:" + ByteArrayUtils.asCSV(packet, ","));
    return encryptData(packet);
}
```

encryptData()?

Reversing the encryption

```
public static byte[] encryptData(SecretKeySpec secretKeySpec, byte[] bArr) throws GeneralSecurityException {
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
    cipher.init(1, secretKeySpec);
    return cipher.doFinal(bArr);
}
```

This was run by another function logging the class name as `BleAESCrypt`

```
private static SecretKeySpec getKey() throws UnsupportedEncodingException {
    return new SecretKeySpec("7b69b00b69420dce".getBytes(Constants.ENC_UTF_8), "AES");
}
```

Hardcoded AES key!

On encryption

- Symmetric encryption – same material used for encrypt and decrypt
- Asymmetric – the two are separate and not easily derivable from each other
- So:
  - Symmetric key
  - + we know the key
  - = we can encrypt and decrypt at will

Dissection of a packet

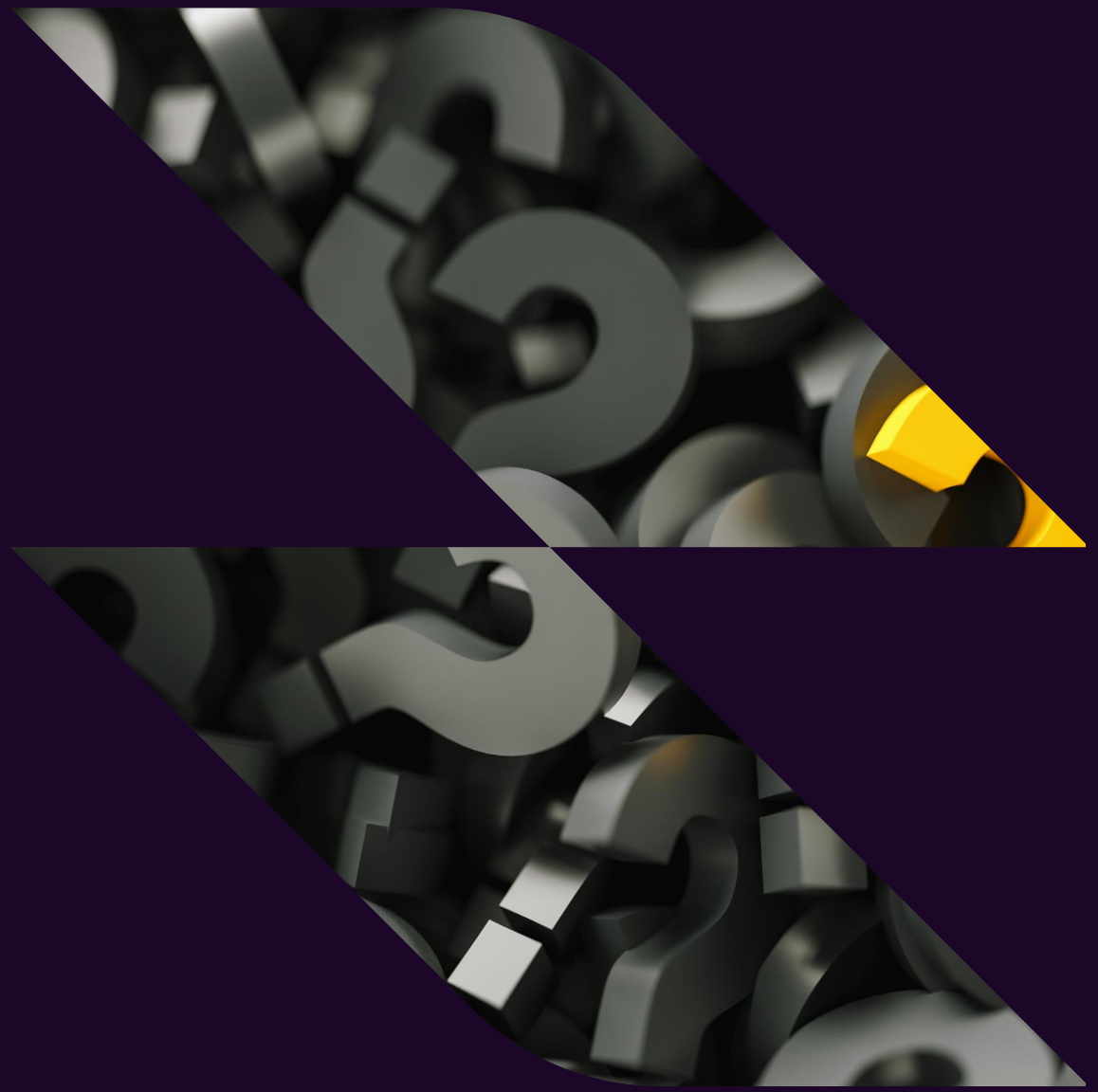
10001200	45512A0B	C3AFD064	34393632	3530
The total length of the packet (2-byte short)	The command code (2-byte short, 0x1200 = 18, the code for Unlock With Passkey)	The Login Token (4-byte integer)	The current date (4-byte integer)	ASCII-encoded passkey, in this case 496250

So how does it unlock?

- Request login token
  - Seemingly random, possibly to prevent replays
- Request unlock + provide 6-digit passkey
- Lock pops open
- At this point we have enough information to perform a replay attack*:
  - Observe unlock once
  - Find out what the passkey is
  - We can request login tokens and unlock the lock
- OK, so what is this passkey?
  - Seems to never change
  - Not even between lock factory resets, or between mobile devices for the same lock

* - sort of

Passkeys



We would like to understand where the passkey comes from. Early candidates:

- Hardcoded? (hopefully not)
- Generated from lock details somehow?
- Does it come from the Web?

Last option likely – you need to be online to pair a new lock, and offline functionality seemed like an afterthought

Let's explore Web traffic then!

Passkey requests

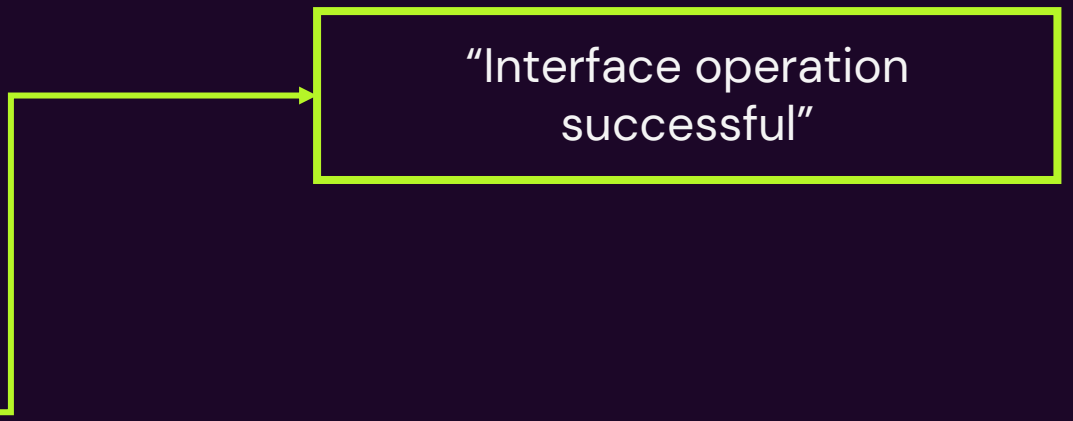
```
POST /?m=lock&a=getLockInfoByMac HTTP/1.1
Host: [...]
Content-Type: application/x-www-form-urlencoded
Content-Length: 109
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
User-Agent: okhttp/3.9.1

