IT SECURITY KNOW-HOW

Matthias Deeg, Gerhard Klostermeier

On the Security of RFID-based TOTP Hardware Tokens

Hacking NFC-enabled Time-Based One-Time Password Hardware Tokens

June 2021

1 Introduction

Time-based one-time passwords (TOTP) have been around for several years now and became more and more widespread as authentication factor in multi-factor authentication (MFA) methods. Protecting user accounts via two-factor authentication (2FA) using a static password and a TOTP is considered a good idea from a security standpoint and a best practice that can prevent different kinds of attacks.

For generating a one-time password, the algorithms described in RFC 4226 [1] titled “HOTP: An HMAC-Based One-Time Password Algorithm” and RFC 6238 [2] titled “TOTP: Time-Based One-Time Password Algorithm” are relevant.

A very popular TOTP generator is the mobile app Google Authenticator [3]. And for implementing TOTPs in software products, a variety of software libraries is available for different programming languages.

The TOTP example in Python shown in Listing 1.1 uses the library PyOTP [4] and illustrates all required configuration parameters for a TOTP:

1. a cryptographic hash function (digest function)
2. a shared secret key (seed value)
3. the length of the generated TOTP
4. the used time interval in seconds (time step between TOTPs)

```python
#!/usr/bin/env python3
import hashlib
import pyotp
import time

# secret key (seed)
SECRET_KEY = "base32topsecret7"

# initialize the TOTP generator with a specific configuration
totp = pyotp.TOTP(SECRET_KEY, digest=hashlib.sha256, digits=6, interval=30)

# generate three TOTPs with a time interval of 30 seconds
for i in range(3):
    password = totp.now()
    print(password)
    time.sleep(30)
```

Listing 1.1: TOTP example in Python

The following output exemplarily shows an output of this example with three sequently generated OTPs with a time interval of 30 seconds.

```
$ python totp_test.py
640660
808281
945719
```

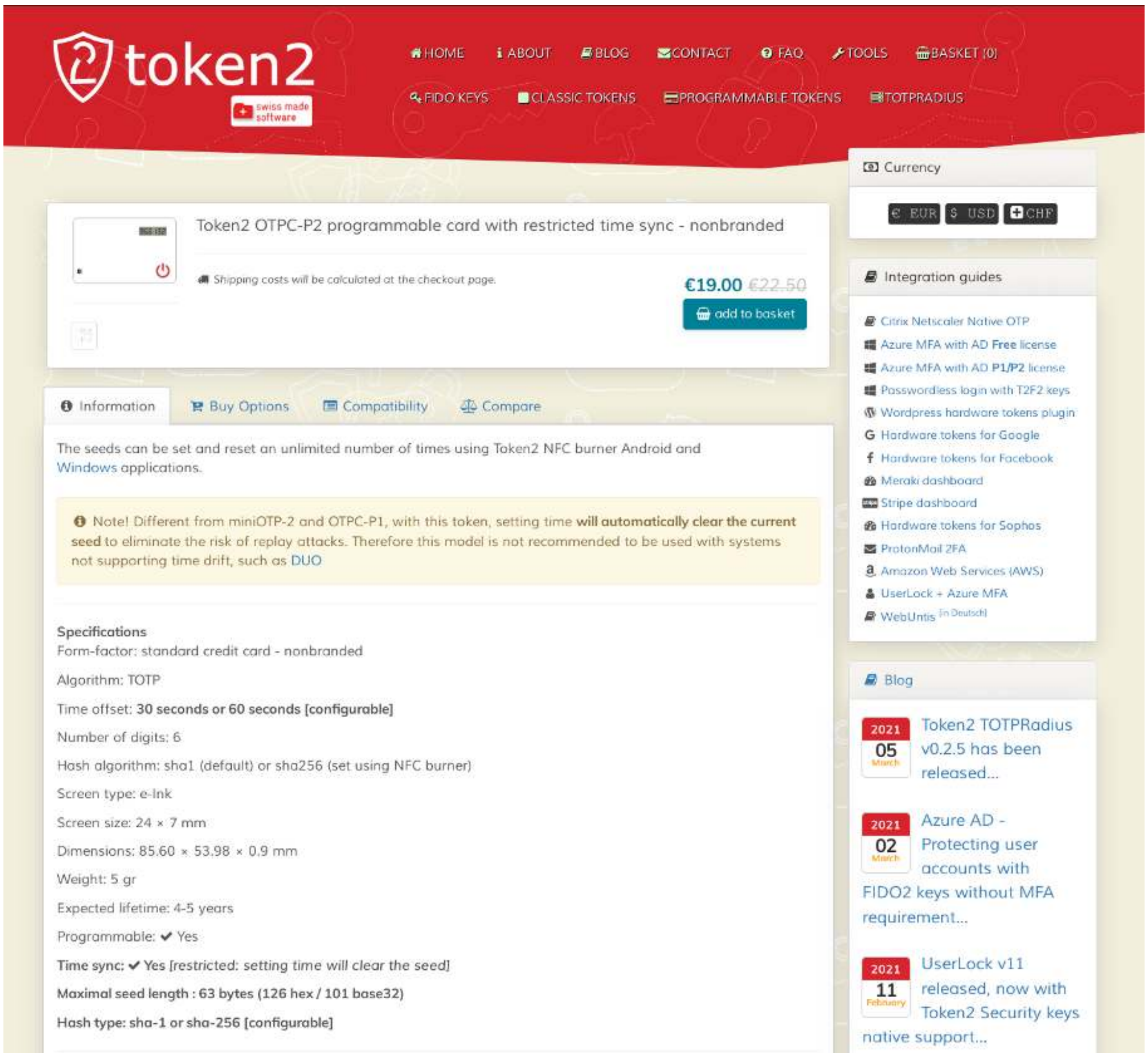
Besides software-based TOTP generators like the mobile app Google Authenticator, there are also different kinds of hardware TOTP tokens.

During a research project, we had a closer look at two such RFID-based tokens which support near-field communication (NFC).

The result of our case studies for the Token2 OTPC-P2 and the Protectimus SLIM NFC are provided in Sections 1 and 2.10.

2 Case Study I: Token2 OTPC-P2

The NFC-based card "OTPC-P2" is distributed by Token2, a Swiss company based in Geneva [6]. It is designed as a token for TOTP-based two-factor authentication. The form factor is identical to a typical credit card. The features of the token are specified by the distributor Token2 as shown in Figure 1.



Token2 OTPC-P2 programmable card with restricted time sync - nonbranded

Shipping costs will be calculated at the checkout page.

€19.00 ~~€22.50~~

The seeds can be set and reset an unlimited number of times using Token2 NFC burner Android and Windows applications.

Note! Different from miniOTP-2 and OTPC-P1, with this token, setting time **will automatically clear the current seed** to eliminate the risk of replay attacks. Therefore this model is not recommended to be used with systems not supporting time drift, such as DUO

Specifications
Form-factor: standard credit card - nonbranded

Algorithm: TOTP

Time offset: **30 seconds or 60 seconds [configurable]**

Number of digits: 6

Hash algorithm: sha1 (default) or sha256 (set using NFC burner)

Screen type: e-Ink

Screen size: 24 × 7 mm

Dimensions: 85.60 × 53.98 × 0.9 mm

Weight: 5 gr

Expected lifetime: 4-5 years

Programmable: ✔ Yes

Time sync: ✔ Yes *[restricted: setting time will clear the seed]*

Maximal seed length : 63 bytes (126 hex / 101 base32)

Hash type: sha-1 or sha-256 [configurable]

Citrix Netscaler Native OTP
Azure MFA with AD **Free** license
Azure MFA with AD **P1/P2** license
Passwordless login with T2F2 keys
Wordpress hardware tokens plugin
Hardware tokens for Google
Hardware tokens for Facebook
Meraki dashboard
Stripe dashboard
Hardware tokens for Sophos
ProtonMail 2FA
Amazon Web Services (AWS)
UserLock + Azure MFA
WebUntis (in Deutsch)

2021 05 March Token2 TOTPRadius v0.2.5 has been released...

2021 02 March Azure AD - Protecting user accounts with FIDO2 keys without MFA requirement...

2021 11 February UserLock v11 released, now with Token2 Security keys native support...

Figure 1: Token2 OTPC-P2 product description

There are two key differences when compared to the token of Protectimus (see Section 2.10):

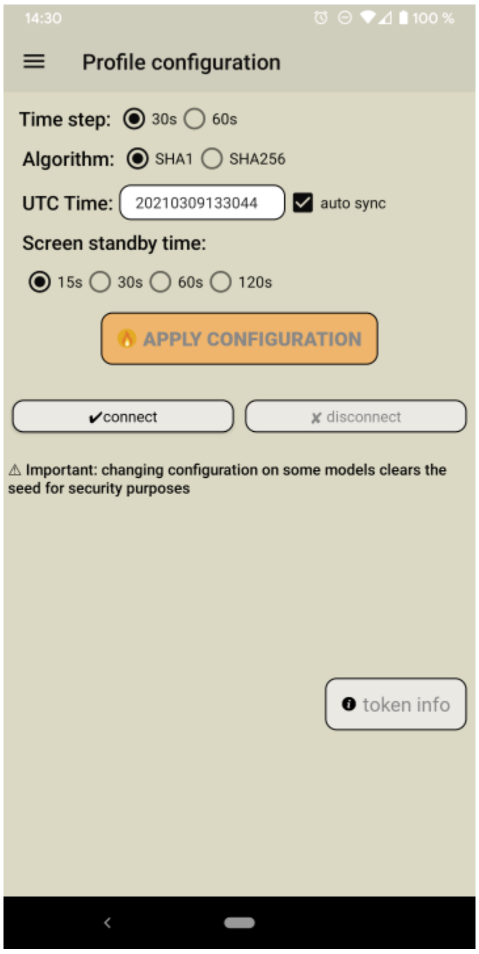
1. The card will wipe the configured secret seed when the time (or other configuration parameters) is updated.
2. The one-time password (OTP) displayed on the e-Ink display cannot be read out via the NFC interface.

The following sections summarize the results of our case study concerning this NFC-based TOTP token. They contain information about the general operation of the token, the journey of reverse engineering some of its functionality, and identified security issues and interesting questions, which could not be answered yet.

2.1 Normal operation

The two-factor token arrives in an unconfigured state. To configure a card, two software solutions are provided by Token2: A Windows application [7] and an Android app [8].

Using the provided software, the token can be configured via NFC. When the token is presented to e.g. an Android phone running the NFC Burner 2 app, the serial number and the current time from the card is read out and displayed. If needed, the first step is to edit the general configuration (see Figure 2). The second step is to "burn" the secret seed for the used TOTP HMAC algorithm onto the card (see Figure 3).


Profile configuration

Time step: [✓ 30s] 60s

Algorithm: [✓ SHA1] SHA256

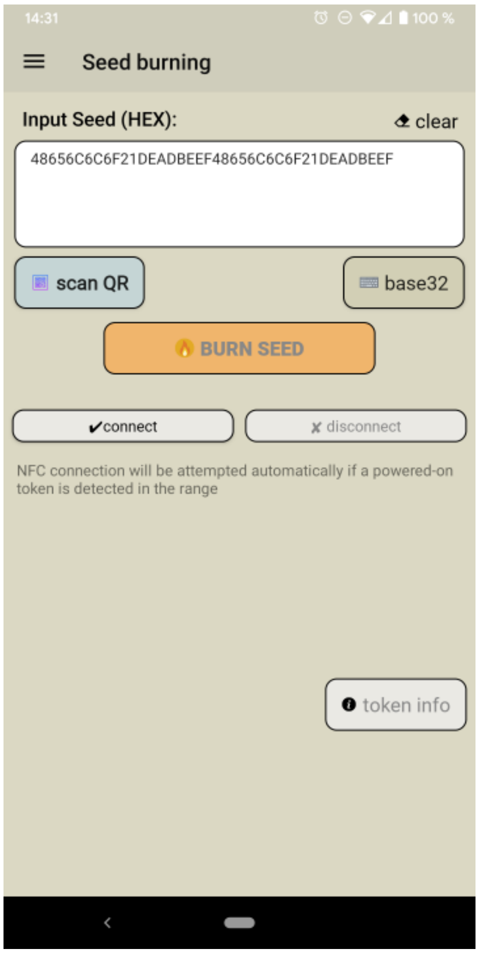
UTC Time: 20210309133044 [✓ auto sync]

Screen standby time:

[✓ 15s] 30s 60s 120s

⚠ Important: changing configuration on some models clears the seed for security purposes

Figure 2: Configuration view of the NFC burner 2 Android app


Seed burning

Input Seed (HEX):

48656C6C6F21DEADBEEF48656C6C6F21DEADBEEF

NFC connection will be attempted automatically if a powered-on token is detected in the range

Figure 3: "Seed burning" view of the NFC burner 2 Android app

Once the configuration and a seed value are set, the token can be used for TOTP-based two-factor authentication. To get a valid one-time password, the user must press the button in the lower right corner of the card. The OTP is then shown on the e-Ink display of the tag. In contrast to the Protectimus card, the OTP cannot be read via the NFC interface. This interface seems to be only for configuration purposes.

2.2 NFC interface

On the NFC interface, the token can be identified as a typical ISO14443-A tag. Additional information imply, that the tag is a Java Card, which means, that ISO 7816-4 Application Protocol Data Units (APDUs) are used to exchange data. [10]

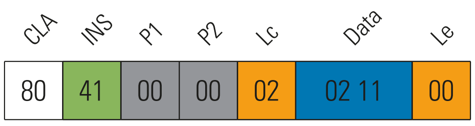


Figure 4: Example: APDU to query the time and serial number of the card

Field name	Length (bytes)	Description
CLA	1	Instruction class - indicates the type of command, e.g. interindustry or proprietary
INS	1	Instruction code - indicates the specific command, e.g. “write data”
P1-P2	2	Instruction parameters for the command, e.g. offset into file at which to write the data
Lc	0, 1 or 3	Encodes the number (Nc) of bytes of command data to follow • 0 bytes denotes Nc=0 • 1 byte with a value from 1 to 255 denotes Nc with the same length • 3 bytes, the first of which must be 0, denotes Nc in the range 1 to 65 535 (all three bytes may not be zero)
Data	Nc	Nc bytes of data

Continues next page ...

Field name	Length (bytes)	Description
Le	0, 1, 2 or 3	Encodes the maximum number (Ne) of response bytes expected • 0 bytes denotes Ne=0 • 1 byte in the range 1 to 255 denotes that value of Ne, or 0 denotes Ne=256 • 2 bytes (if extended Lc was present in the command) in the range 1 to 65 535 denotes Ne of that value, or two zero bytes denotes 65 536 • 3 bytes (if Lc was not present in the command), the first of which must be 0, denote Ne in the same way as two-byte Le

Table 1: Structure of APDU command

Field name	Length (bytes)	Description
Response Data	Nr (at most Ne)	Response data
SW1-SW2	2	Command processing status, e.g. 0x9000 indicates success

Table 2: Structure of APDU response

Byte 0 of a double sized UID (cascade level 2, 7 bytes) should always contain a manufacturer code. [9] In the case of the Token2 OTPC-P2 token, this byte is set to `0x1D`, which is the code for Shanghai Fudan Microelectronics Group Company Ltd. from China. The UID is not random and can therefore be used for tracking.

```
[usb] pm3 --> hf search
    Searching for ISO14443-A tag...
[+]  UID: 1D 01 A8 01 58 34 78
[+] ATQA: 00 44
[+]  SAK: 20 [1]
[+] MANUFACTURER: Shanghai Fudan Microelectronics Co. Ltd. P.R. China
[+]    JCOP 31/41
[=] -------------------------- ATS --------------------------
[+] ATS: 05 72 F7 A6 02 [ 9d 00 ]
[=]      05...............  TL    length is 5 bytes
[=]         72............  T0    TA1 is present, TB1 is present, TC1 is present, FSCI ⤶
    is 2 (FSC = 32)
[=]            F7.........  TA1   different divisors are NOT supported, DR: [2, 4, 8], ⤶
    DS: [2, 4, 8]
[=]               A6......  TB1   SFGI = 6 (SFGT = 262144/fc), FWI = 10 (FWT = 4194304/⤶
    fc)
[=]                  02...  TC1   NAD is NOT supported, CID is supported
[#] Auth error


[+] Valid ISO14443-A tag found
```

Listing 2.1: Tag detection on the Proxmark3

The NFC interface of the token has two different states:

- Restricted: If the button on the tag has not been pressed, the card can still be discovered by tools like the Proxmark3 (see Listing 2.1). Android devices do not discover the card in that state, because sending and receiving APDUs fails.
- Activated: If the button is pressed, the tag gets activated. It will show a OTP on the e-Ink display and it will respond to APDUs. If there is no NFC traffic, the card will go back to sleep after a short timeout. However, if there is traffic and a field of an NFC reader, the card will stay active.

