hexway

Apple bleee. Everyone knows What Happens on Your iPhone

In this article, we will outline our research process, from our initial ideas to first POCs. Technical specialists may find this interesting.

What if…

Let's take a look at the BLE traffic. For this purpose, we've slightly modified the scripts from py-bluetooth-utils repository'

Unlock the phone and run the `BLE` sniffer.

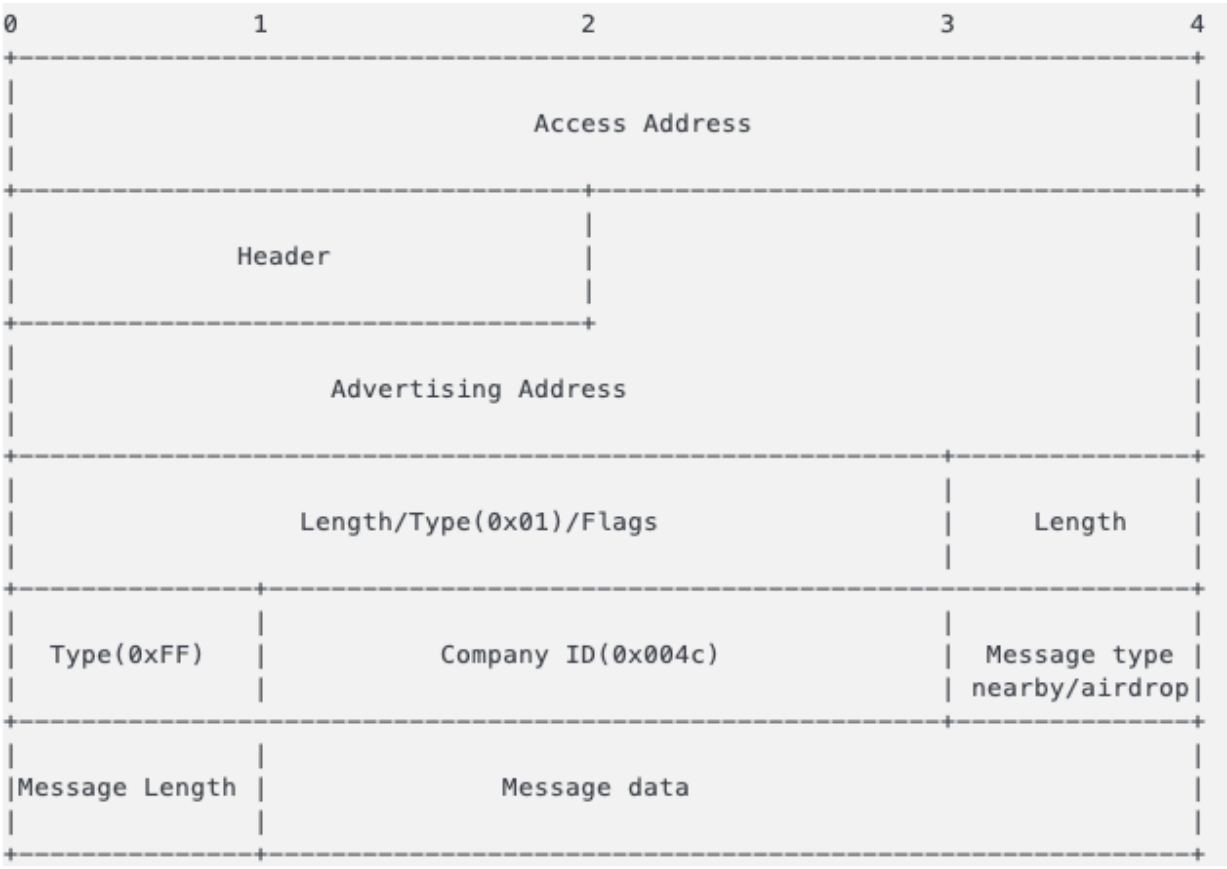
```
python ble_adv_search.py -m 54:69:F1:23:2B:47
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
...
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
...
```

Turn the phone off.

```
...
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c0010050b1c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c001005031c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c001005031c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c001005031c0fc556
[54:69:F1:23:2B:47] 0e02011a0aff4c001005031c0fc556
...
```

We can see that only one byte reflects the change in screen status. Apple uses `ADV_IND` messages to send out current status data.