mac=A4:C1:38:21:95:CF&
user_name=testacct&
loginToken=54ab8b2a7b23216a1c1c461771a33052&
type=2&
cp=el
```

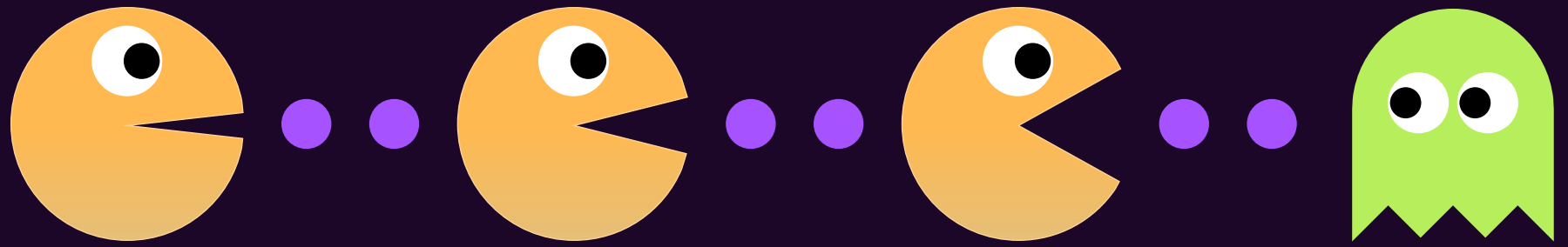
Passkey requests

```
HTTP/1.1 200 OK
[...]
X-Powered-By: PHP/7.2.24
Content-Length: 197


{
        "state":"success",
        "type":0,
        "desc":"接口操作成功",
        "data":
        {
                "name":"lock",
                "mac":"A4:C1:38:21:95:CF",
                "isBind":1,
                "password":"",
                "reset":1,
                "lock_status":1,
                "admin_password":"496250",
                "apply_mode":0
        }
}
```



We now understand the full chain



API Comms
Mobile app requests unlock code from API

Initial Handshake
Mobile app requests temporary token from lock

Construct unlock request
App builds BLE packet including previous info

Lock procesing
The lock confirms the validity of the token and passkey and, if successful, unlocks.

What's actually needed?

```
POST /?m=lock&a=getLockInfoByMac HTTP/1.1
Host: [...]
Content-Type: application/x-www-form-urlencoded
Content-Length: 109
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
User-Agent: okhttp/3.9.1

mac=A4:C1:38:21:95:CF&
user_name=testacct&
loginToken=54ab8b2a7b23216a1c1c461771a33052&
type=2&
cp=el
```

What's actually needed?

```
POST /?m=lock&a=getLockInfoByMac HTTP/1.1
Host: [...]
Content-Type: application/x-www-form-urlencoded
Content-Length: 109
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
User-Agent: okhttp/3.9.1

mac=A4:C1:38:21:95:CF&
user_name=testacct_randomjunk&
loginToken=randomjunk123123123&
type=2&
cp=el
```

What's actually needed?

```
POST /?m=lock&a=getLockInfoByMac HTTP/1.1
Host: [...]
Content-Type: application/x-www-form-urlencoded
Content-Length: 109
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
User-Agent: okhttp/3.9.1

mac=A4:C1:38:21:95:CF&
user_name=testacct_randomjunk&
loginToken=randomjunk1231123123&
type=2&
cp=el
```

What's actually needed?