To better understand how e.g. the NFC Burner 2 Android app by Token2 communicates with the card, the initial communication was sniffed using a Proxmark3. The sniffed data show three stages that are typical when sniffing communication between a tag and an Android device.

1. Tag detection part 1: Android searches for tags in the reader's field and performs the ISO 14443-3 initialization, anti-collision, and selects one tag.
2. Tag detection part 2: Android searches for more information on the tag. For example, it tries to find out if the application identifier (AID) `0xD2760000850101` – the identifier for the NDEF application on MIFARE DESFire tags – is presently using APDU commands.
3. Communication by the app: The actual communication between the tag and the app that is handling the tag (in this case NFC Burner 2 by Token2) takes place.

Listing 2.2 shows the sniffed data with the three stages annotated.

```
[usb] pm3 --> hf 14a sniff

[#] Starting to sniff. Press PM3 Button to stop.
[#] trace len = 1499


[usb] pm3 --> hf list -t 14a
[=] downloading tracelog data from device
[+] Recorded activity (trace len = 1499 bytes)
[=] start = start of start frame end = end of frame. src = source of transfer
[=] ISO14443A - all times are in carrier periods (1/13.56MHz)

   Start |      End | Src |  Daa ( deote prit eror)                             | CRC | Annotation
---------+----------+-----+ -------------------------------------------------------+-----+--------------------

[ Stage 1: Tag detection part 1 ]

       0 |     1056 | Rdr | 26(7)                                                  |     | REQA
    2260 |     4628 | Tag | 44 00                                                  |     |
   12688 |    17456 | Rdr | 50 00 57 cd                                            |  ok | HALT
   43392 |    44384 | Rdr | 52(7)                                                  |     | WUPA
   45652 |    48020 | Tag | 44 00                                                  |     |
   56480 |    58944 | Rdr | 93 20                                                  |     | ANTICOLL
   60148 |    65972 | Tag | 88 1d 01 a8 3c                                         |     |
   83952 |    94416 | Rdr | 93 70 88 1d 01 a8 3c cb 81                             |  ok | SELECT_UID
   95684 |    99204 | Tag | 04 da 17                                               |     |
  106912 |   109376 | Rdr | 95 20                                                  |     | ANTICOLL-2
  110580 |   116468 | Tag | 01 58 34 78 15                                         |     |
  124240 |   134768 | Rdr | 95 70 01 58 34 78 15 3c 26                             |  ok | SELECT_UID-2
  135972 |   139556 | Tag | 20 fc 70                                               |     |
  151984 |   156752 | Rdr | e0 80 31 73                                            |  ok | RATS
  157956 |   166148 | Tag | 05 72 f7 a6 02 3d 9d                                   |  ok |
  466736 |   470288 | Rdr | c2 e0 b4                                               |  ok | RESTORE(224)
  471556 |   475076 | Tag | c2 e0 b4                                               |     |
 2911456 |  2912512 | Rdr | 26(7)                                                  |     | REQA
 2913700 |  2916068 | Tag | 44 00                                                  |     |
 2934304 |  2939072 | Rdr | 50 00 57 cd                                            |  ok | HALT
 2963568 |  2964560 | Rdr | 52(7)                                                  |     | WUPA
 2965828 |  2968196 | Tag | 44 00                                                  |     |
 2976144 |  2978608 | Rdr | 93 20                                                  |     | ANTICOLL
 2979796 |  2985620 | Tag | 88 1d 01 a8 3c                                         |     |
 2993840 |  3004304 | Rdr | 93 70 88 1d 01 a8 3c cb 81                             |  ok | SELECT_UID
 3005572 |  3009092 | Tag | 04 da 17                                               |     |
 3017408 |   3019872 | Rdr | 95 20                                                    |     | ANTICOLL-2
 3021076 |   3026964 | Tag | 01 58 34 78 15                                           |     |
 3033936 |   3044464 | Rdr | 95 70 01 58 34 78 15 3c 26                               |  ok | SELECT_UID-2
 3045668 |   3049252 | Tag | 20 fc 70                                                 |     |
 3060112 |   3064880 | Rdr | e0 80 31 73                                              |  ok | RATS
 3066084 |   3074276 | Tag | 05 72 f7 a6 02 3d 9d                                     |  ok |
 3366896 |   3370448 | Rdr | c2 e0 b4                                                 |  ok | RESTORE(224)
 3371716 |   3375236 | Tag | c2 e0 b4                                                 |     |

[ Stage 2: Tag detection part 2 ]

 3444720 |   3445712 | Rdr | 52(7)                                                    |     | WUPA
 3446980 |   3449348 | Tag | 44 00                                                    |     |
 3456816 |   3467280 | Rdr | 93 70 88 1d 01 a8 3c cb 81                               |  ok | SELECT_UID
 3468532 |   3472052 | Tag | 04 da 17                                                 |     |
 3479760 |   3490288 | Rdr | 95 70 01 58 34 78 15 3c 26                               |  ok | SELECT_UID-2
 3491492 |   3495076 | Tag | 20 fc 70                                                 |     |
 3506944 |   3511712 | Rdr | e0 80 31 73                                              |  ok | RATS
 3512916 |   3521108 | Tag | 05 72 f7 a6 02 3d 9d                                     |  ok |
 3814080 |   3832608 | Rdr | 02 00 a4 04 00 07 d2 76 00 00 85 01 01 00 35 c0          |  ok |
 4232468 |   4237204 | Tag | f2 07 a7 25                                              |     |
 4245824 |   4250592 | Rdr | f2 07 a7 25                                              |  ok |
 4448276 |   4454164 | Tag | 02 6a 82 93 2f                                           |     |
 4502624 |   4520000 | Rdr | 03 00 a4 04 00 07 d2 76 00 00 85 01 00 82 1d             |  ok |
 4835764 |   4840500 | Tag | f2 07 a7 25                                              |     |
 4848384 |   4853152 | Rdr | f2 07 a7 25                                              |  ok |
 5051604 |   5057492 | Tag | 03 6a 82 4f 75                                           |     |
 5105376 |   5108928 | Rdr | c2 e0 b4                                                 |  ok | RESTORE(224)
 5110196 |   5113716 | Tag | c2 e0 b4                                                 |     |
 5189904 |   5190896 | Rdr | 52(7)                                                    |     | WUPA
 5192164 |   5194532 | Tag | 44 00                                                    |     |
 5201888 |   5212352 | Rdr | 93 70 88 1d 01 a8 3c cb 81                               |  ok | SELECT_UID
 5213620 |   5217140 | Tag | 04 da 17                                                 |     |
 5224720 |   5235248 | Rdr | 95 70 01 58 34 78 15 3c 26                               |  ok | SELECT_UID-2
 5236452 |   5240036 | Tag | 20 fc 70                                                 |     |
 5306128 |   5309680 | Rdr | c2 e0 b4                                                 |  ok | RESTORE(224)
 5435184 |   5436176 | Rdr | 52(7)                                                    |     | WUPA
 5437444 |   5439812 | Tag | 44 00                                                    |     |
 5448144 |   5458608 | Rdr | 93 70 88 1d 01 a8 3c cb 81                               |  ok | SELECT_UID
 5459876 |   5463396 | Tag | 04 da 17                                                 |     |
 5470720 |   5481248 | Rdr | 95 70 01 58 34 78 15 3c 26                               |  ok | SELECT_UID-2
 5482452 |   5486036 | Tag | 20 fc 70                                                 |     |
 5496960 |   5501728 | Rdr | e0 80 31 73                                              |  ok | RATS
 5502932 |   5511124 | Tag | 05 72 f7 a6 02 3d 9d                                     |  ok |
 5811744 |   5830272 | Rdr | 02 00 a4 04 00 07 d2 76 00 00 85 01 01 00 35 c0          |  ok |
 6239348 |   6244084 | Tag | f2 07 a7 25                                              |     |
 6252464 |   6257232 | Rdr | f2 07 a7 25                                              |  ok |
 6455156 |   6461044 | Tag | 02 6a 82 93 2f                                           |     |
 6498752 |   6516128 | Rdr | 03 00 a4 04 00 07 d2 76 00 00 85 01 00 82 1d             |  ok |
 6842644 |   6847380 | Tag | f2 07 a7 25                                              |     |
 6855136 |   6859904 | Rdr | f2 07 a7 25                                              |  ok |
 7058612 |   7064500 | Tag | 03 6a 82 4f 75                                           |     |
 7118976 |   7122528 | Rdr | c2 e0 b4                                                 |  ok | RESTORE(224)
 7123796 |   7127316 | Tag | c2 e0 b4                                                 |     |

[ Stage 3: Actual communication between the tag and app ]

 7199856 |   7200848 | Rdr | 52(7)                                                    |     | WUPA
 7202100 |   7204468 | Tag | 44 00                                                    |     |
 7213040 |   7223504 | Rdr | 93 70 88 1d 01 a8 3c cb 81                               |  ok | SELECT_UID
 7224772 |   7228292 | Tag | 04 da 17                                                 |     |
 7235744 |   7246272 | Rdr | 95 70 01 58 34 78 15 3c 26                               |  ok | SELECT_UID-2
 7247460 |   7251044 | Tag | 20 fc 70                                                 |     |
 7262304 |   7267072 | Rdr | e0 80 31 73                                              |  ok | RATS
 7268276 |   7276468 | Tag | 05 72 f7 a6 02 3d 9d                                     |  ok |
 8474336 |   8490624 | Rdr | 02 00 a4 04 00 06 b0 00 00 00 00 23 a5 20                |  ok |
 8796084 |   8800820 | Tag | f2 07 a7 25                                              |     |
 8809072 |   8813840 | Rdr | f2 07 a7 25                                              |  ok |
 9019588 |   9025412 | Tag | 02 90 00 f1 09                                           |     |
 9080304 |   9091920 | Rdr | 03 80 41 00 00 02 02 11 0d d8                            |  ok |
 9343428 |   9348164 | Tag | f2 07 a7 25                                              |     |
 9356544 |   9361312 | Rdr | f2 07 a7 25                                              |  ok |
 9818308 |   9850692 | Tag | 03 95 15 02 0d 38 36 35 39 36 32 31 34 36 35 33 38 31 |     |
         |          |     | 11 04 60 47 88 e8 90 00 61 a4                       |  ok |
11593328 | 11596944 | Rdr | b2 67 c7                                            |  ok |
11770948 | 11774468 | Tag | a3 6f c6                                            |     |
```

Listing 2.2: Sniffing the tag enumeration and the "get serial number and time" command with a Proxmark3

Listing 2.2 shows that the Android app sends a "get serial number and time" command to the token. Part of the response, e.g. `0x38363539363231343635333831`, is encoded as ASCII and can easily be decoded to `8659621465381` – the serial number of the token. The time is encoded as four byte UNIX timestamp in big endian.

Sniffing the communication between tag and app was often used in this analysis as a method of reverse engineering. This was combined with analyzing the decompiled or disassembled code from the Android app, the Windows tool, or one of their libraries.

2.3 Internal card layout

The process of "delayering" an RFID card can unveil interesting information about the used chip, antenna design, or other important internals. This is typically achieved by putting the card into acetone. In most cases, acetone will weaken or dissolve some of the cards plastic parts and used adhesives by leaving the chips and antenna unharmed.

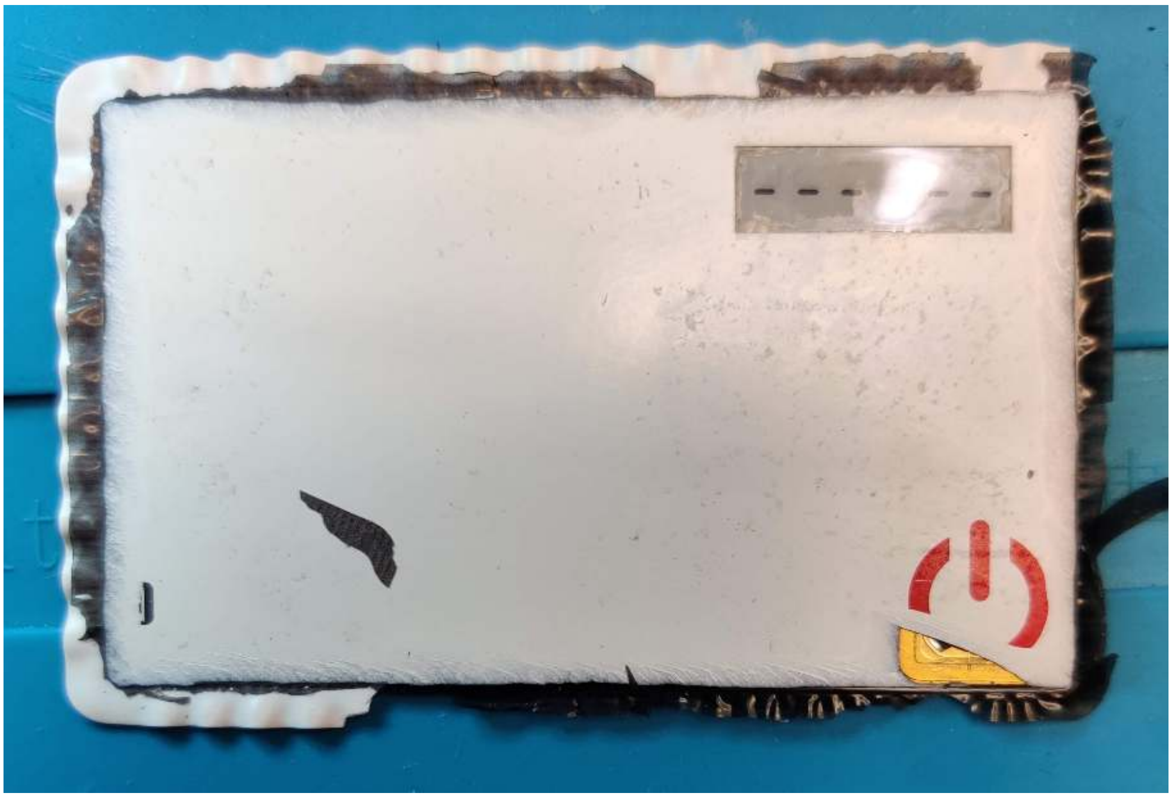


Figure 5: Card is only slightly damaged/dissolved by acetone (image by Philippe Teuwen)

Unfortunately, the Token2 OTPC-P2 card was pretty resistant to acetone. However, Philippe Teuwen [11], a very well known security researcher, hacker, and RFID specialist, was kind enough to help out and take things further.

By removing the plastics with sandpaper and a blade, Philippe Teuwen was able to delayer the whole card and get a closer look at its components.