That's the structure of a typical advertise data packet:



Turning the phone on and off, navigating the menu, changing settings and running different apps (phone, calendar, photos, settings), we've identified the fields responsible for the Wi-Fi and buffer status and several types of BLE messages.

```
0x05 - Airdrop
0x07 - Airpods
0x10 - Nearby
0x0b - Watch Connection
0x0c - Handoff
0x0d - Wi-Fi Settings
0x0e - Hotspot
0x0f - Wi-Fi Join Network
```

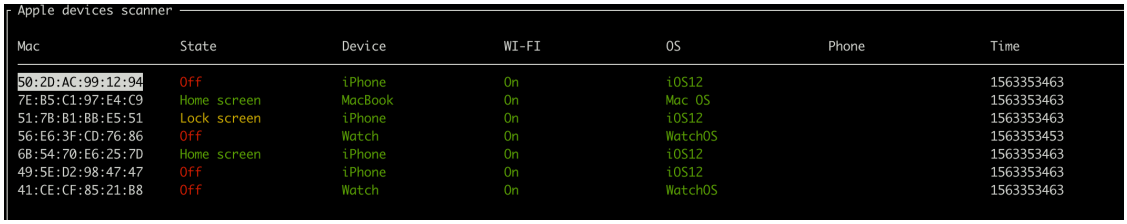
Nearby

That's a Nearby message:

where status can be:

```
0x0b - Home screen
0x1c - Home screen
0x1b - Home screen
0x11 - Home screen
0x03 - Off
0x18 - Off
0x09 - Off
0x13 - Off
0x0a - Off
0x1a - Off
0x01 - Off
0x07 - Lock screen
0x17 - Lock screen
0x0e - Calling
0x5b - Home screen
0x5a - Off
```

That's enough to write a simple packet analyzer that allows us to get data from all nearby Apple devices in real-time.

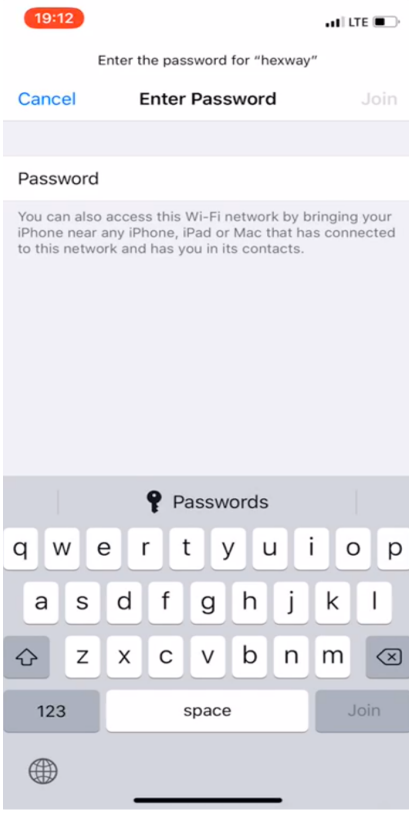


Video Demo:
https://www.youtube.com/watch?v=Bi602yAIBAw

To analyze the `BLE` packets further, we chose to use Adafruit Bluefruit LE Sniffer that helps analyze the `Wireshark` `BLE` traffic.

Wi-Fi password sharing

Then, we analized how users are identified when two devices interact. This process is used for conencting to Wi-Fi.