```
POST /?m=lock&a=getLockInfoByMac HTTP/1.1
Host: [...]
Content-Type: application/x-www-form-urlencoded
Content-Length: 109
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
User-Agent: okhttp/3.9.1

mac=A4:C1:38:21:95:CF
```

Public information!

Putting it together

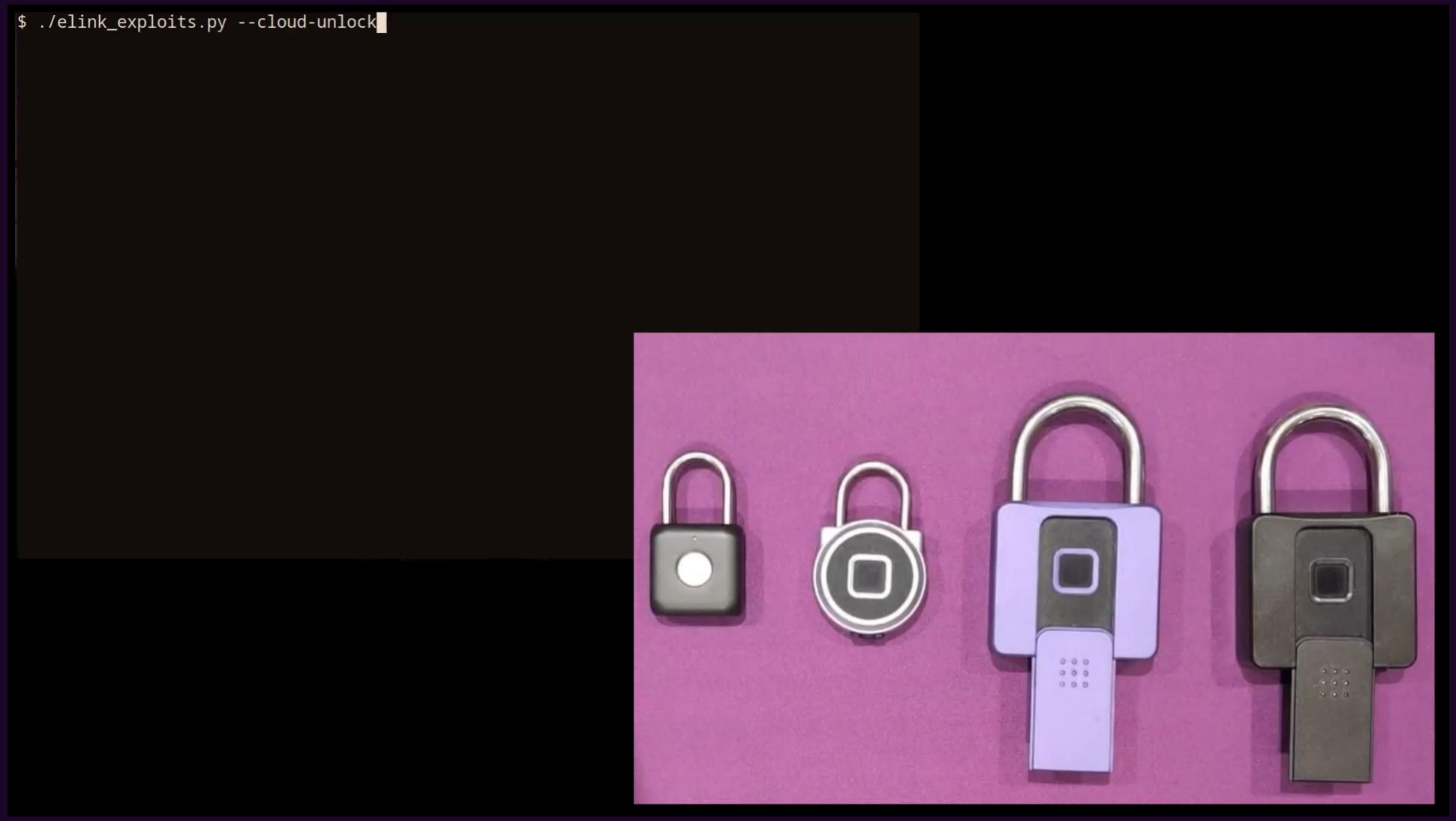
Proof of concept

1. Look for any locks currently advertising – get their MAC addresses
2. Request lock info (passkey) from API
3. Connect to the lock, get a temporary token
4. Politely ask the lock to open
5. ?????
6. Plunder!

Demo!

Live demo disaster in 3... 2... 1...

Backup demo!



Other cool and normal endpoints

- This app does a lot of things
- Too many things
- Query any user, enumerate their locks
- Persistent location of mobile unlocks! :D