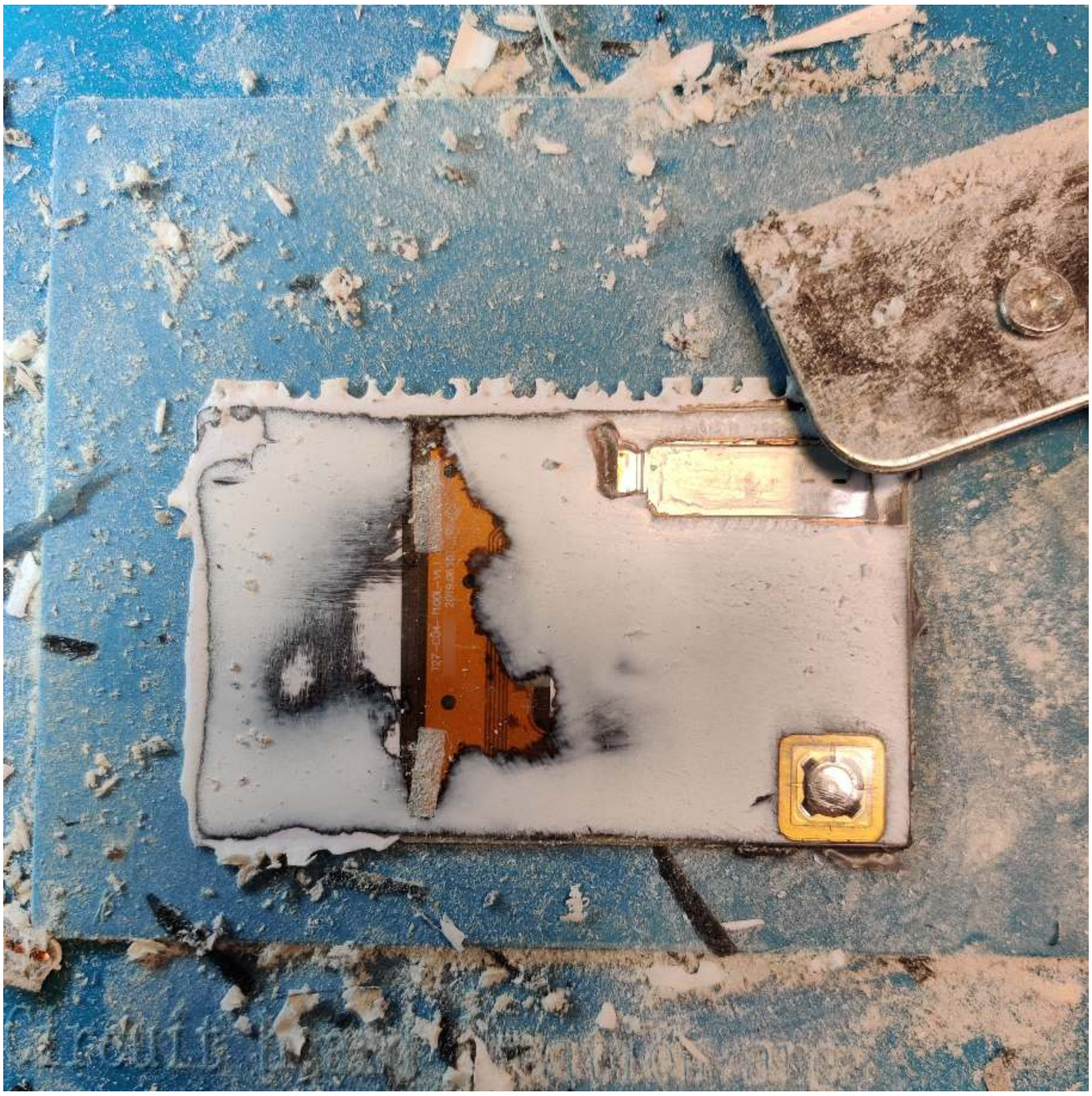


Figure 6: Delayered card (image by Philippe Teuwen)

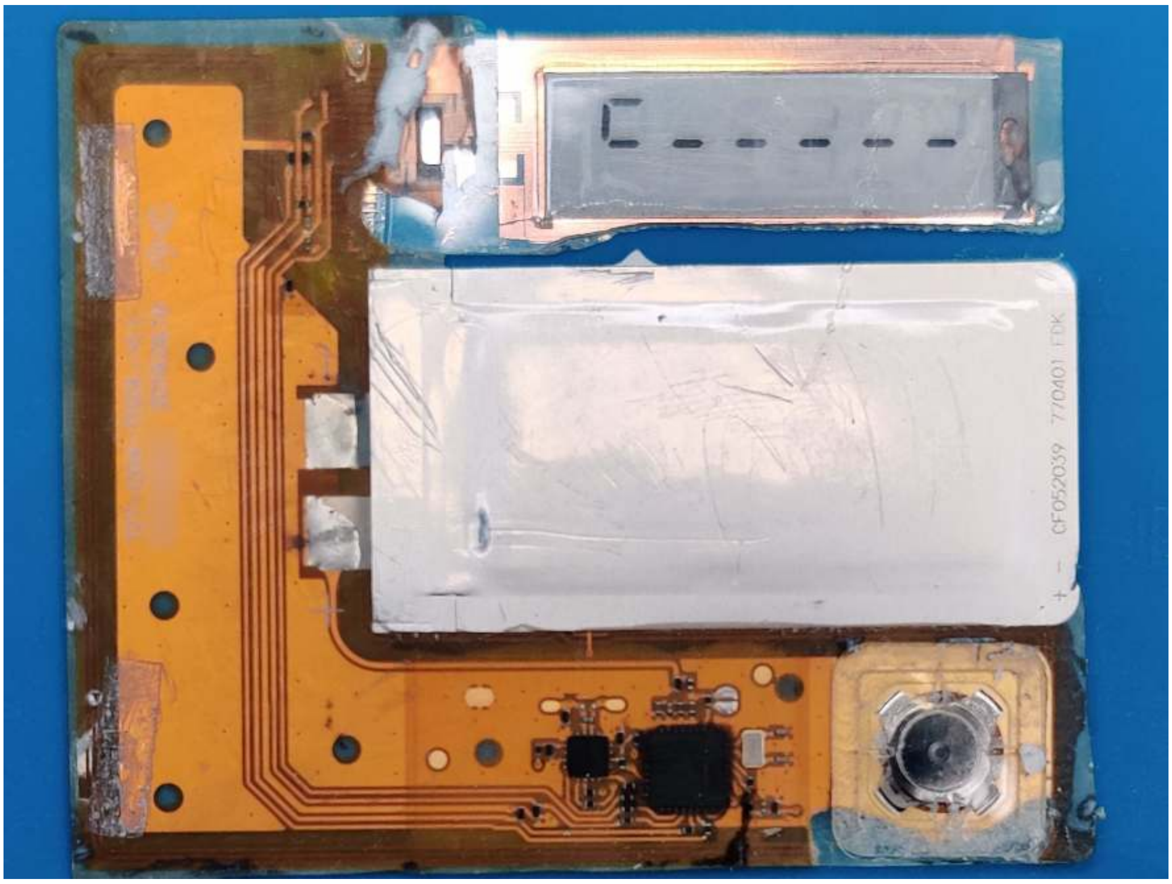


Figure 7: Delayered card (image by Philippe Teuwen)

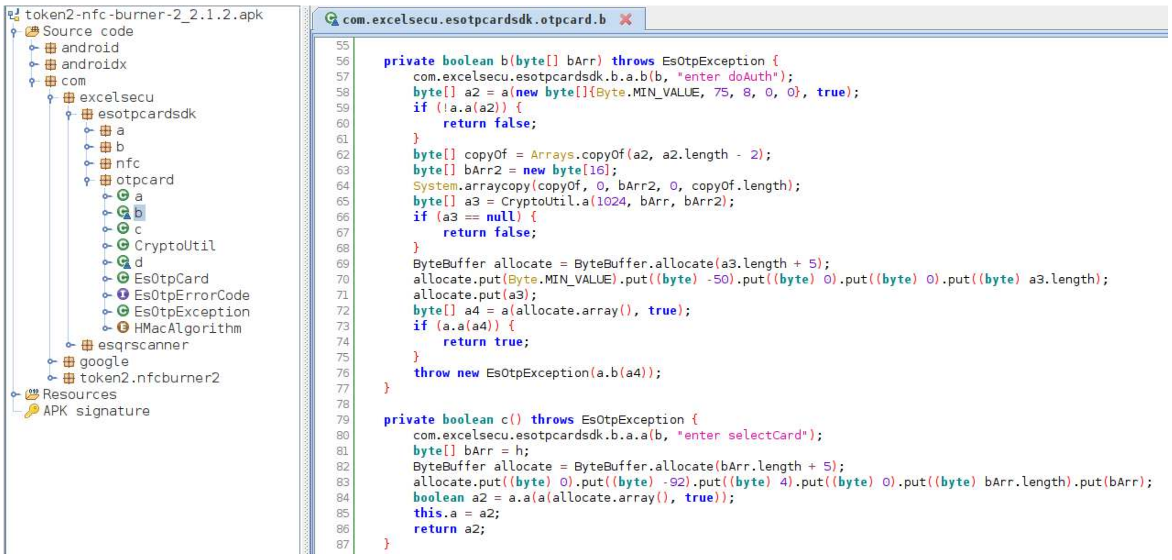
There are several markings on the internal parts of the card, which provide more information about it.

- Big chip: `T27D0 1951`, most likely the main controller with the firmware
- Small chip: `04J0 1907`, most likely a NFC controller
- Battery: `CF052039 770401 FDK`, a 3V lithium battery by FDK
- Markings on the PCB: `T27-C04-T100L-V1.1 2019.06.10`

The NFC controller on the card or the initial firmware is most likely built by Shanghai Fudan Microelectronics Group Co., Ltd. The first byte of a seven or ten byte UID indicates the manufacturer. In the case of Token2 OTPC-P2 card, this byte is `0x1D`, which is associated with Fudan. The manufacturer is well known for RFID chips.

2.4 Authentication

An authentication is required to change the configuration of the token or to write a seed. The authentication procedure was visible when the communication between a tag and the Android app was sniffed with a Proxmark3. This observation could be confirmed by looking at the disassembled code of the Android app (see Figure 8).



```java

    private boolean b(byte[] bArr) throws EsOtpException {
        com.excelsecu.esotpcardsdk.b.a.b(b, "enter doAuth");
        byte[] a2 = a(new byte[]{Byte.MIN_VALUE, 75, 8, 0, 0}, true);
        if (!a.a(a2)) {
            return false;
        }
        byte[] copyOf = Arrays.copyOf(a2, a2.length - 2);
        byte[] bArr2 = new byte[16];
        System.arraycopy(copyOf, 0, bArr2, 0, copyOf.length);
        byte[] a3 = CryptoUtil.a(1024, bArr, bArr2);
        if (a3 == null) {
            return false;
        }
        ByteBuffer allocate = ByteBuffer.allocate(a3.length + 5);
        allocate.put(Byte.MIN_VALUE).put((byte) -50).put((byte) 0).put((byte) 0).put((byte) a3.length);
        allocate.put(a3);
        byte[] a4 = a(allocate.array(), true);
        if (a.a(a4)) {
            return true;
        }
        throw new EsOtpException(a.b(a4));
    }

    private boolean c() throws EsOtpException {
        com.excelsecu.esotpcardsdk.b.a.a(b, "enter selectCard");
        byte[] bArr = h;
        ByteBuffer allocate = ByteBuffer.allocate(bArr.length + 5);
        allocate.put((byte) 0).put((byte) -92).put((byte) 4).put((byte) 0).put((byte) bArr.length).put(bArr);
        boolean a2 = a.a(a(allocate.array(), true));
        this.a = a2;
        return a2;
    }
```

Figure 8: Decompiled authentication method of the NFC Burner 2 Android application

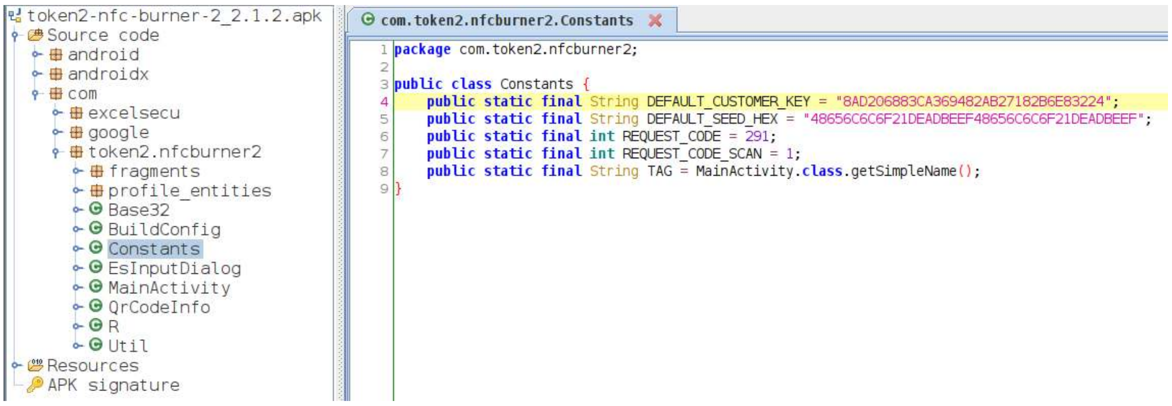
By diving deeper into the code and the sniffed communication, the authentication process was reconstructed as follows:

1. Request a challenge from the tag using the APDU `804B080000`.
2. Receive a challenge from the tag, e.g. `F29E08B17821653E`.
3. Expand the challenge to a 16 byte value by padding zeros: `F29E08B17821653E0000000000000000`.
4. Encrypt the challenge using the SM4 algorithm in ECB mode with a secret 16 byte long key.
5. Send back the encrypted challenge using the APDU: `80CE00001073EA53B7E7E77DD81AEE5BC106-9E053A`.
6. The tags responds with `9000` indicating that the authentication was successful.

The whole part about encrypting the challenge with the SM4 cipher is not implemented in the Java- or Kotlin-based code of the Android app. The native library `libesotpcommon.so` is used for this. The Android app ships with different versions of this library, each for a specific platform. Fortunately, the 32 bit x86 library did still include the function names and debug symbols, as the following output illustrates.

```
> tree --charset=ascii lib
lib
|-- arm64-v8a
|   `-- libesotpcommon.so
|-- armeabi
|   `-- libesotpcommon.so
|-- armeabi-v7a
|   `-- libesotpcommon.so
|-- x86
|   `-- libesotpcommon.so
`-- x86_64
    `-- libesotpcommon.so

> file lib/x86/libesotpcommon.so
lib/x86/libesotpcommon.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), ⤸
    dynamically linked, BuildID[sha1]=e35a887407b6adc3ba0e05298c006b1a9572e1c8, with ⤸
    debug_info, not stripped
```

However, the most interesting ingredient of the authentication is the key that is used to encrypt the challenge. It turned out that this key is hard-coded into the app (see Figure 9).



```java
package com.token2.nfcburner2;

public class Constants {
    public static final String DEFAULT_CUSTOMER_KEY = "8AD206883CA369482AB27182B6E83224";
    public static final String DEFAULT_SEED_HEX = "48656C6C6F21DEADBEEF48656C6C6F21DEADBEEF";
    public static final int REQUEST_CODE = 291;
    public static final int REQUEST_CODE_SCAN = 1;
    public static final String TAG = MainActivity.class.getSimpleName();
}
```

Figure 9: Decompiled class of the NFC Burner 2 Android application with static constants, e.g. the key for encrypting the challenge (`DEFAULT_CUSTOMER_KEY`)

Although the name `DEFAULT_CUSTOMER_KEY` implies that the key might be changeable, no such functionality was found.

We developed a small Python script which allows to perform an authentication using a cheap USB RFID reader like the ACR 122u. The source code of this Python script is provided in Listing 5.1.