While trying to connect to a network, the device sends the following `BLE` packet:

`c03080fc5563125c9d087555a77e3e2005f10020b0c`

Trying to connect to various `SSID` on different devices (yes, we could've just reversed `sharingd` in IDA/radare/gydra) we found out that this message has the following format:

```
0        1       2                       5                       8                       12                      15                      18
+--------+-------+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
|        |       |                       |                       |                       |                       |                       |
| flags  | type  |     auth tag          |     sha(appleID)      |   sha(phone_nbr)      |  sha(email)           |   sha(SSID)           |
|        | (0x08)|                       |                       |                       |                       |                       |
+--------+-------+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
```

As you can see, the device sends the first 3 bytes of the `SHA256` hash of phone number/email/appleID `sha256(phone_number)[0:3]`

Probably, devices hash all contacts and then compare them to the values from `BLE` messages, and in case there is a match offer to share the Wi-Fi password (if they have it, of course). We'll look into this process in more detail in the following articles. What we tried to do here is to recover a phone number from those 3 bytes of the hash.

First, we have to understand phone number formats. `E.164` is a recommendation describing various formats.

- A phone number can have a maximum of 15 digits
- And it can be divided into:

1. Country code (1-3 digits)
2. National destination code (NDC)
3. Subscriber number

Of course, formats vary with country, but the idea is the same. So, we can calculate the values of SHA256 for the numbers of a particular city.

```
<sha256[0:3]>:<phone_number>
```

We made a table of phone numbers for a city with a population of about 5 000 000. Considering the large number of subscribers, collisions are inevitable. In average, we have a collision of 10-15 numbers for 3 bytes of hash.

We made an API for quick requests to the table to get a number from a hash:

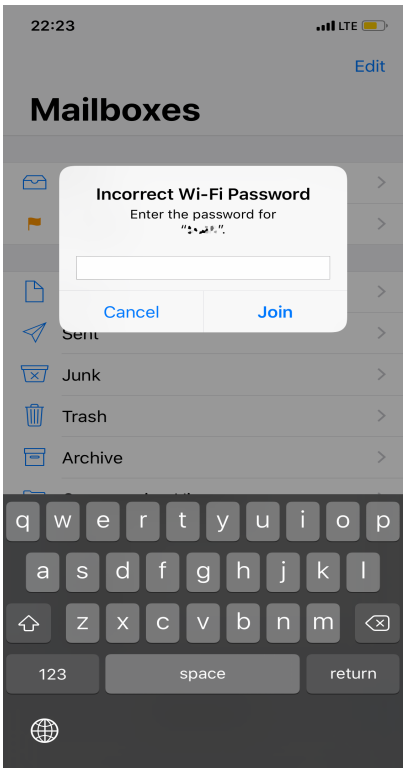
There are two approaches to ensuring the accuracy of identification:

1. HLR (Home Location Register) Lookup that allows identifying inactive subscribers and subscribers from other regions

2. A number must be associated with an AppleID, so we can identify valid numbers by checking if iMessage is available for a certain number (we'll talk about this approach in more details in the future articles).

Combining these two approaches, we can accurately identify a phone number in almost 100% of cases.

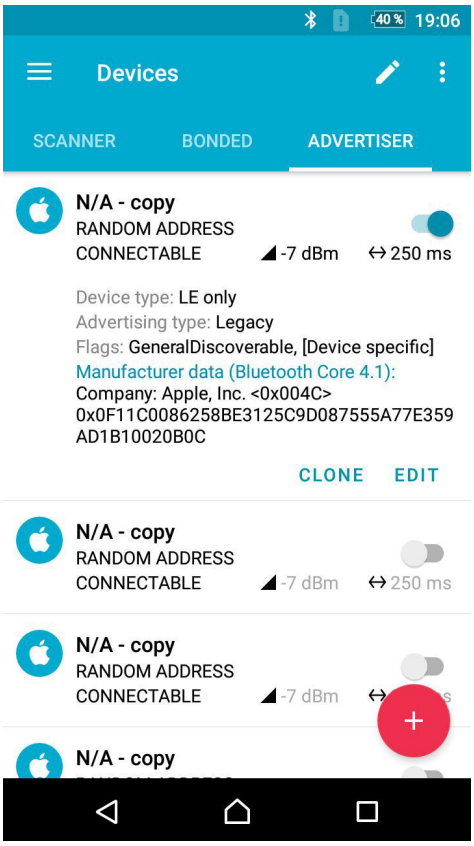
Can we activate the Wi-Fi password sharing popup on a device? That's an open issue.