```
{
        "mac":"A4:C1:38:21:95:CF",
        "time":"2023-11-26 22:01:35",
        "timeUTC":"2023-11-26 14:01:35",
        "unlockType":3,
        "userName":"testacct",
        "nickname":"testacct",
        "way":2,
        "latitude":"51.50208710000000000",
        "longitude":"-0.07538620000000000",
        [...]
}
```


Summary of issues

API vulnerabilities

- Lack of authentication/authorisation – critically sensitive information + ability to change settings
- Other very basic problems

Hardcoded encryption material

- Essentially ineffective – except as a small hurdle for the reverse-engineer

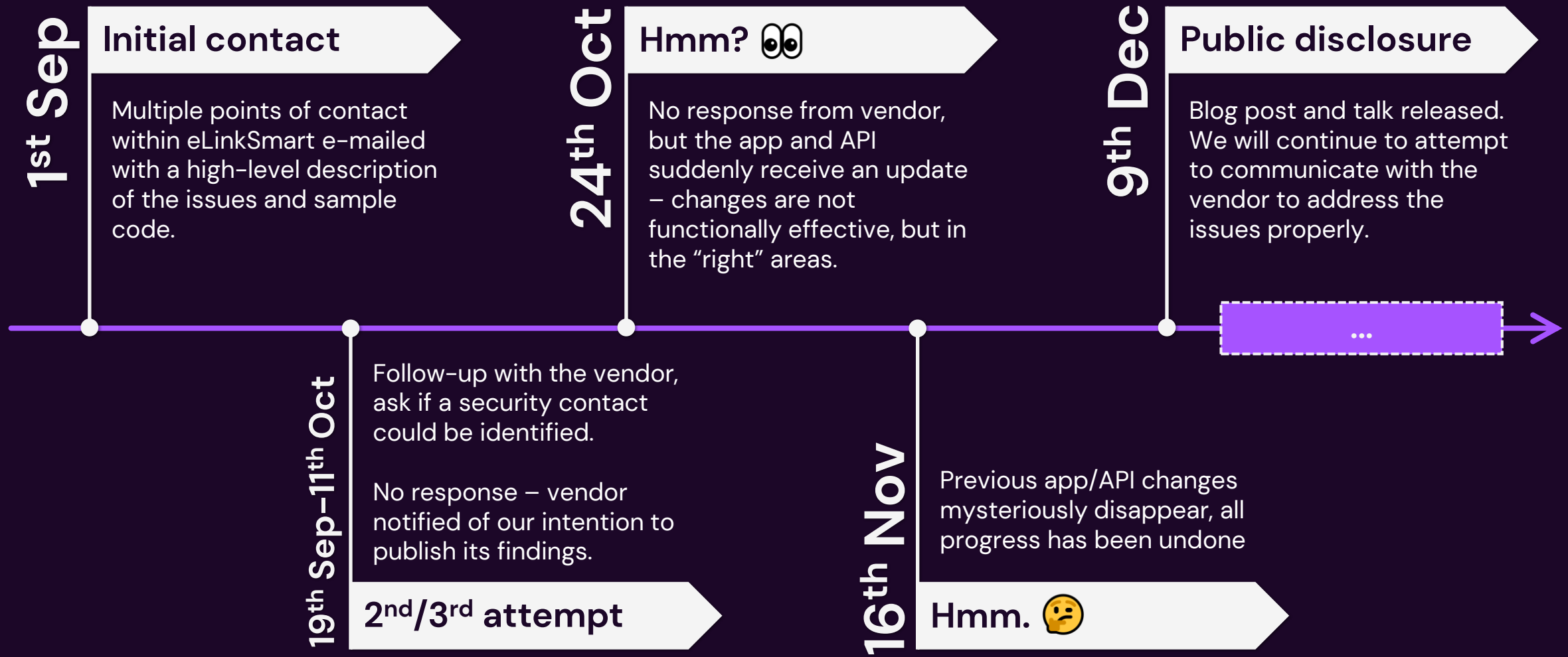
Static passkeys

- Endlessly reusable
- No way for victim to prevent future attacks

Mitigations

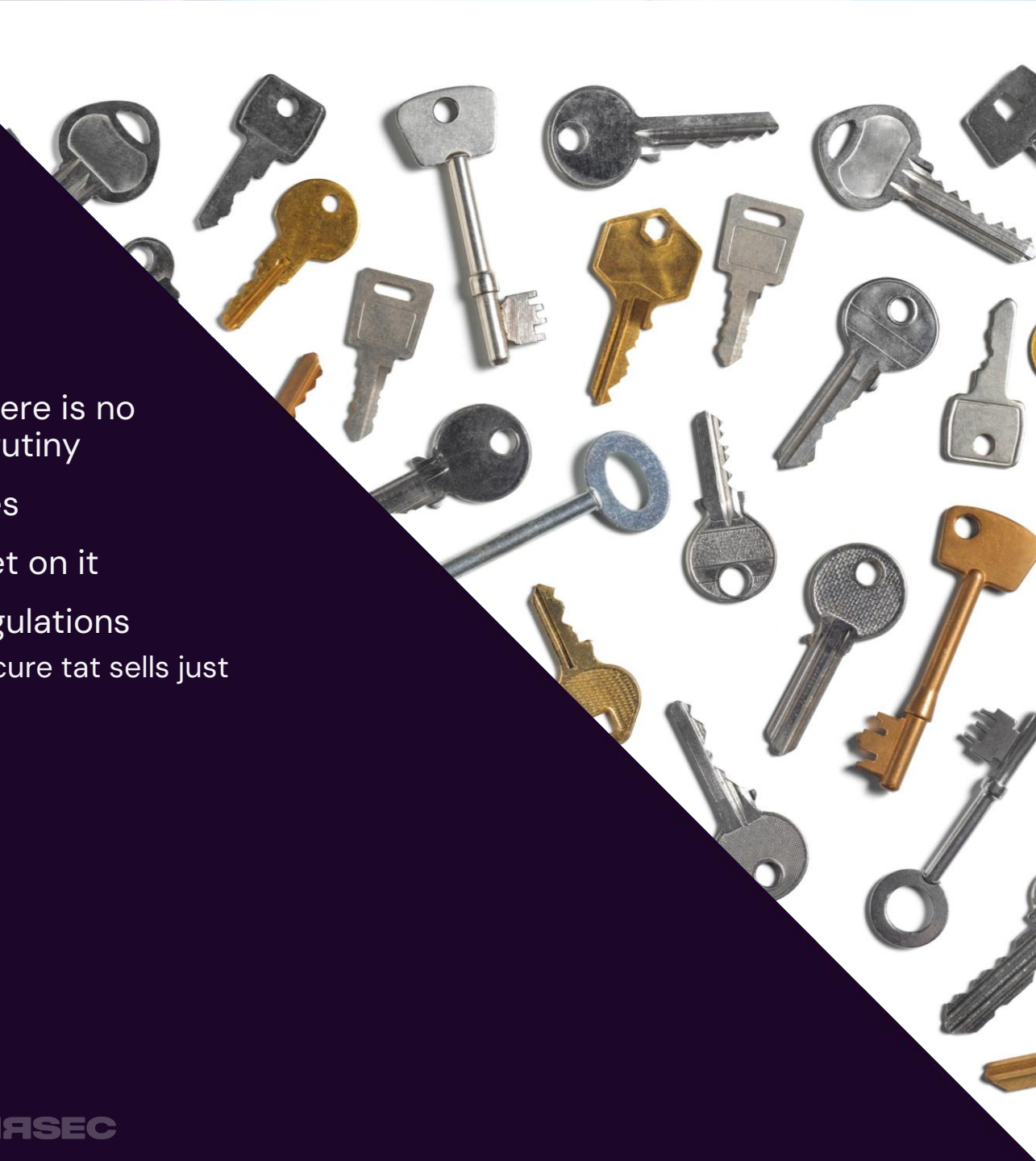
- Could switch locks into fingerprint-only mode
  - Still low-security, but that was a given from the get-go
  - Lose some functionality, but no more random unlocks
- Could gut the battery/USB port out of the keyed lock and use it as an overpriced but otherwise acceptable dumb lock
- Anything else would require co-operation from the manufacturer

Communications with eLinkSmart (2023)



Conclusions (2023)

- Don’t buy this crap (unless it’s for fun)
- Maybe this vendor will fix things eventually, but currently there is no assurance that any smart padlock will stand up to basic scrutiny
- Other cheap brands are known to have near-identical issues
- Would expensive brands be better? Maybe, but wouldn’t bet on it
- Things probably won’t get better without standards and regulations
  - And it’s not in the marketplaces’ interest to have those – insecure tat sells just as well
- You have the tools to look into similar issues!
  - More public scrutiny is always good
  - The skillset is not too hard to develop, but still quite rare
  - Go hack some locks and other IoT devices!



But wait, there's more!

Fast-forward to 2025...

New lock acquired!

- Same-same, but different
- Anweller/eLinkSmart P12BC
- No fingerprint scanner, but we get other fun bits!
  - PIN code entry
    - PIN is user-settable!
  - Temporary PINs
    - Auto-generated…?
  - RFID card unlock
- And of course there's still Bluetooth unlock
- …so that's gotta work the same, right?

Wrong!

- The approach of “let’s just run the script and hope for the best” **doesn’t** work out!
  - But it still works on the old locks…
- The API clearly hasn’t changed – we can still get the PIN from the cloud, still completely unauthenticated, and it works on the physical pad.
- So: why doesn’t BLE unlock work?
- Clearly, **clearly**, they’ve changed something about the protocol.
- So now we know that we must go **back… to the reversing!**

So, what's changed?

- We told you before that log statements made reversing easier
- They removed… some of them!
  - (seemingly just the ones we directly referenced in the blog post)
  - But, y'know, they left at least 1 statement in the most interesting class, just to keep it easy to find.