By evaluating the authentication it became clear, that for each wrong authentication a counter is decreased. After five failed authentication attempts, the authentication method is blocked and it is not longer possible to execute any command that requires authentication.

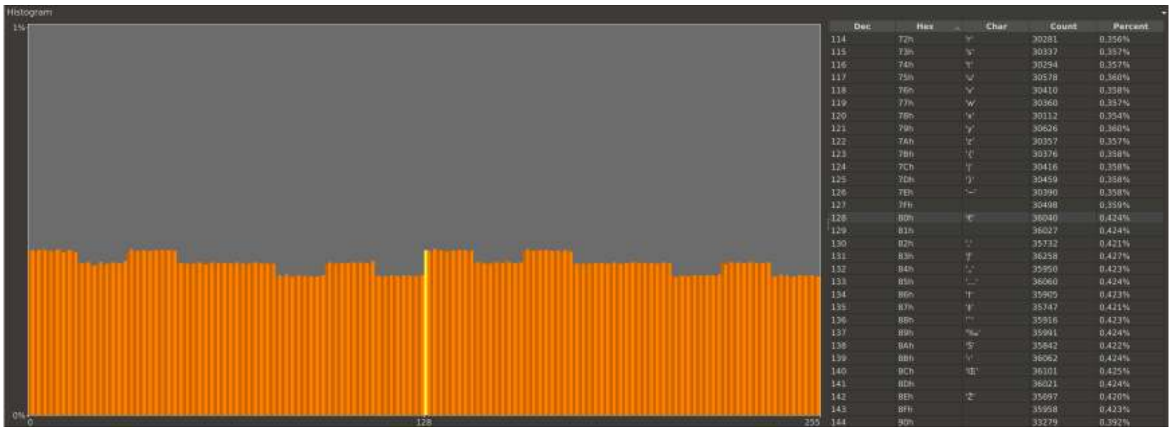
Another interesting observation was made about the random number generator of the card. Details can be found in Section 2.5.

2.5 Bad RNG

The OTPC-P2 token has an internal random number generator (RNG). For most security related applications, a random number generator must generate true random numbers and not pseudorandom numbers.

One use case where the RNG of the card is used is when an NFC reader asks the token for a challenge in order to authenticate (see Section 2.4). To evaluate the quality of the RNG, a simple Python script was developed which asks the token for a large number of challenges.

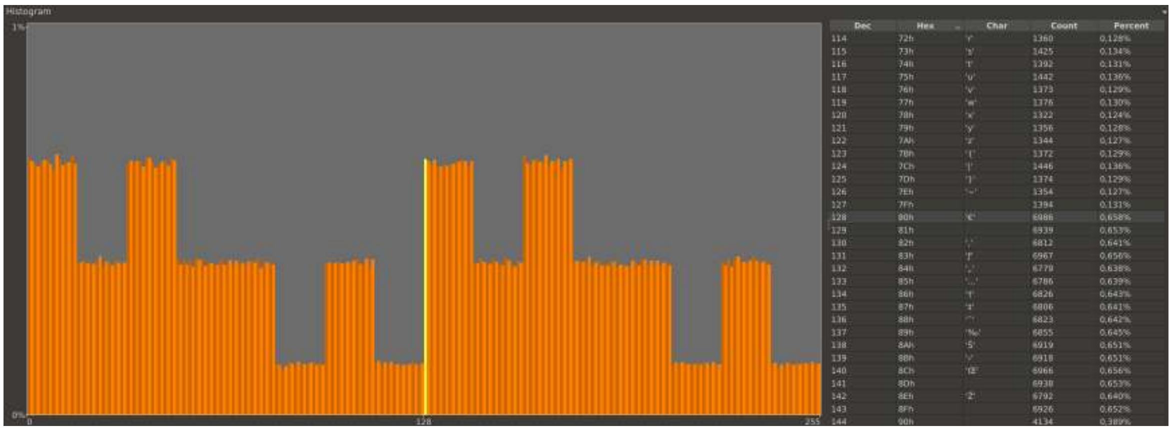
Over eight megabytes of random data was collected this way. The data was collected in chunks of eight bytes, since a challenge from the token consists of eight bytes. A histogram visualization (that shows which byte is present how many times) quickly revealed that there are issues with the used RNG (see Figure 10).



Dec	Hex	Char	Count	Percent
114	72h	'r'	30281	0,356%
115	73h	's'	30337	0,357%
116	74h	't'	30294	0,357%
117	75h	'u'	30578	0,360%
118	76h	'v'	30410	0,358%
119	77h	'w'	30360	0,357%
120	78h	'x'	30112	0,354%
121	79h	'y'	30626	0,360%
122	7Ah	'z'	30357	0,357%
123	7Bh	'{'	30376	0,358%
124	7Ch	'|'	30416	0,358%
125	7Dh	'}'	30459	0,358%
126	7Eh	'~'	30390	0,358%
127	7Fh		30498	0,359%
128	80h	'€'	36040	0,424%
129	81h		36027	0,424%
130	82h	'‚'	35732	0,421%
131	83h	'ƒ'	36258	0,427%
132	84h	'„'	35950	0,423%
133	85h	'…'	36060	0,424%
134	86h	'†'	35905	0,423%
135	87h	'‡'	35747	0,421%
136	88h	'ˆ'	35916	0,423%
137	89h	'‰'	35991	0,424%
138	8Ah	'Š'	35842	0,422%
139	8Bh	'‹'	36062	0,424%
140	8Ch	'Œ'	36101	0,425%
141	8Dh		36021	0,424%
142	8Eh	'Ž'	35697	0,420%
143	8Fh		35958	0,423%
144	90h		33279	0,392%

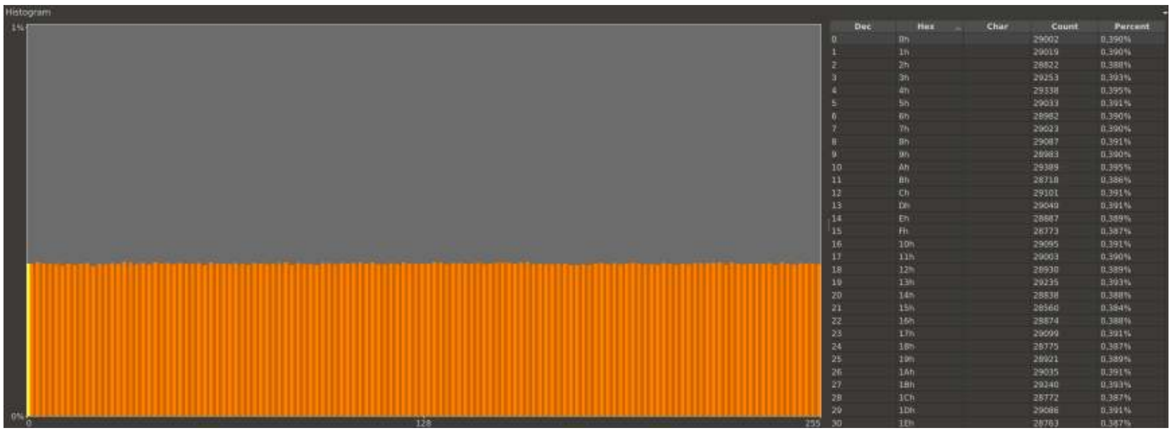
Figure 10: Histogram of the random data collected by requesting challenges from the token

Upon further inspection it became clear, that it is the first byte of the eight byte challenge, which introduces the bad randomness. This is clearly visible when the histogram of only byte number 1 (see Figure 11) is compared with the histogram of bytes number 2 to 8 (see Figure 12).



Dec	Hex	Char	Count	Percent
114	72h	'r'	1360	0,128%
115	73h	's'	1425	0,134%
116	74h	't'	1392	0,131%
117	75h	'u'	1442	0,136%
118	76h	'v'	1373	0,129%
119	77h	'w'	1376	0,130%
120	78h	'x'	1322	0,124%
121	79h	'y'	1356	0,128%
122	7Ah	'z'	1344	0,127%
123	7Bh	'{'	1372	0,129%
124	7Ch	'|'	1446	0,136%
125	7Dh	'}'	1374	0,129%
126	7Eh	'~'	1354	0,127%
127	7Fh		1394	0,131%
128	80h	'€'	6986	0,658%
129	81h		6939	0,653%
130	82h	'‚'	6812	0,641%
131	83h	'ƒ'	6967	0,656%
132	84h	'„'	6779	0,638%
133	85h	'…'	6786	0,639%
134	86h	'†'	6826	0,643%
135	87h	'‡'	6806	0,641%
136	88h	'ˆ'	6823	0,642%
137	89h	'‰'	6855	0,645%
138	8Ah	'Š'	6919	0,651%
139	8Bh	'‹'	6918	0,651%
140	8Ch	'Œ'	6966	0,656%
141	8Dh		6938	0,653%
142	8Eh	'Ž'	6792	0,640%
143	8Fh		6926	0,652%
144	90h		4134	0,389%

Figure 11: Histogram of byte number 1 of all collected challenges



Dec	Hex	Char	Count	Percent
0	0h		29002	0,390%
1	1h		29019	0,390%
2	2h		28822	0,388%
3	3h		29253	0,393%
4	4h		29338	0,395%
5	5h		29033	0,391%
6	6h		28982	0,390%
7	7h		29023	0,390%
8	8h		29087	0,391%
9	9h		28983	0,390%
10	Ah		29389	0,395%
11	Bh		28718	0,386%
12	Ch		29101	0,391%
13	Dh		29049	0,391%
14	Eh		28887	0,389%
15	Fh		28773	0,387%
16	10h		29095	0,391%
17	11h		29003	0,390%
18	12h		28930	0,389%
19	13h		29235	0,393%
20	14h		28838	0,388%
21	15h		28560	0,384%
22	16h		28874	0,388%
23	17h		29099	0,391%
24	18h		28775	0,387%
25	19h		28921	0,389%
26	1Ah		29035	0,391%
27	1Bh		29240	0,393%
28	1Ch		28772	0,387%
29	1Dh		29086	0,391%
30	1Eh		28763	0,387%

Figure 12: Histogram of byte number 2 to byte number 8 of all collected challenges

Inspecting the first byte of each challenge at bit level, the RNG error was narrowed down to two bits (bit number 5 and bit number 7) which do not have a 50:50 chance of being 1 or 0:

- bit 1: 0:50%, 1:50%
- bit 2: 0:50%, 1:50%
- bit 3: 0:50%, 1:50%
- bit 4: 0:50%, 1:50%
- bit 5: 0:66%, 1:33%
- bit 6: 0:50%, 1:50%
- bit 7: 0:66%, 1:33%
- bit 8: 0:50%, 1:50%

This reduces the 256 bit of entropy of a eight byte challenge to 254 bit of good entropy. This is likely still enough entropy for the authentication to be sufficiently secure. It is unclear, if the bad RNG can be exploited in other attack scenarios.

2.6 Known and unknown commands

A Java Card like the OTPC-P2 by Token2 can have lots of different APDUs. Although there is only one byte in an APDU that is declaring the instruction (INS), there could be many different contexts and parameters for one command. Even more commands or custom protocols can be designed within the data/payload field of an APDU.

It is close to impossible to enumerate all commands/payloads of an ISO 7816-4 tag. There is virtually an endless number of possibilities and the protocol or the RFID tags are not fast enough. Furthermore, ISO 7816 tags can have more than one application, with each application having its own APDUs and data format.

The OTPC-P2 seem to only use one application: `0xB000000000023`. This was reconstructed from sniffing the communication between OTPC-P2 tags and the NFC Burner 2 Android app with a Proxmark3. Similar to enumerating all known APDUs/payloads, enumerating all applications on a Java Card is not feasible.

In an attempt to find applications that might have an application identifier (AID) close to the known application, a Python script was developed (see Listing 5.3). This script just enumerates AIDs over a given range. However, no other valid AID was found.

Although it is close to impossible to find all valid APDUs, the next step was to find at least some of them. A possibly complex tag like the OTPC-P2 is likely to have more commands than the known commands used by the customer software. To search for APDUs of the type "case 1" or "case 2 (short)", the client software of the Proxmark3 was extended by the command `hf 14a apdufind`. The maintainers of the advanced Proxmark3 repository at `https://github.com/RfidResearchGroup/proxmark3` were kind enough to merge the changes.

```
[usb] pm3 --> hf 14a apdufind -h

Enumerate APDU's of ISO7816 protocol to find valid CLS/INS/P1/P2 commands.
It loops all 256 possible values for each byte.
The loop oder is INS -> P1/P2 (alternating) -> CLA.
Tag must be on antenna before running.

usage:
    hf 14a apdufind [-hlv] [-c <hex>] [-i <hex>] [--p1 <hex>] [--p2 <hex>] [-r <number>]⤶
     [-e <number>] [-s <hex>]...


options:
    -h, --help                        This help
    -c, --cla <hex>                   Start value of CLASS (1 hex byte)
    -i, --ins <hex>               Start value of INSTRUCTION (1 hex byte)
    --p1 <hex>                    Start value of P1 (1 hex byte)
    --p2 <hex>                    Start value of P2 (1 hex byte)
    -r, --reset <number>          Minimum seconds before resetting the tag (to prevent ⤸
    timeout issues). Default is 5 minutes
    -e, --error-limit <number>    Maximum times an status word other than 0x9000 or 0x6⤸
    D00 is shown. Default is 512.
    -s, --skip-ins <hex>          Do not test an instructions (can be specified ⤸
    multiple times)
    -l, --with-le                 Search  for APDUs with Le=0 (case 2S) as well
    -v, --verbose                 Verbose output

examples/notes:
    hf 14a apdufind
    hf 14a apdufind --cla 80
    hf 14a apdufind --cla 80 --error-limit 20 --skip-ins a4 --skip-ins b0 --with-le
```

From sniffing the RFID traffic and from reverse engineering the client software for the OTPC-P2 token, the following commands (INS of APDU) were known:

- `0x41`: Request data like the current time and serial number.
- `0xA4`: Select an application or file. Only AID `0xB00000000023` seem to be used.
- `0x4B`: Get a challenge for the authentication procedure.
- `0xCE`: Authenticate. This must be done before configuration changes or burning a seed. The encrypted challenge must be sent for the authentication to be successful (see Section 2.4).
- `0xD4`: Write/burn a configuration.
- `0xC5`: Write/burn a seed.
- `0xC7`: "SealCard". This command was never used by the Android app and was recovered from the library `SeedFlash.dll` of the Windows application. It remains unsure, what exactly this command is used for.

With the `apdufind` command of the Proxmark3 some more commands/APDUs were detected. The unknown commands/APDUs are as follows:

- `0x808D000000`: Unknown authentication mechanism. Using the authentication procedure and key from Section 2.4 was not successful. It is limited to three attempts.
- `0x8040000000`: Unknown command. The response is just the status word `0x9000`, indicating success.

Some known commands were tested with different payloads. One of them, the "get time and serial number" command (`0x41`) showed an interesting behavior, where the data specify which values are queried. As the following figures show, `0x02` seems to be the current time and `0x11` the serial number. Other valid values were not found.

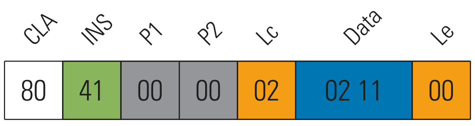


Figure 13: Original "get time and serial number" command

CLA	INS	P1	P2	Lc	Data	Le
80	41	00	00	02	02 02	00

Figure 14: Modified data in "get time and serial number" command returning the serial number twice

CLA	INS	P1	P2	Lc	Data	Le
80	41	00	00	02	11 11	00

Figure 15: Modified data in "get time and serial number" command returning the current time twice

```
[usb] pm3 --> hf 14a apdu -s -d 80 41 0000 02 0202
[+] ( select )
[+] >>> 80410000020202
[=] APDU: case=0x03 cla=0x80 ins=0x41 p1=0x00 p2=0x00 Lc=0x02(2) Le=0x00(0)
[+] <<< 951E020D38363539363231343635333831020D38363539363231343635333831 9000 | ....86596↩
    21465381..8659621465381..
[+] <<< status: 90 00 - Command successfully executed (OK).
```

It is likely that there are more unknown applications, instructions, or APDU structures and payloads. Within this research, it did not become clear what the unknown commands are used for or what they exactly do. The unknown authentication could be debug access or a "backdoor" for the reseller or manufacturer (Token2).