Thus, we have a script that identifies the users connected to Wi-Fi.

https://www.youtube.com/watch?v=kTtNX5Tmk3Q

Moreover, we can send BLE requests for a Wi-Fi password hoping that the victim will provide us, for example, with a corporate network password. We'll talk about this vector more in our other articles as well. By the way, you can use an Android app (like nRF Connect) to clone and repeat different BLE messages.

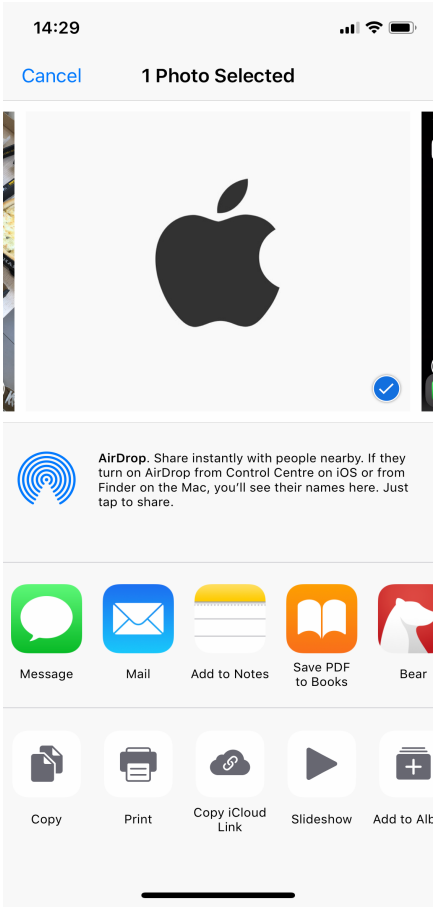


Airdrop

Let's see if this mechanism of identifying users for the purpose of sharing Wi-Fi passwords is universal. Apple Airdrop is one of the points we can investigate, as it has 3 privacy options:

1. Receiving Off
2. Contacts Only
3. Everyone

So, how do devices identify each other?

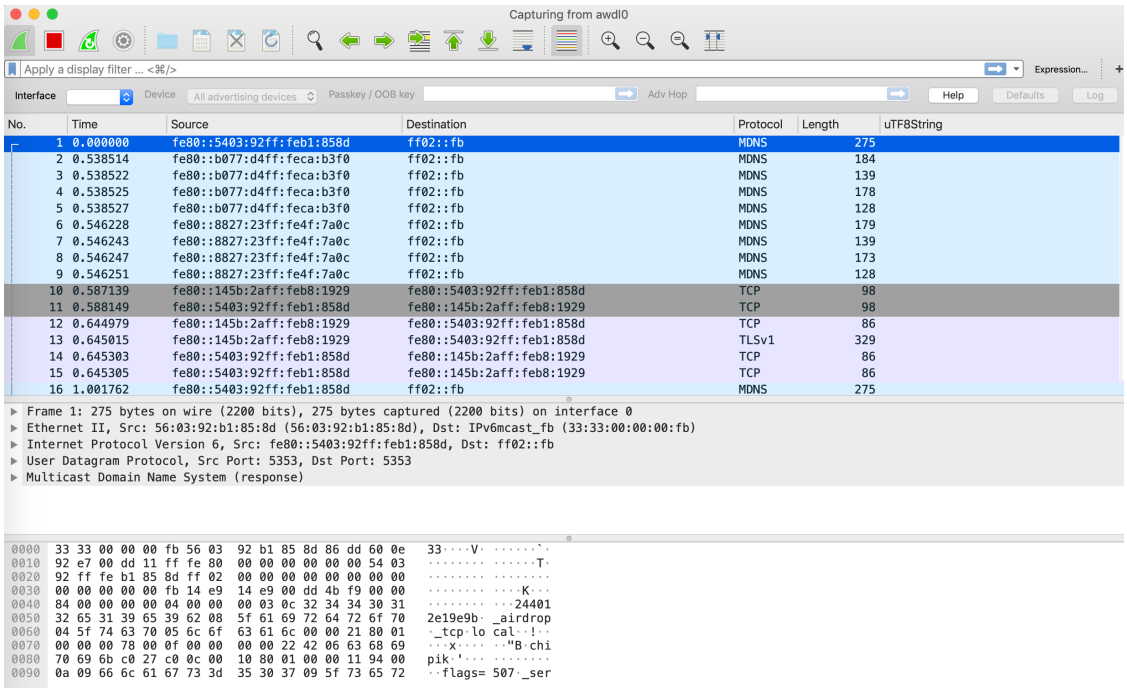


We run Airdrop and follow the `BLE` sniffer:

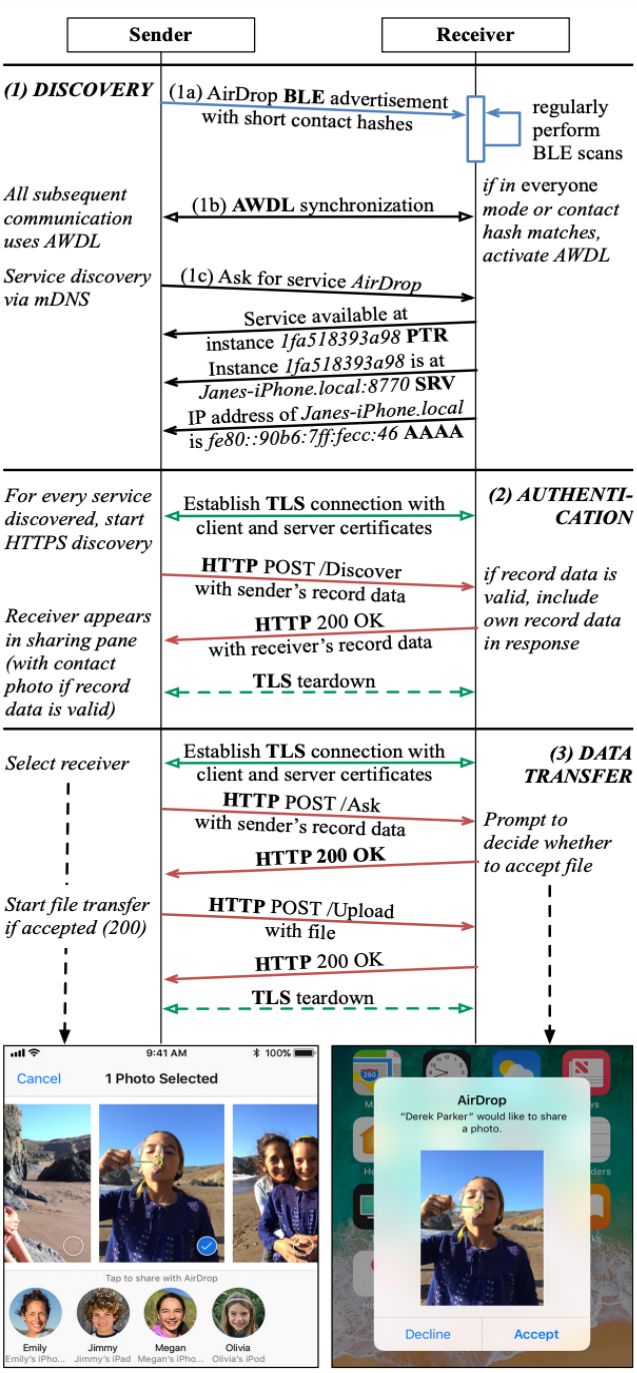
`00000000000000000112341234123400`

```
0                                       8       9               11                13                15                17      18
+---------------------------------------+-------+---------------+-----------------+-----------------+-----------------+-------+
|                                       |       |               |                 |                 |                 |       |
|          zeros                        |st(0x01)| sha(AppleID)  | sha(phone)      |  sha(email)     |   sha(email2)   |  zero |
|                                       |       |               |                 |                 |                 |       |
+---------------------------------------+-------+---------------+-----------------+-----------------+-----------------+-------+
```

As you can see, in this case, devices sends only 2 bytes of the SHA256 hash, which is enough to identify the phone number. Traffic analysis showed that BLE is only used to initiate AirDrop transfer. The transfer itself happens via Wi-Fi using the `AWDL` technology that establishes a peer2peer connection between the server (receiver) and the client (sender). To analyze the AWDL packets, we can use Wireshark and the `awdl0` interface available on all devices.