```
ia.f.b("BleProtocolUtils--parsePwdList-E->" + e10);
```

- OK, so we can still easily find BleProtocolUtils, and quickly look for changes.
- Huh… nothing has changed, but the password it's using is not the `admin_password`
- Instead, it's another field of the same API – `password`
- But… that's now usually empty in our testing, so what gives?

Example response from before

```
HTTP/1.1 200 OK
[...]
X-Powered-By: PHP/7.2.24
Content-Length: 197


{
        "state":"success",
        "type":0,
        "desc":"接口操作成功",
        "data":
        {
                "name":"lock",
                "mac":"A4:C1:38:21:95:CF",
                "isBind":1,
                "password":"",
                "reset":1,
                "lock_status":1,
                "admin_password":"496250",
                "apply_mode":0
        }
}
```

But if we grab one from Burp…

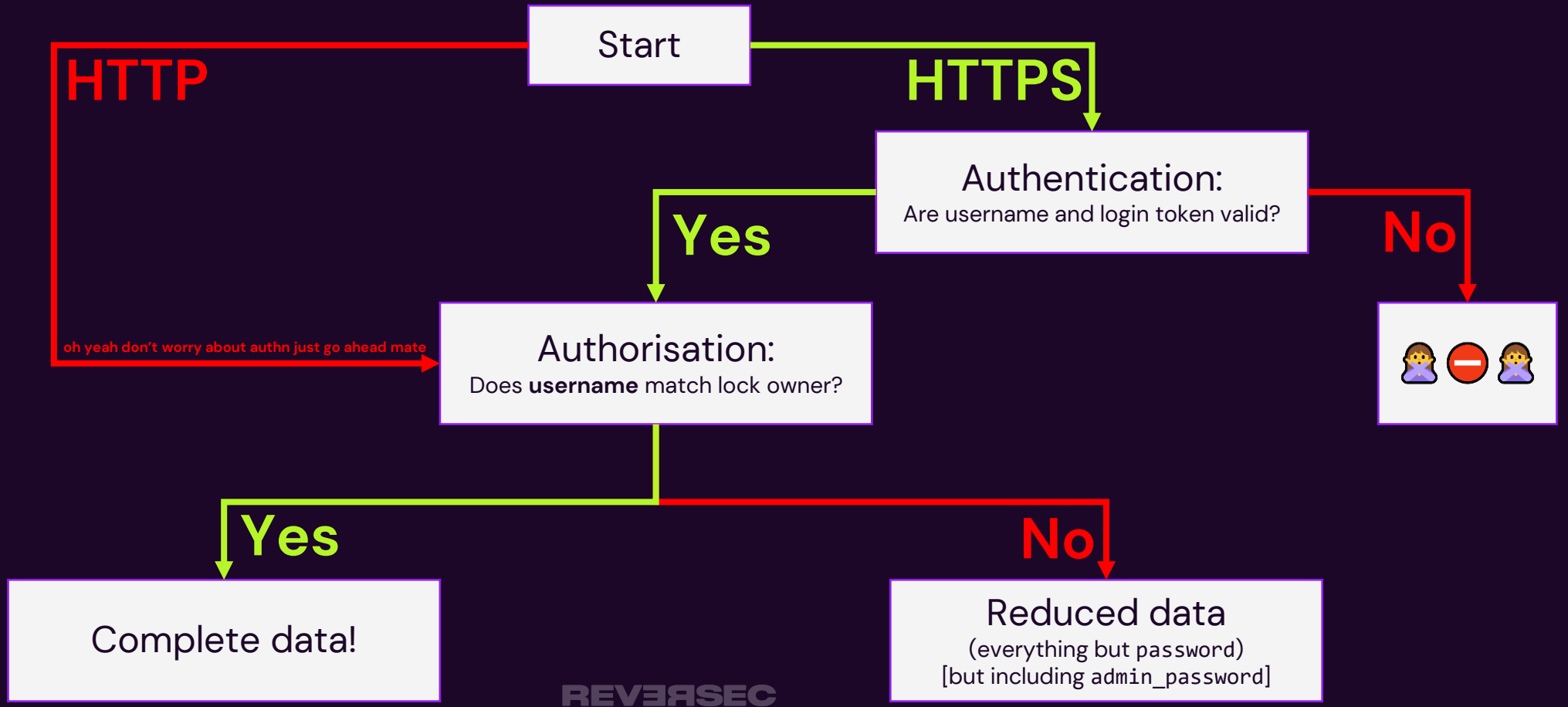
```
HTTP/1.1 200 OK
[...]
X-Powered-By: PHP/7.2.24
Content-Length: 197