2.7 Denial of service

A denial of service attack (DoS) aims to make a device or software unusable for its designed purpose. The observations of the authentication procedure (see Section 2.4) showed that a trivial attack vector for DoS is present: Everyone can change the card's secret or configuration with the provided mobile application. This is because all cards use the same key for authentication. If someone overwrites the key or configuration, the token will no longer produce valid OTPs, resulting in a denial of service state. Admittedly, this "attack" needs close physical proximity to an activated card. Furthermore, the card can be reinitialized with the correct data so that it will work again.

Another simple and partial DoS state is when the authentication failed five times in a row. After this, the card locks the authentication method, making changes to the card's configuration impossible.

By accident, we were able to produce another DoS state for a card. After sending some random test APDUs to the token, it did not respond anymore. Even the display was no longer cleared. This state was permanent and the card was completely unusable afterwards – in other words "bricked". However, reproducing this state on a second card was not successful. It could be that not only the sent data was responsible for entering this state, but also a sudden cut-off of the reader field at a specific point in time (tear-off attack).

2.8 e-Ink display issues

The Token2 OTPC-P2 card shows the current valid OTP on an e-Ink display after the button is pressed. Depending on the card's configuration, the OTP is valid for either 30 or 60 seconds. The display can be configured to "stay on" for either 15, 30, 60 or 120 seconds.

To turn the i-Ink display "off", the shown values must be reset. The card does this by coloring the full display to black and than back to white. After that, the OTP should no longer be visible. However, this is not the case. There seems to be a "burn in" effect on the display, that slightly shows the last OTP, even after the display was cleared (see Figures 16 and 17). This could be a security issue in case the card is configured to OTPs being valid for 60 seconds and a display sleep time of e.g. 15 seconds. In that case, the valid OTP should only be visible for 15 seconds. However, an attacker might get a glimpse at the display shortly after and use the slightly visible and still valid OTP.

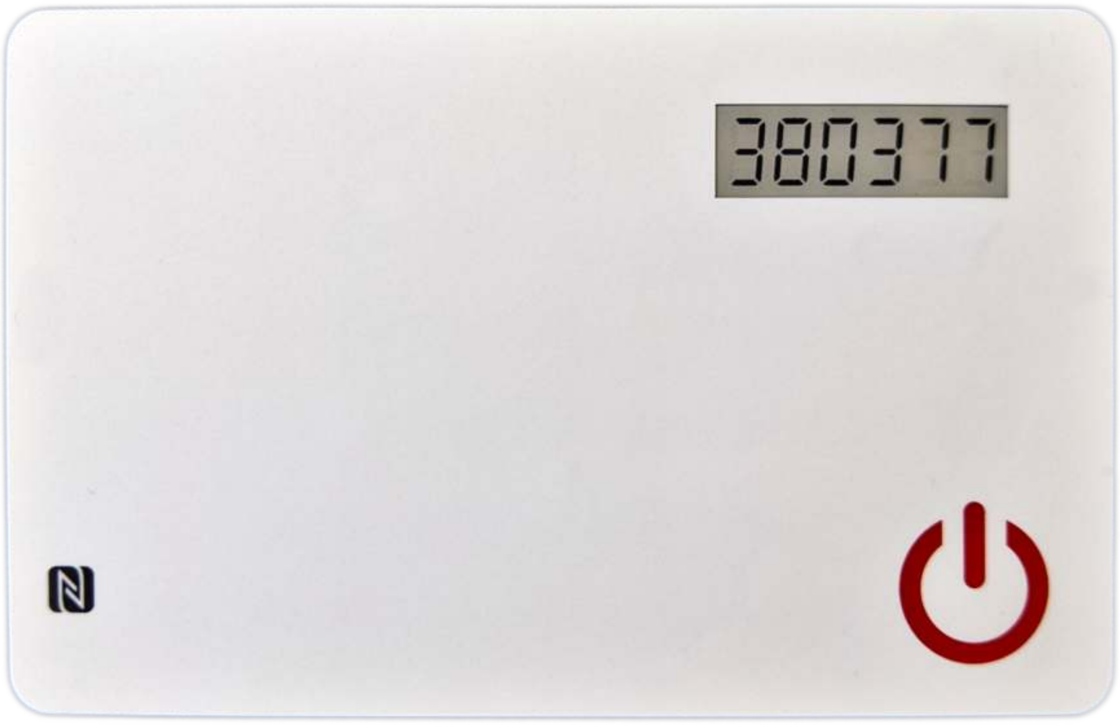


Figure 16: Activated OTPC-P2 token

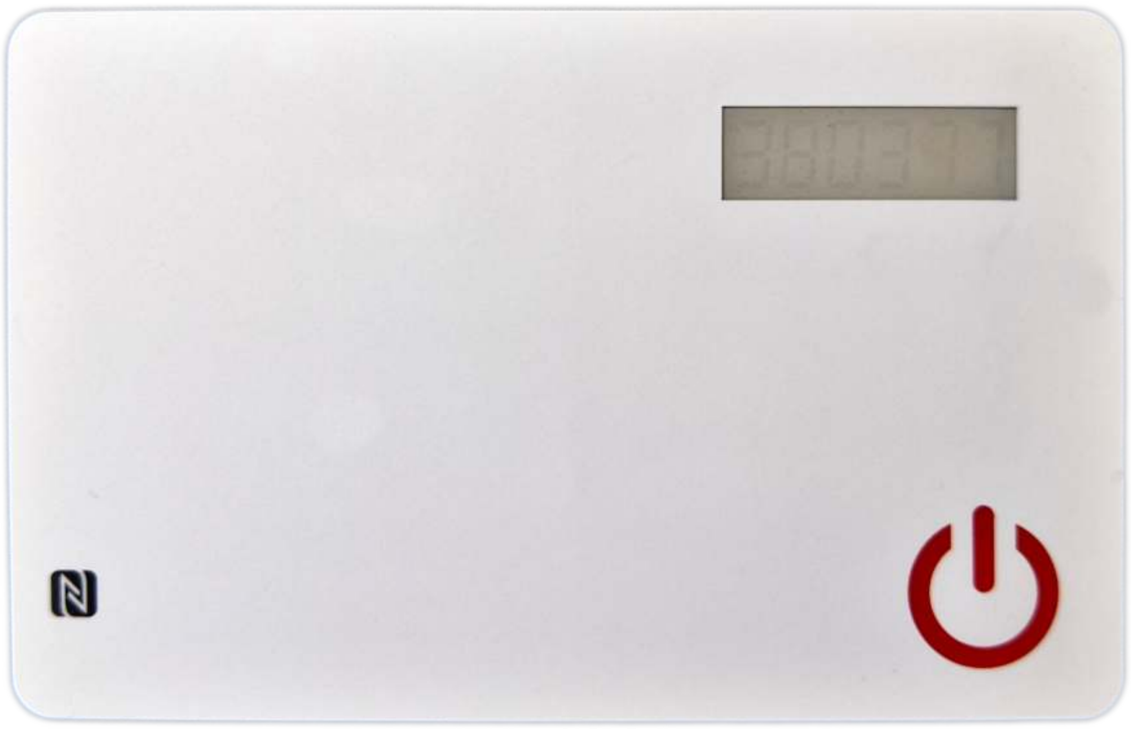


Figure 17: Deactivated OTPC-P2 token with the last OTP still slightly visible

The Protectimus card (see Section 2.10) is not affected by this issue. This might be because the token uses another e-Ink display or because it clears the display twice.

2.9 Instructions for destroying a card

On the backside of the Token2 OTPC-P2 card, there are instructions on how to cut the card in order to safely destroy it (see Figure 18).

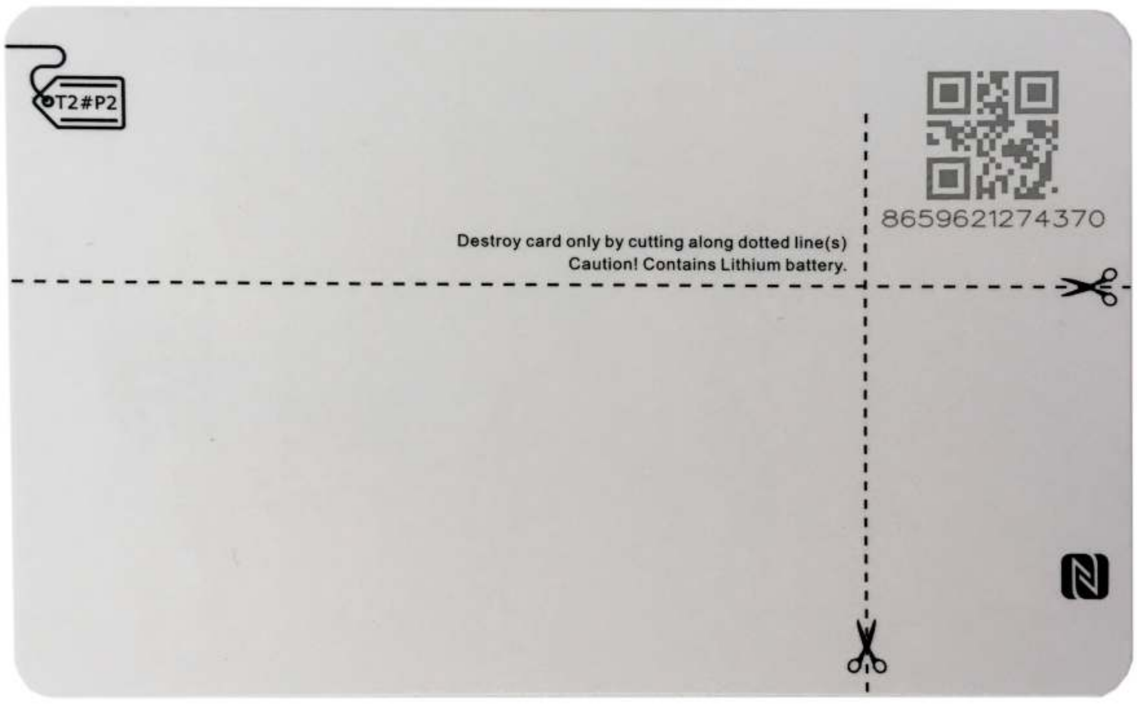

T2#P2
8659621274370
Destroy card only by cutting along dotted line(s)
Caution! Contains Lithium battery.

Figure 18: Instructions for destroying the card printed on the backside

However, when shining bright light through a card (see Figure 19), it is visible that the horizontal cut will damage the Lithium battery. Destroying the battery will render the card unusable, because the real time clock (RTC) for the TOTP authentication would go out of sync. Cutting batteries, however, is considered dangerous. The manufacturer seems to know this, because there is also a warning label on the card saying "*Caution! Contains Lithium battery*". The vertical cut does not destroy anything.

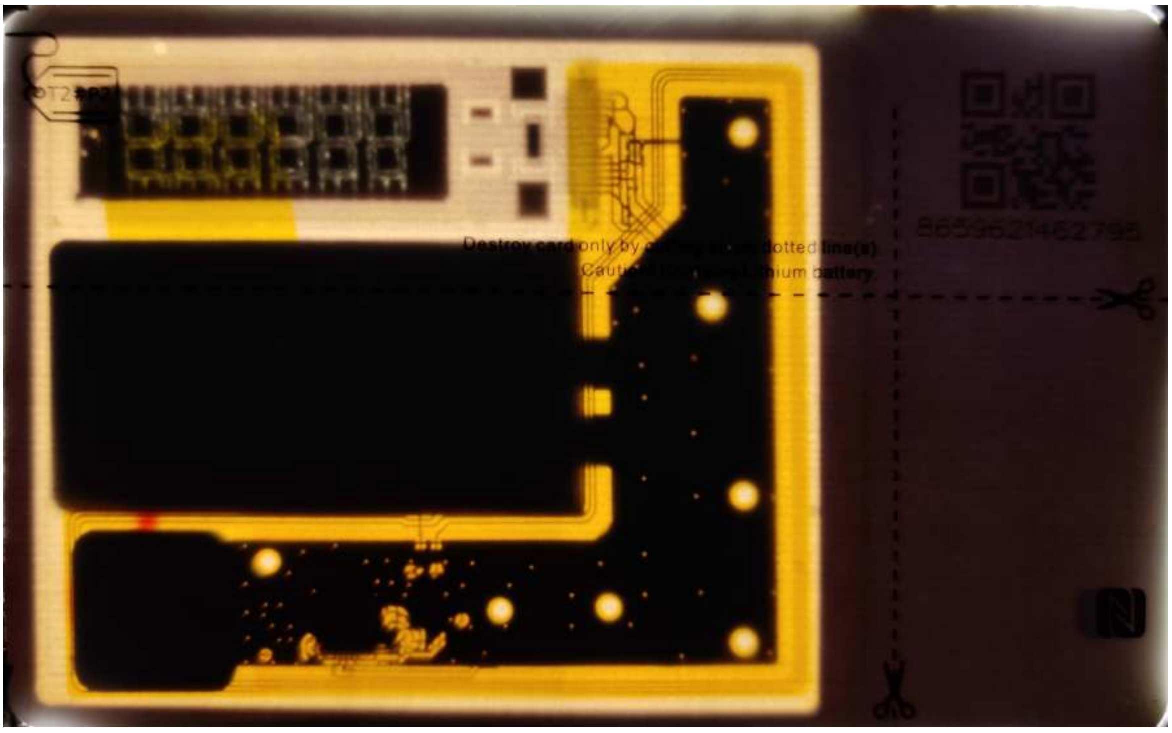


Figure 19: Internal components visible by shining bright light through the card (image by Philippe Teuwen)

It is unclear why the instructions are printed in this way. If the horizontal cut would be a bit higher, it would not destroy the battery. In that case, however, the battery would still be connected to the unharmed chips. An attacker could just reconnect the antenna and display and the card should work again with the previously stored secret still intact.

The most effective way would be to cut the chip containing the seed for the TOTP authentication. In this way, the battery would not be harmed and the secret seed would be very hard or impossible to recover.

2.10 Interesting data and unanswered questions

At the point of writing this case study, a lot of effort has gone into understanding how this card works. Although some interesting findings and security relevant observations were made, there are still a lot of open questions.

Some strange practices by the manufacturer or hints in applications raise even more questions.

The following list sums up some of the more interesting hints and open questions we had no time to further investigate yet.

- The Token2 OTPC-P2 uses the same “customer key” for authentication on all cards. Are there other customer keys for similar products? At least one other key was found in the Windows application distributed by Token2. Also, can the customer key be changed by a user?
- What else can the card do? There are hidden commands – one is another authentication – but for which purpose?
- How can the permanent denial of service sate (see Section 2.7) be reproduced? What is causing it?
- The Windows application by Token2 has a test function to verify whether a card produces valid OTPs after programming a seed. However, this test function just opens an OTP test website and sends the secret seed to it. This is considered a very bad practice, because the secret is therefore published to an untrusted website.
- Even if no button is pressed, the card responds to some commands. For example, the full ISO 14443 discovery process is supported (anti-collision, retrieving the UID, etc.), but the token does not respond to APDUs. Are there any other undocumented commands that might work at that stage?
- Is it possible to read out the current OTP via NFC? So far, neither the Android nor the Windows application have a function for that. Is there an undocumented function?
- Is there a way to change the time of the internal real-time clock (RTC) of the token without losing the configured secret seed? This has been possible in the previous generation of this token, maybe there is still some legacy code in the firmware.