As you see, after sending a few `MDNS` packets, the devices exchange their certificates and then use the secure `TLS` connection for data transfer. We were about to begin the reverse engineering of the `sharingd` app, which is responsible for AirDrop, when people from Technische Universität Darmstadt released a white papper A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link about AirDrop (sadface.png). This white paper beautifully describes the functioning of AirDrop. Now, we'll take a brief look at the AirDrop protocol workflow:



During authentication, the device sends its identification data (sender's record data), that contains the **full** SHA256 hash of the phone number. Thus, if we answer all Airdrop BLE requests, we'll get the sender's contact data, including phone number hash.

We've slightly modified the opendrop utility to do that.

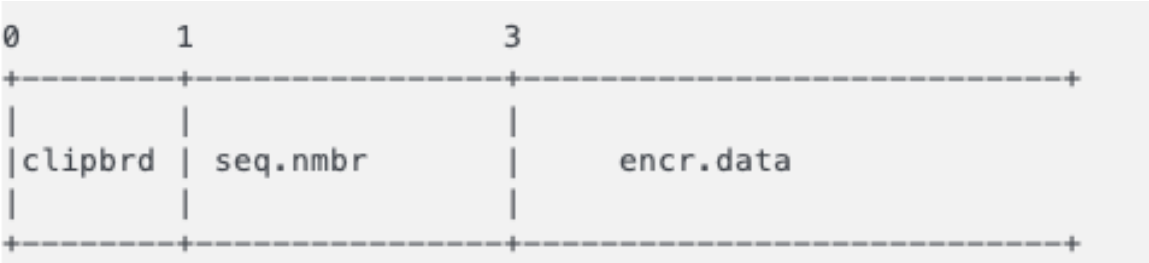
Here are the results:

https://www.youtube.com/watch?v=mREIeH_s3z8

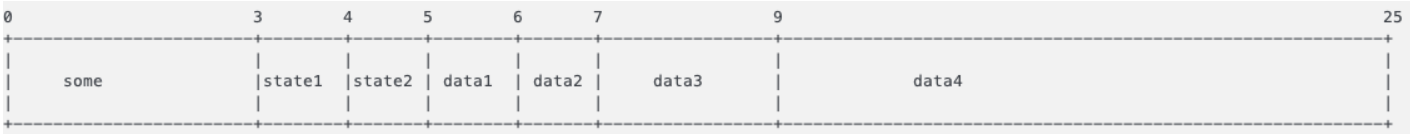
Others

Below, you can see various BLE message formats that we encountered during our research.

Handoff



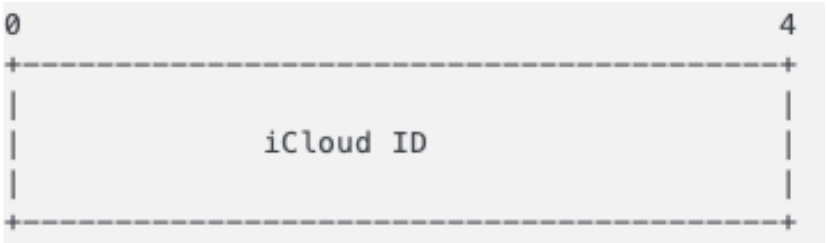
Airpods



By sending this message, we can make Apple devices display AirPods inforamtion as if they were connected. Just watch this funny video:

https://www.youtube.com/watch?v=HoSuLUtrkXo

Wi-Fi settings



Hotspot

PoCs

You can find all scripts in our GitHub repository: Apple bleee

Links

- https://github.com/hexway/apple_bleee
- https://hexway.io/blog/apple-bleee/
- https://arxiv.org/pdf/1904.10600.pdf
- https://www.usenix.org/system/files/sec19fall_stute_prepub.pdf
- https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf

Contact us:

contact@hexway.io

https://hexway.io