{
        "state":"success",
        "type":0,
        "desc":"接口操作成功",
        "data":
        {
                "name":"lock",
                "mac":"A4:C1:38:21:95:CF",
                "isBind":1,
                "password":“746284",
                "reset":1,
                "lock_status":1,
                "admin_password":"496250",
                "apply_mode":0
        }
}
```

Easy to make wrong assumptions!

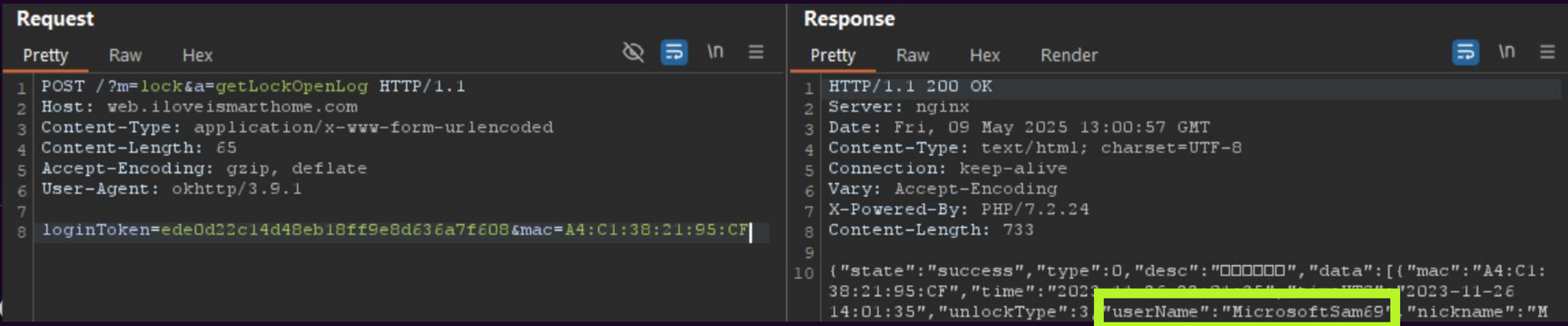
Back in 2023: the API had no authentication/authorisation (we could always fetch the sensitive info we wanted!)

Now: the API exhibits subtly different behaviours depending on how you talk to it.



So, a little more complicated…

- We don't need auth, but we **do** need a username…
- And, of course, the API happily discloses lots of information…
- You can get a lock list for a numeric user ID – that'll reveal the admin username
  - but then we're just trading one piece of info we don't have for another