3 Case Study II: Protectimus SLIM NFC

In our second case study, we analyzed the programmable hardware TOTP token Protectimus SLIM NFC shown in Figure 20.

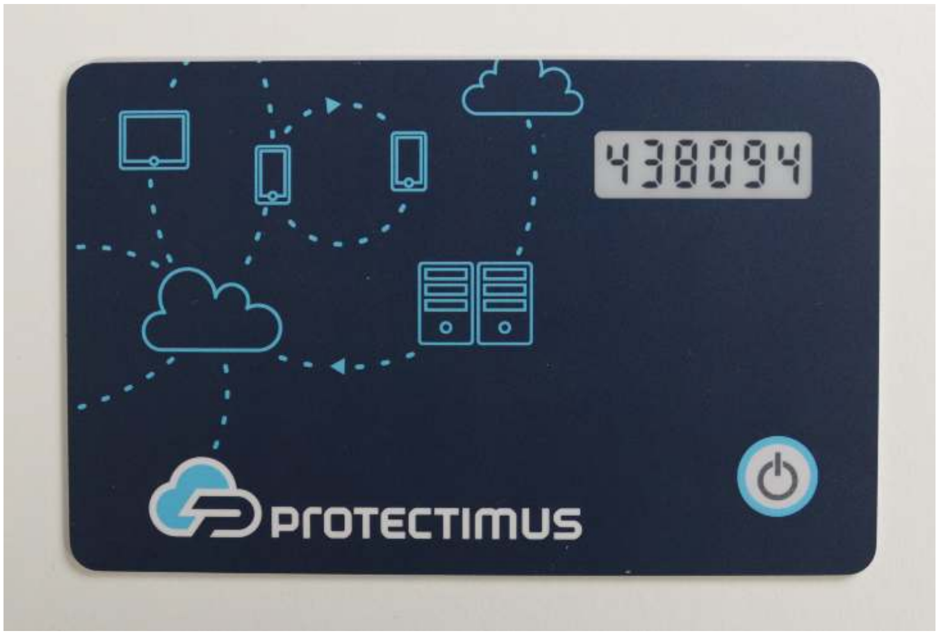
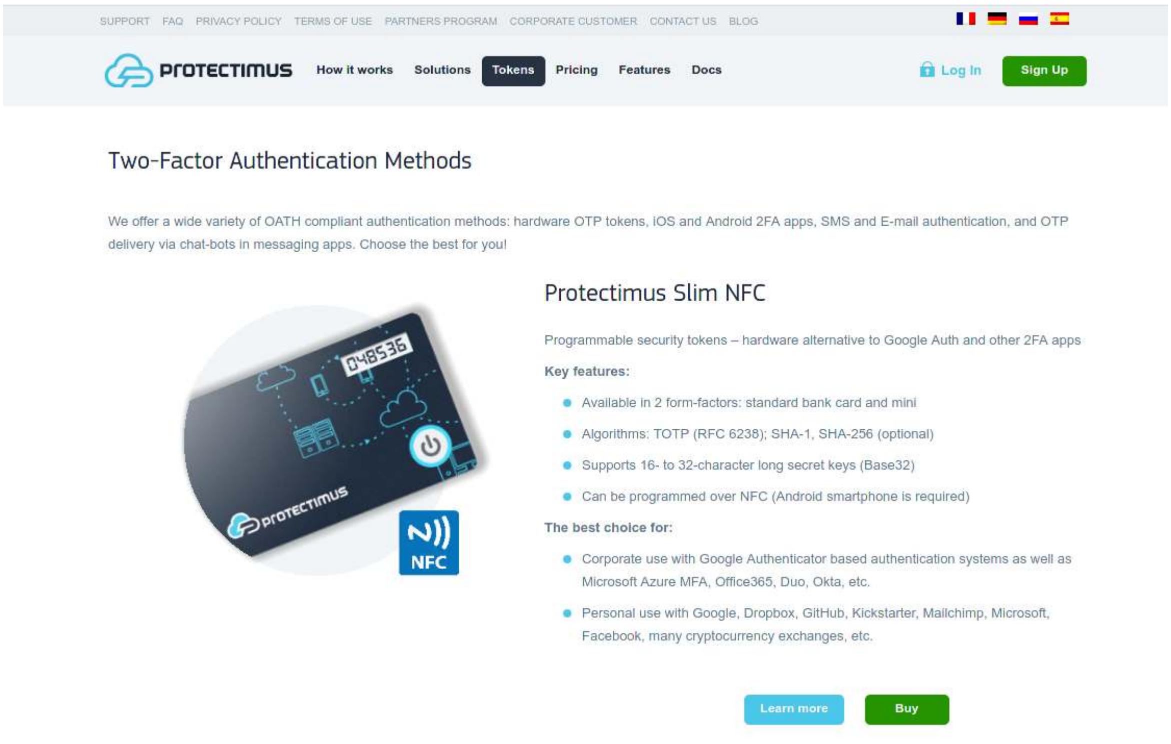


Figure 20: Protectimus SLIM NFC hardware TOTP token

This hardware token is designed for the use with two-factor authentication methods and can be programmed via NFC using an Android app. Its product description is shown in Figure 21.



Two-Factor Authentication Methods

We offer a wide variety of OATH compliant authentication methods: hardware OTP tokens, iOS and Android 2FA apps, SMS and E-mail authentication, and OTP delivery via chat-bots in messaging apps. Choose the best for you!

Protectimus Slim NFC

Programmable security tokens – hardware alternative to Google Auth and other 2FA apps

Key features:

- Available in 2 form-factors: standard bank card and mini
- Algorithms: TOTP (RFC 6238); SHA-1, SHA-256 (optional)
- Supports 16- to 32-character long secret keys (Base32)
- Can be programmed over NFC (Android smartphone is required)

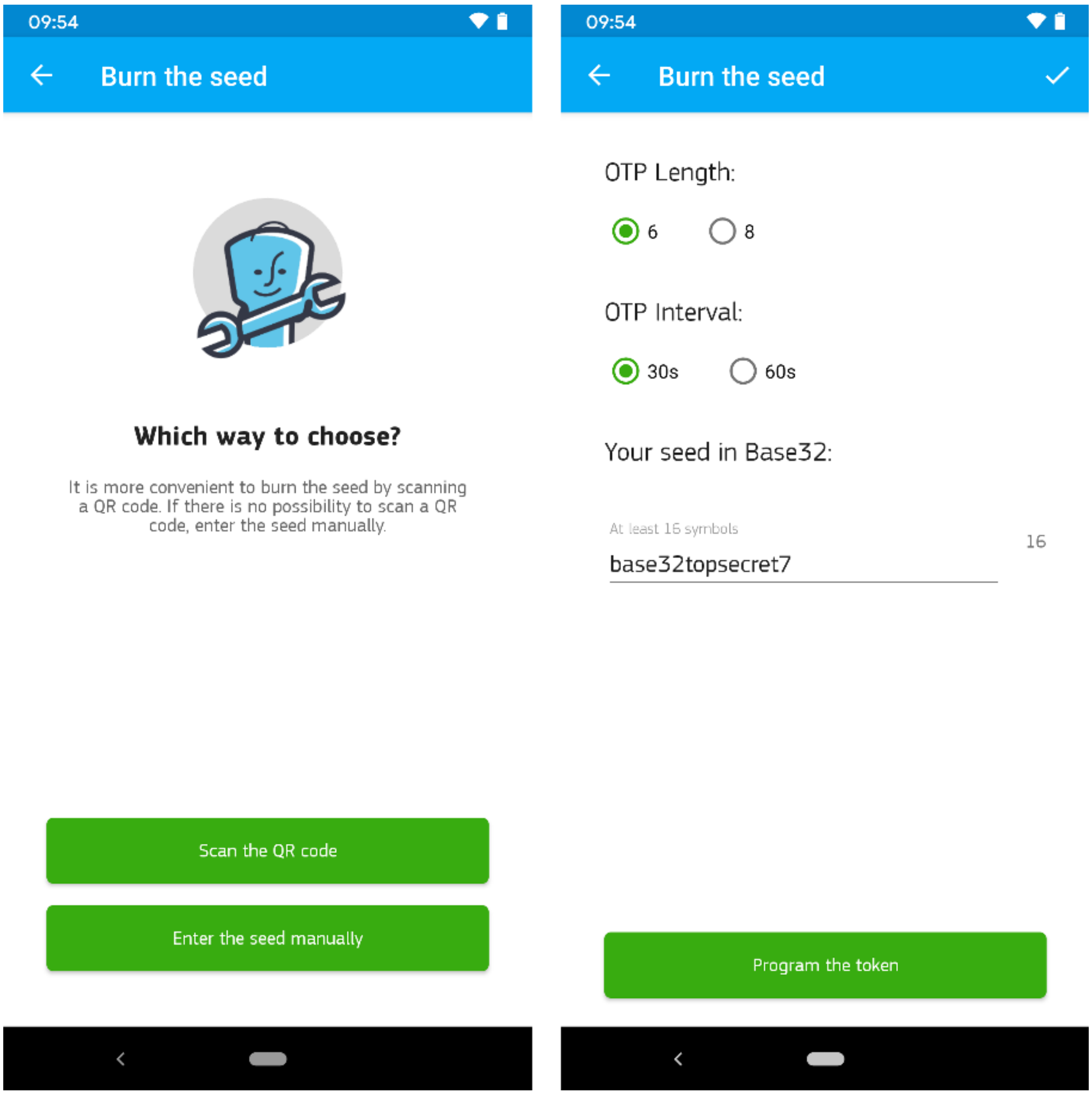
The best choice for:

- Corporate use with Google Authenticator based authentication systems as well as Microsoft Azure MFA, Office365, Duo, Okta, etc.
- Personal use with Google, Dropbox, GitHub, Kickstarter, Mailchimp, Microsoft, Facebook, many cryptocurrency exchanges, etc.

Figure 21: Protectimus SLIM NFC product description

3.1 Normal operation

Before the TOTP token can be used, it has to be configured via the PROTECTIMUS TOTP BURNER app [12] for Android. This can either be done by scanning a corresponding QR code or by manually entering the required configuration parameters consisting of the OTP length (6 or 8 digits), the OTP time interval (30 or 60 seconds), and the seed (Base32-encoded secret), as Figure 22 illustrates.



Which way to choose?

It is more convenient to burn the seed by scanning a QR code. If there is no possibility to scan a QR code, enter the seed manually.

OTP Length: [✓ 6] 8

OTP Interval: [✓ 30s] 60s

Your seed in Base32:

At least 16 symbols

base32topsecret7 16

Figure 22: Programming options of PROTECTIMUS TOTP BURNER app and configuration parameters

Once a configuration is set, the token can be used for TOTP-based two-factor authentication. To get a valid OTP, the user must press the button in the lower right corner of the card. The current one-time password is then shown on the e-Ink display of the tag. Via NFC, it is also possible to read the OTP, as Figure 23 with the corresponding Android app shows.

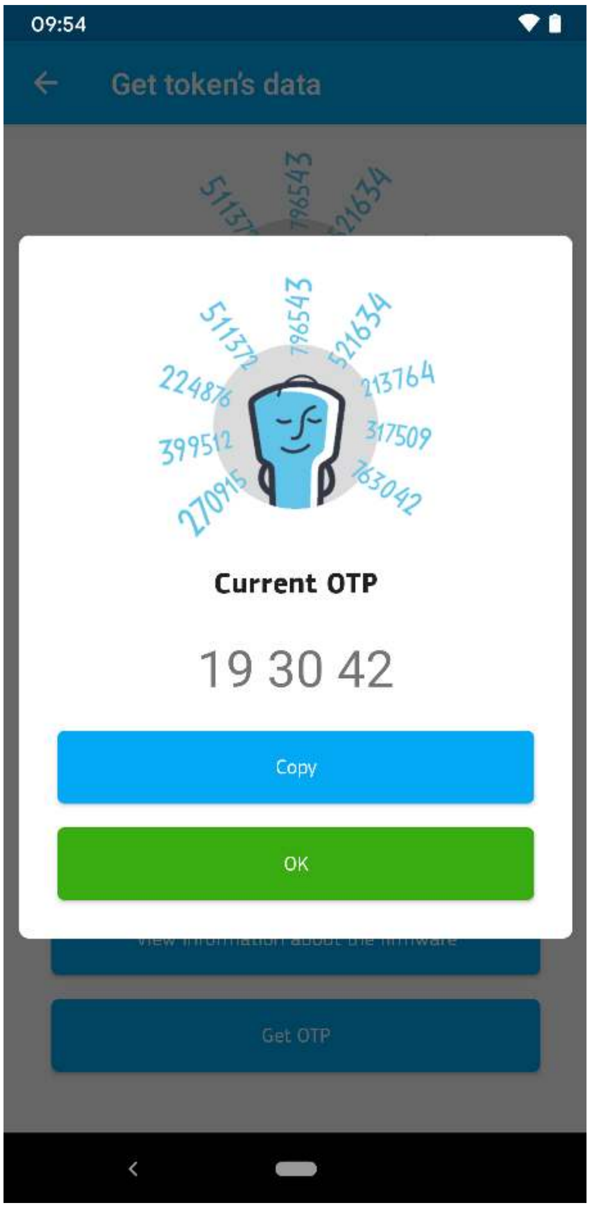


Figure 23: Reading the current one-time password via NFC

3.2 NFC interface

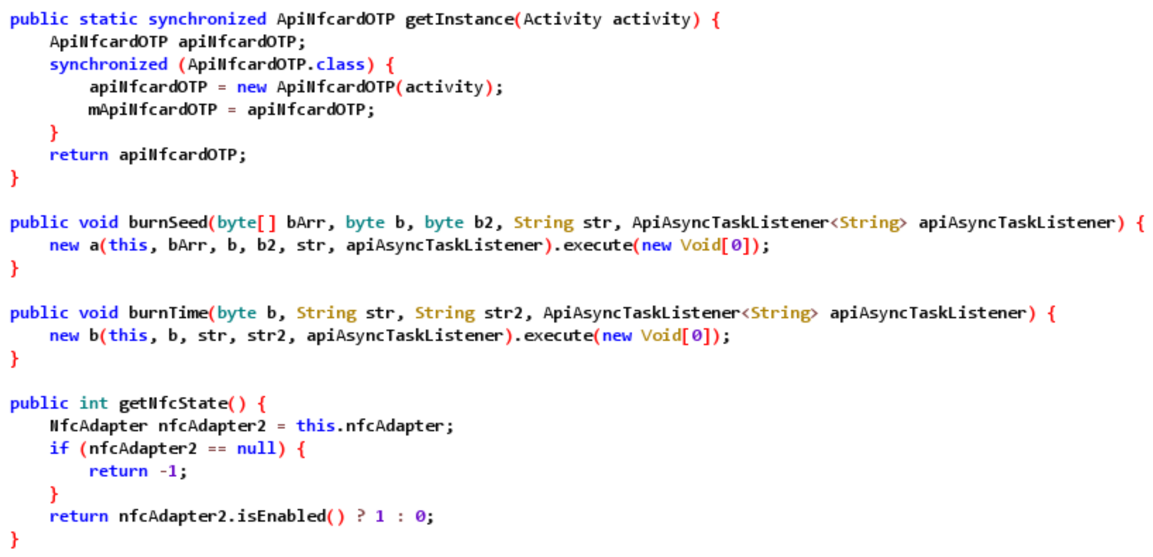
The Protectimus SLIM NFC token can be identified as a typical ISO14443-A tag manufactured by the Austriamicrosystems AG, as the following Proxmark3 output shows.

```
[usb] pm3 --> hf search
 ?  Searching for ISO14443-A tag...
[+]  UID: 3F 10 00 03 23 38 0C
[+] ATQA: 00 44
[+]  SAK: 20 [1]
[+] MANUFACTURER:    Austriamicrosystems AG (reserved) Austria
[+]    JCOP 31/41
[=] -------------------------- ATS --------------------------
[+] ATS: 05 72 00 B0 02 [ 5c 00 ]
[=]      05...............  TL    length is 5 bytes
[=]        72...........  T0    TA1 is present, TB1 is present, TC1 is present, FSCI ↩
    is 2 (FSC = 32)
[=]           00.........  TA1   different divisors are supported, DR: [], DS: []
[=]              B0......  TB1   SFGI = 0 (SFGT = (not needed) 0/fc), FWI = 11 (FWT = 8↩
    388608/fc)
[=]                 02...  TC1   NAD is NOT supported, CID is supported


[+] Valid ISO14443-A tag found
```

Listing 3.1: Tag detection on the Proxmark 3

In order to understand the near-field communication between the hardware token and the PROTECTIMUS TOTP BURNER app, the communication was sniffed using a Proxmark3. Furthermore, the Android app was analyzed using static code analysis of the decompiled code.