- The unlock log *might* give us what we need
  - but what if the user never BLE-unlocked the lock?

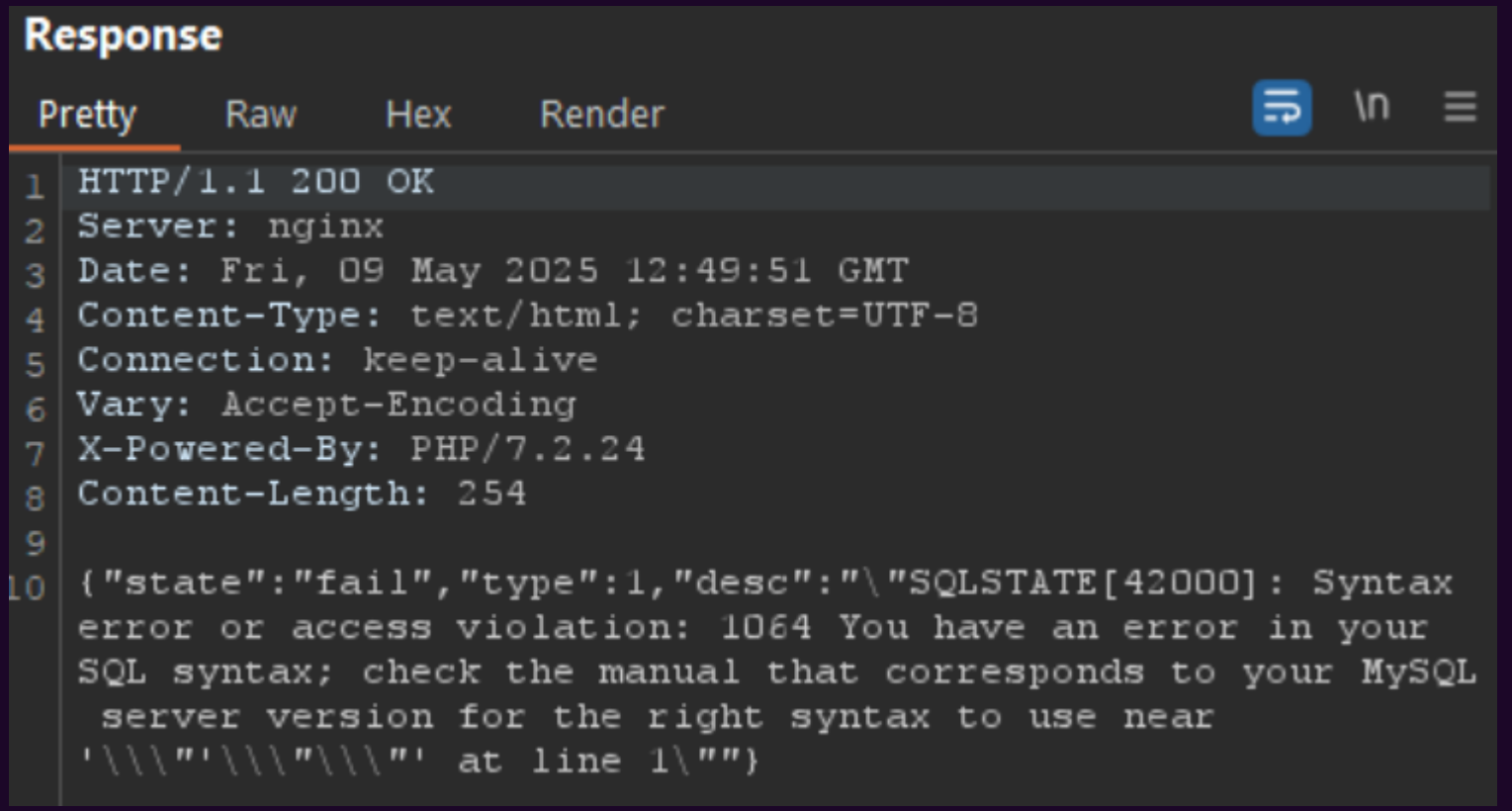


Dropping the other shoe

- We may have omitted a ***tiny*** little detail up until now.

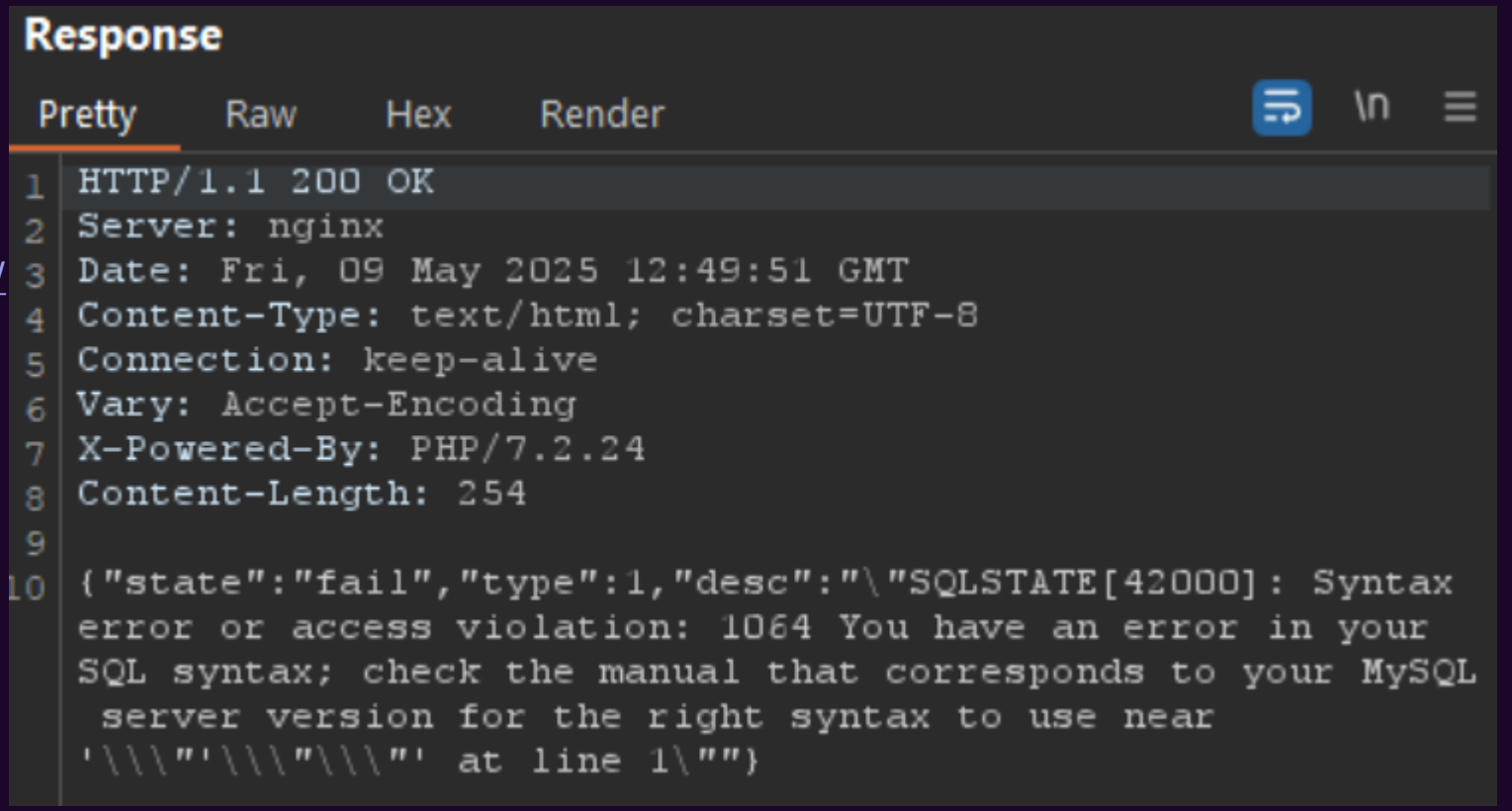
Dropping the other shoe

- We may have omitted a ***tiny*** little detail up until now.
- yea



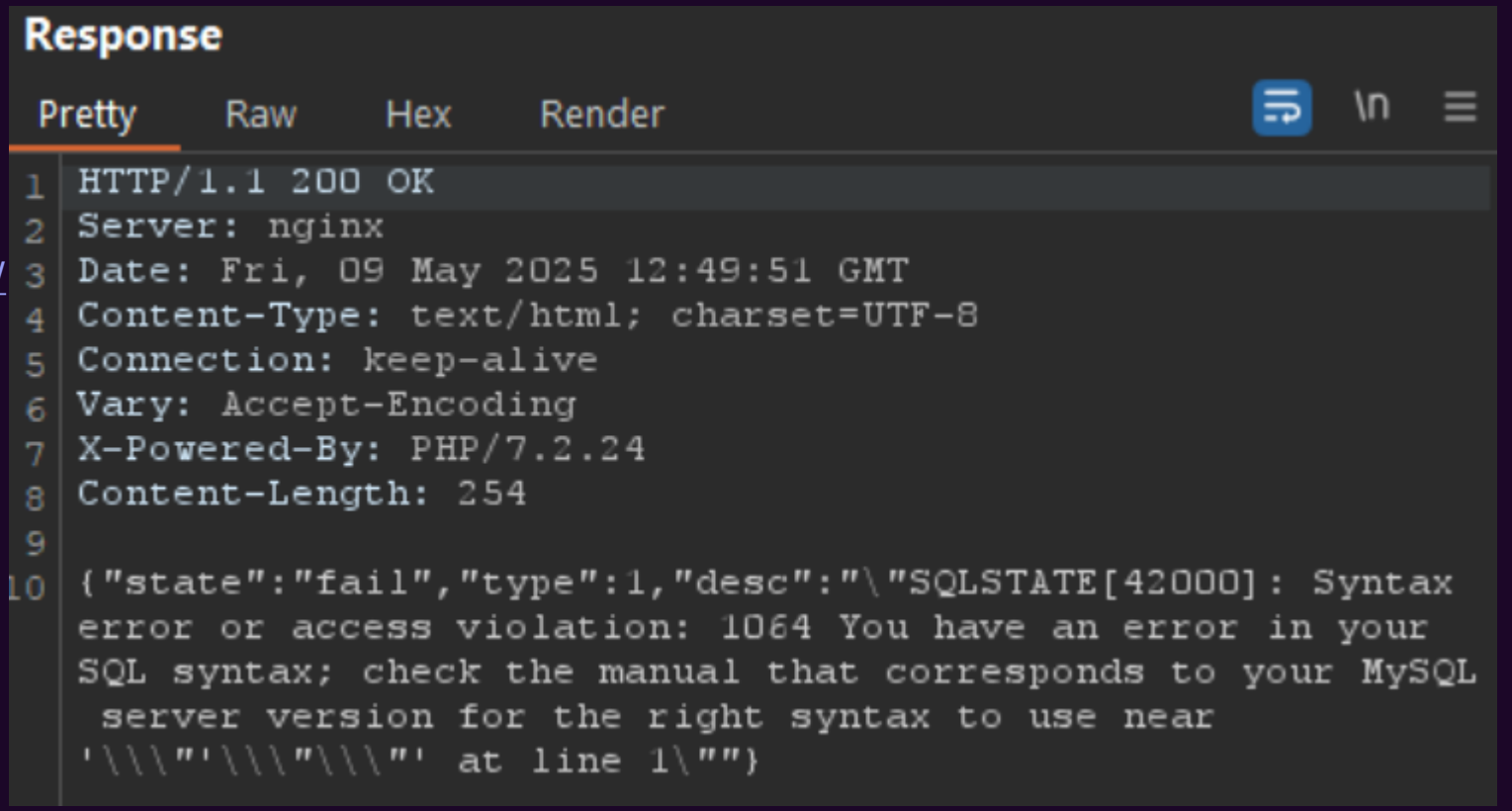
Dropping the other shoe

- We may have omitted a ***tiny*** little detail up until now.
- yea
- They know – we told them
- As did others before us!
  - Shout out to @nv1t / @nv1t@chaos.social
  - https://nv1t.github.io/blog/the-weired-ble-lock/



Dropping the other shoe

- We may have omitted a ***tiny*** little detail up until now.
- yea
- They know – we told them
- As did others before us!
  - Shout out to @nv1t / @nv1t@chaos.social
  - https://nv1t.github.io/blog/the-weired-ble-lock/
- 2.5 years later, it's still unfixed
- Minutes with sqlmap -> a script that always gets the password :)



Conclusions 2.0

- *Don't buy/use this crap unless it's **purely** for fun*
- Don't think these are getting fixed anytime soon…
- Even when they "improved" things, the locks can still be popped in 100 different ways
  - Walk up to the PIN lock and just type in the passcode
  - Get the passcode by supplying the right username, which 99 times out of 100 you'll get from the unlock log
  - Abuse SQLi
  - Probably other ways we haven't thought about!
- Sometimes, the oldest trick in the book is all you need
- You should go mess about with a cheap IoT device!

REVERSEC

Questions?

Blog posts & misc.

- Our 2023 post: https://labs.reversec.com/posts/2024/02/multiple-vulnerabilities-in-elinksmart-padlocks
- Follow-up 2025 post: coming Soon™
- @nv1t's post: https://nv1t.github.io/blog/the-weired-ble-lock/
- More about us: https://www.reversec.com/
  - (or talk to us after the talk?)

REVERSEC