Figure 24 exemplarily shows an excerpt of decompiled app code containing the two methods `burnSeed` and `burnTime`, which obviously play a crucial role, as the cryptographic secret, also called seed, and the time are the two important parameters for generating a time-based one-time password.

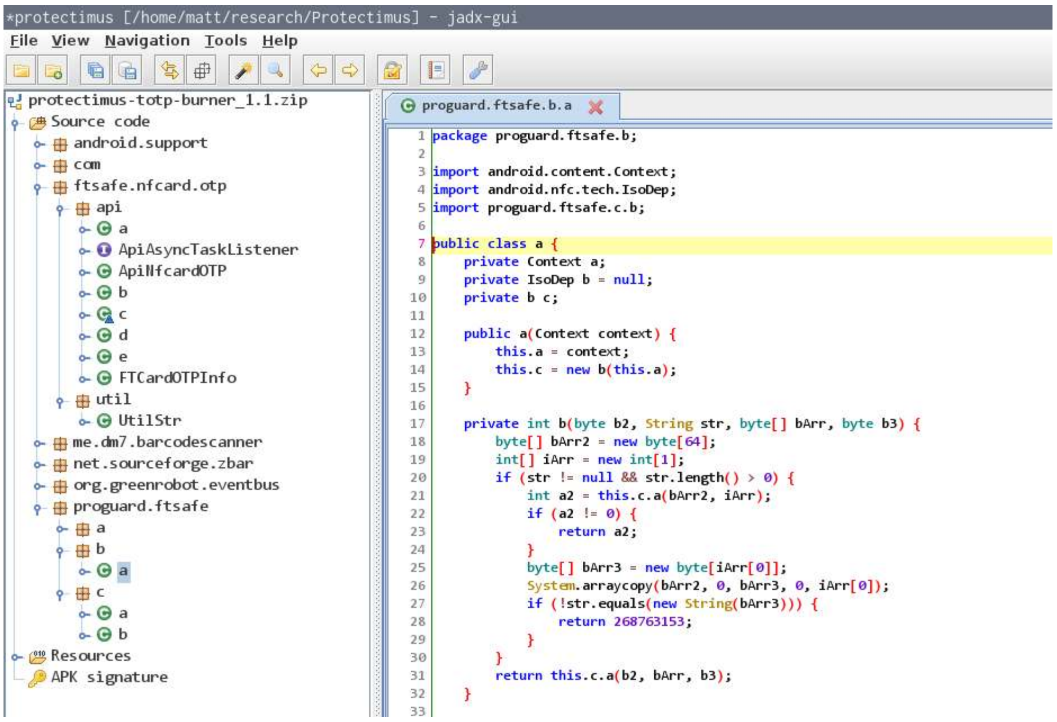


```java
public static synchronized ApiNfcardOTP getInstance(Activity activity) {
    ApiNfcardOTP apiNfcardOTP;
    synchronized (ApiNfcardOTP.class) {
        apiNfcardOTP = new ApiNfcardOTP(activity);
        mApiNfcardOTP = apiNfcardOTP;
    }
    return apiNfcardOTP;
}

public void burnSeed(byte[] bArr, byte b, byte b2, String str, ApiAsyncTaskListener<String> apiAsyncTaskListener) {
    new a(this, bArr, b, b2, str, apiAsyncTaskListener).execute(new Void[0]);
}

public void burnTime(byte b, String str, String str2, ApiAsyncTaskListener<String> apiAsyncTaskListener) {
    new b(this, b, str, str2, apiAsyncTaskListener).execute(new Void[0]);
}

public int getNfcState() {
    NfcAdapter nfcAdapter2 = this.nfcAdapter;
    if (nfcAdapter2 == null) {
        return -1;
    }
    return nfcAdapter2.isEnabled() ? 1 : 0;
}
```

Figure 24: Example of decompiled app code

The interesting code parts of the app were obfuscated via ProGuard. The name `ftsafe` within the code base suggested that the technology used is actually manufactured by FEITAN, a manufacturer that also offers this kind of OTP display cards [13].

Figure 25 exemplarily shows an excerpt of the obfuscated `ftsafe` package.



```java
package proguard.ftsafe.b;

import android.content.Context;
import android.nfc.tech.IsoDep;
import proguard.ftsafe.c.b;

public class a {
    private Context a;
    private IsoDep b = null;
    private b c;

    public a(Context context) {
        this.a = context;
        this.c = new b(this.a);
    }

    private int b(byte b2, String str, byte[] bArr, byte b3) {
        byte[] bArr2 = new byte[64];
        int[] iArr = new int[1];
        if (str != null && str.length() > 0) {
            int a2 = this.c.a(bArr2, iArr);
            if (a2 != 0) {
                return a2;
            }
            byte[] bArr3 = new byte[iArr[0]];
            System.arraycopy(bArr2, 0, bArr3, 0, iArr[0]);
            if (!str.equals(new String(bArr3))) {
                return 268763153;
            }
        }
        return this.c.a(b2, bArr, b3);
    }

```

Figure 25: Obfuscated code of `ftsafe` package

Based on our static code analysis and the sniffed NFC, a Lua script for Proxmark3 [14] was developed for using different implemented functionalities of the Protectimus SLIM NFC token.

Figure 26 shows the general packet format used by the NFC based on our analysis.

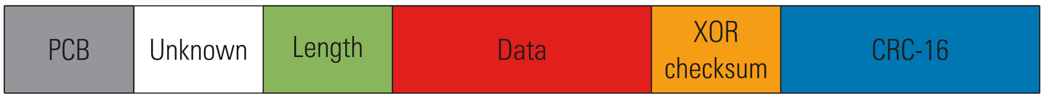


Figure 26: Protectimus SLIM NFC packet format

Besides the CRC-16 checksum related to ISO14443-A, there is also a one-byte XOR checksum for the data bytes.

Figures 27 and 28 exemplarily show two actual packets for reading general token information and reading the current OTP.

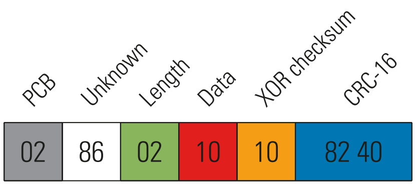


Figure 27: "Read token info" packet

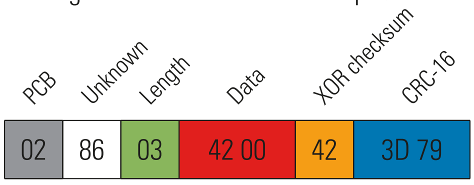


Figure 28: "Read OTP" packet

The following Proxmark3 output exemplarily shows how token information can be read via our developed Lua script.

```
[usb] pm3 --> script run hf_14a_protectimus_nfc -i
[+] executing lua /home/matt/research/proxmark3/client/luascripts/hf_14a_protectimus_nfc
    .lua
[+] args '-i'
Proxmark3 Protectimus SLIM NFC Script v0.8 by Matthias Deeg - SySS GmbH
Perform different operations on a Protectimus SLIM NFC hardware token
[+] Found token with UID 3F10000323380C
[+] Try to read token info
[+] Token info
    Hardware schema:  70
    Firmware version: 10.10
    Hardware RTC:     true
    OTP interval:     30

[+] finished hf_14a_protectimus_nfc
```

And this Proxmark3 output illustrates reading the current one-time password of a Protectimus SLIM NFC token using a Proxmark3.

```
[usb] pm3 --> script run hf_14a_protectimus_nfc -r
[+] executing lua /home/matt/research/proxmark3/client/luascripts/hf_14a_protectimus_nfc
    .lua
[+] args '-r'
Proxmark3 Protectimus SLIM NFC Script v0.8 by Matthias Deeg - SySS GmbH
Perform different operations on a Protectimus SLIM NFC hardware token
[+] Found token with UID 3F10000323380C
[+] Try to read one-time password (OTP)
[+] OTP: 768591

[+] finished hf_14a_protectimus_nfc
```

3.3 Internal card layout

In order to analyze what the Protectimus SLIM NFC token is internally made of, we tried a simple delayering method using an acetone bath and some manual scratching.

Figure 29 shows a Protectimus TOTP token with already partly dissolved layers in an acetone bath.

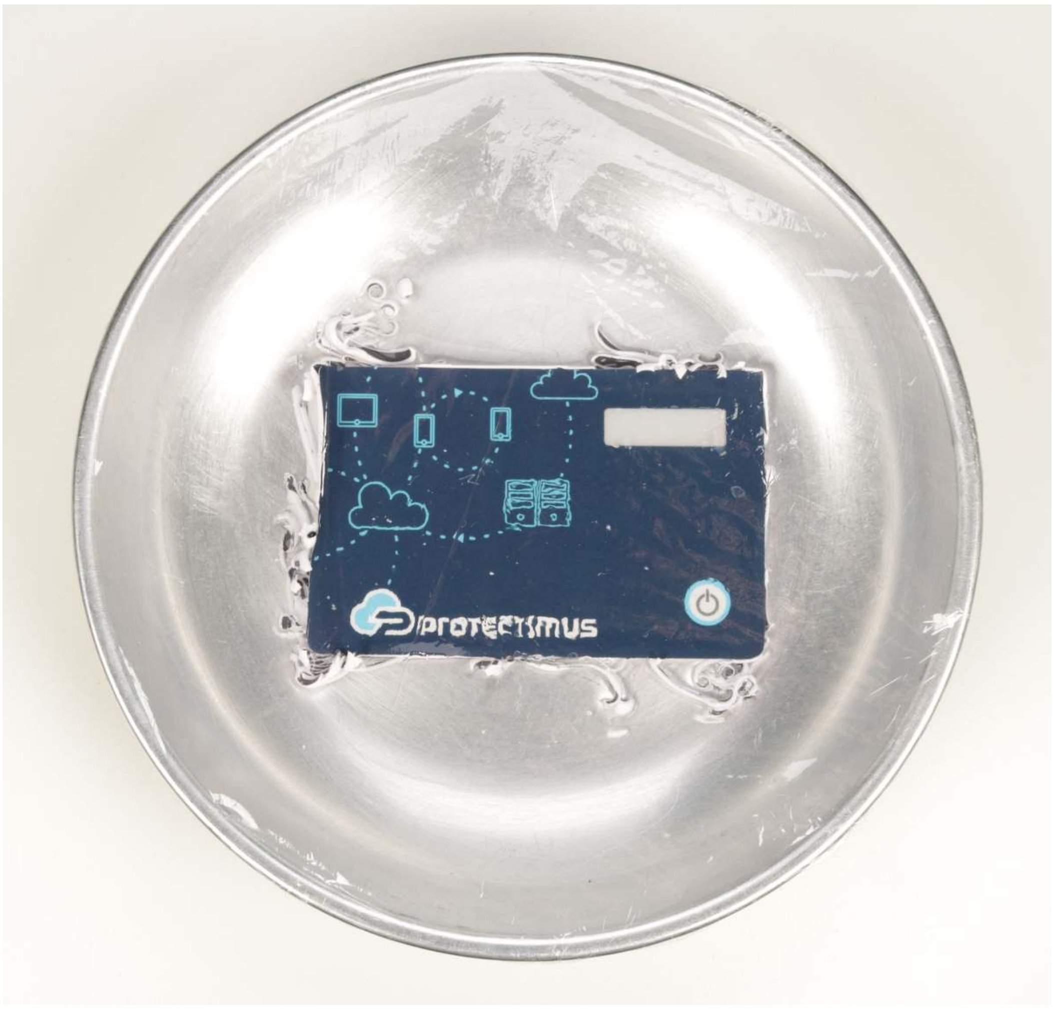


Figure 29: Dissolving some parts in acetone

Figure 30 shows the PCB front side after the acetone bath and manually removing the outer layer.

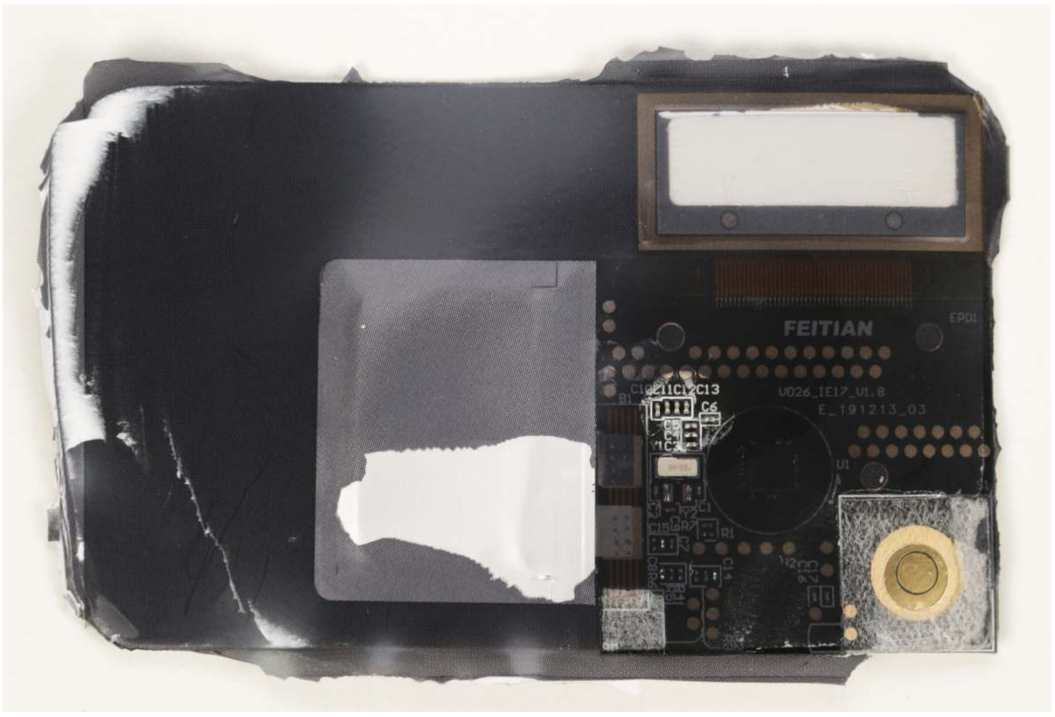


Figure 30: PCB front side after acetone bath

Figure 31 shows the PCB front side after removing a further protective layer via manually scratching it away using some sharp tools. The token still works in this state.

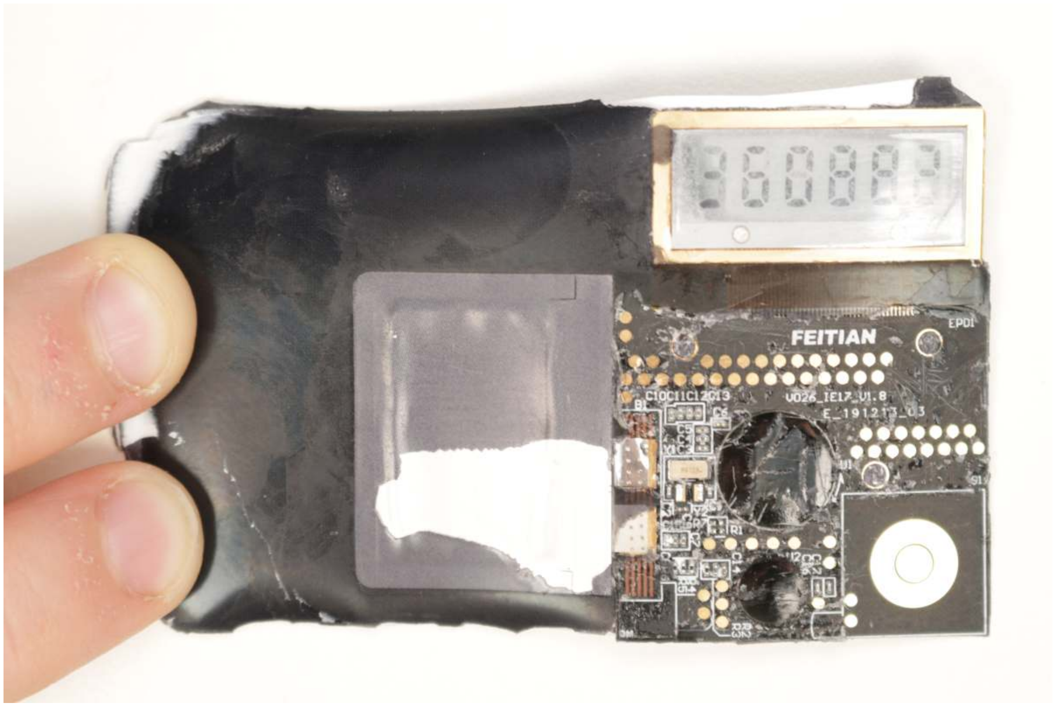


Figure 31: PCB front side after scratching away a protective coating

As the PCB images show, the Protectimus SLIM NFC hardware token is actually produced by FEITAN Technologies Co., Ltd., as we already assumed during the analysis of the Android app. Up to now, we were not able to identify the used chips under the two – probably some kind of epoxy – blobs and also did not perform any further analysis using all those test points.

3.4 To the future – and back

When analyzing the operating mode of the Protectimus SLIM NFC token, we found out that the time used by the hardware token can be set independently from the used cryptographic secret (seed value) for generating time-based one-time passwords without requiring any authentication via NFC.

This enables an attacker with short-time physical access to a Protectimus SLIM token to set the internal real-time clock (RTC) to the future, generate one-time passwords, and then reset the clock to the current time. This allows for generating valid future OTPs without having further access to the hardware token, which is an undesired property of such a TOTP device.

For demonstrating this "time traveler attack" exploiting the described security vulnerability, we developed a Lua script for the Proxmark3 which implements the required operations (also see Section 3.2).

Figure 32 shows our test setup used, consisting of the target device, a Protectimus SLIM NFC token, an Android smartphone with the PROTECTIMUS SLIM TOTP BURNER app for configuring the TOTP token, and a Proxmark3 with our developed Lua script connected to our attacker laptop.

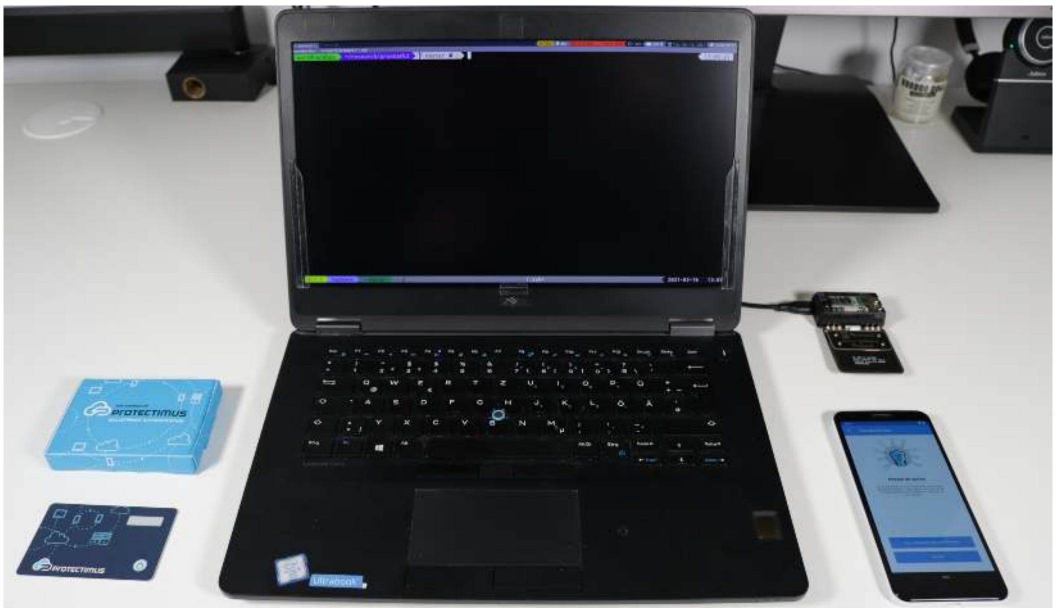


Figure 32: Test setup used for our time traveler attack

The following Proxmark3 output exemplarily shows a successful attack for generating a valid future one-time password for an attacker-chosen point in time against a vulnerable Protectimus SLIM TOTP hardware token.

```
[usb] pm3 --> script run hf_14a_protectimus_nfc -t 2021-03-14T13:37:00+01:00
[+] executing lua /home/matt/research/proxmark3/client/luascripts/hf_14a_protectimus_nfc⤸
    .lua
[+] args '-t 2021-03-14T13:37:00+01:00'
[+] Found token with UID 3F10000323540E
[+] Set Unix time 1615725420
[!] Please power the token and press <ENTER>

[+] The future OTP on 2021-03-14T13:37:00+01:00 (1615725420) is 303831
[+] Set Unix time 1612451460

[+] finished hf_14a_protectimus_nfc
```

We have reported this security issue according to our responsible disclosure program via our security advisory SYSS-2021-007 [15]. The assigned CVE ID for this security vulnerability is CVE-2021-32033.

A proof of concept video [16] demonstrating the described time traveler attack can also be found on our SySS YouTube channel [17].

As TOTP tokens like the Protectimus SLIM NFC are supposed to be used as a further authentication factor in a multi-factor authentication method, the demonstrated time traveler attack only poses a threat in specific real-world scenarios. This could be, for example, a situation when the attacker can gain short-time physical access to the Protectimus SLIM NFC token and furthermore knows the other required authentication factors, for instance user credentials consisting of username and password. This may be the case in scenarios in which third-party service providers are involved, who are provided with time-restricted access to IT systems in order to fulfill their duties.

3.5 Unanswered questions

Concerning the Protectimus SLIM NFC token, we were only able to answer some questions regarding its operation mode. As with the Token2 OTPC-P2, there are still a lot of interesting open questions like:

- Are there any hidden commands not present in the available Android app that can be used via near-field communication?
- Is it possible to generate new TOTP tokens without always powering the device in between, allowing for better time traveler attacks?
- Can the cryptographic secret (seed) be extracted with or without destructing the TOTP token?

4 Conclusion

TOTP hardware tokens are an interesting device class – especially when they support near-field communication which provides an easily accessible attack surface.

Unfortunately, TOTP tokens like the two we analyzed in our case studies are usually a black box to the user and it is not evident from reading publicly available product specifications and documentation how they internally work and whether their operating mode is insecure.

During our research, we were able to find out some more details about the inner workings of the two specific TOTP hardware tokens Token2 OTPC-P2 and Protectimus SLIM NFC and could also identify some security issues. However, there are still many open questions concerning those two devices and the class of TOTP hardware tokens in general which will hopefully be answered and publicly documented in the future.

5 Appendix

In the following sections, the source code of different developed Python scripts is provided.

5.1 OTPC-P2 authentication

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# Copyright 2021 Gerhard Klostermeier (@iiiikarus)
#
# Get the challenge, calculate the response and send it back, performing a full authentication.
# Usage: ./do-authentication.py [key]
# Some info on sending APDUs with nfcpy:
#   https://nfcpy.readthedocs.io/en/latest/modules/tag.html#nfc.tag.tt4.Type4Tag.send_apdu


from sm4 import SM4Key
from nfc.clf import ContactlessFrontend
from nfc.tag.tt4 import Type4TagCommandError
from binascii import hexlify, unhexlify


def main(args):
  # Connect to reader.
  clf = ContactlessFrontend("usb")
  tag = clf.connect(rdwr={'on-connect': lambda tag: False})

  # Get challenge.
  print("[*] Requesting challenge from tag")
  cla = 0x80
  ins = 0x4b
  p1  = 0x08
  p2  = 0x00
  data = unhexlify("00")
  try:
    challenge = tag.send_apdu(cla, ins, p1, p2, data, check_status=True)
  except Type4TagCommandError as ex:
     print(f"[-] Error: No response from tag")
     return 1
  challenge = bytes(challenge)
  print(f"[+] Got challenge {hexlify(challenge).upper()}")

  # Challenge.
  print("[*] Inflating challenge")
  challenge = challenge + b'\x00' * 8
  print(f"[*] Challenge is now: {hexlify(challenge).upper()}")

  # Key.
  key = "8AD206883CA369482AB27182B6E83224"
  if (len(args) > 1):
    key = args[1]
  key_raw = unhexlify(key)
```

```python
  key_sm4 = SM4Key(key_raw)
  print(f"[*] Using key: {hexlify(key_raw).upper()}")

  # Encrypt challenge (SM4).
  print("[*] Encrypting challenge with key using SM4")
  response_raw = key_sm4.encrypt(challenge)
  print(f"[+] Response is: {hexlify(response_raw).upper()}")

  # Send response.
  print("[*] Sending response to tag")
  cla = 0x80
  ins = 0xce
  p1  = 0x00
  p2  = 0x00
  data = response_raw
  auth = tag.send_apdu(cla, ins, p1, p2, data, check_status=False)
  auth = bytes(auth)
  print(f"[*] Got authentication response: {hexlify(auth).upper()}")
  if auth == b'\x90\x00':
    print(f"[+] Authentication was successfull!")
  else:
    print(f"[-] Authentication was not successfull!")

  clf.close()
  return 0


if __name__ == '__main__':
  import sys
  sys.exit(main(sys.argv))
```

Listing 5.1: Perform an authentication on an OTPC-P2 tag

5.2 Collect OTP-P2 challenges

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# Copyright 2020 Gerhard Klostermeier (@iiiikarus)
#
# Get challenges to verify the RNG of the tag.
# Usage: ./collect-challenges.py [challenge count] [retry count]
# Some info on sending APDUs with nfcpy:
#   https://nfcpy.readthedocs.io/en/latest/modules/tag.html#nfc.tag.tt4.Type4Tag.send_apdu


from sm4 import SM4Key
from nfc.clf import ContactlessFrontend, TransmissionError, TimeoutError
from nfc.tag.tt4 import Type4TagCommandError
from binascii import hexlify, unhexlify


def main(args):
  # Default values.
  challenge_count = 10
  max_retry = 3

  # Connect to reader.
  clf = ContactlessFrontend("usb")
  tag = clf.connect(rdwr={'on-connect': lambda tag: False})

  # Get challenge count.
  if (len(args) > 1):
    challenge_count = int(args[1])

  # Get retry count.
  if (len(args) > 2):
    max_retry = int(args[2])

  # Get challenge.
  print("[*] Requesting challenges from tag")
  cla = 0x80
  ins = 0x4b
  p1  = 0x08
  p2  = 0x00
  data = unhexlify("00")
  retry_counter = 0
  for challenge_nr in range(0, challenge_count):
    try:
      challenge = tag.send_apdu(cla, ins, p1, p2, data, check_status=True)
    except (Type4TagCommandError, TransmissionError, TimeoutError) as ex:
      print(f"[-] Error: {ex}. Reconnecting...")
      retry_counter +=1
      tag = clf.connect(rdwr={'on-connect': lambda tag: False})
      if retry_counter >= max_retry:
        print("[-] Too many errors. Abort!")
        return 1
      continue
    challenge = bytes(challenge)
    print(f"[+] Challenge {challenge_nr}: {hexlify(challenge).upper()}")

  clf.close()
  return 0


if __name__ == '__main__':
  import sys
  sys.exit(main(sys.argv))
```

Listing 5.2: Collect challenges of an OTPC-P2 token

5.3 Search for ISO 7816 tag application files

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# Copyright 2020 Gerhard Klostermeier (@iiiikarus)
#
# Search for files/AIDs using the select command.
# Usage: ./find-files.py [file id start] [file id stop] [retry count]
# Some info on sending APDUs with nfcpy:
#   https://nfcpy.readthedocs.io/en/latest/modules/tag.html#nfc.tag.tt4.Type4Tag.send_apdu


from nfc.clf import ContactlessFrontend, TransmissionError, TimeoutError
from nfc.tag.tt4 import Type4TagCommandError
from binascii import hexlify, unhexlify


def main(args):
```

```python
  # Default values.
  file_id_start = 0xb00000000023
  file_id_stop =  0xffffffffffff
  max_reconnect = 50

  # Connect to reader.
  clf = ContactlessFrontend("usb")
  tag = clf.connect(rdwr={'on-connect': lambda tag: False})

  # Get file ID range.
  if (len(args) > 1):
    file_id_start = int(args[1])
  if (len(args) > 2):
    file_id_stop = int(args[2])

  # Get max. reconnect count.
  if (len(args) > 3):
    max_reconnect = int(args[3])

  # Test file/AIDs.
  print("[*] Testing for files/AIDs")
  cla = 0x00
  ins = 0xa4
  p1  = 0x04
  p2  = 0x00
  reconnect_counter = 0
  for file_nr in range(file_id_start, file_id_stop):
    data = file_nr.to_bytes((file_nr.bit_length() + 7) // 8, 'big')
    #print(f"[*] Sending data {hexlify(data).upper()}") # Print verbose.
    try:
      response = tag.send_apdu(cla, ins, p1, p2, data, check_status=True)
    except (Type4TagCommandError, TransmissionError, TimeoutError) as ex:
      #print(f"[-] Error: {ex}") # Print verbose.
      if type(ex) != Type4TagCommandError or str(ex) == "unrecoverable timeout error":
        print(f"[-] Error during command {hexlify(data).upper()}: {ex}. Reconnecting...")
        reconnect_counter +=1
        tag = clf.connect(rdwr={'on-connect': lambda tag: False})
        if reconnect_counter >= max_reconnect:
          print("[-] Too many errors. Abort!")
          return 1
      continue
    response = bytes(response)
    print(f"[+] Got response for data {hexlify(data).upper()}: {hexlify(response).upper()}")

  clf.close()
  return 0


if __name__ == '__main__':
  import sys
  sys.exit(main(sys.argv))
```

Listing 5.3: Search for applications (files) on an ISO 7816 tag

References

[1] HOTP: An HMAC-Based One-Time Password Algorithm (RFC 4226), D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, O. Ranen, `https://tools.ietf.org/html/rfc4226`, December 2005 1

[2] TOTP: Time-Based One-Time Password Algorithm (RFC 6238), D. M'Raihi, S. Machani, M. Pei, J. Rydell, `https://tools.ietf.org/html/rfc6238`, May 2011 1

[3] Google Authenticator, Google LLC, `https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2` 2021 1

[4] PyOTP – The Python One-Time Password Library, Andrey Kislyuk, `https://pyauth.github.io/pyotp/`, 2021 1

[5] Proxmark3, RFID Research Group, `https://github.com/RfidResearchGroup/proxmark3`, 2021

[6] OTPC-P2, Token2, `https://www.token2.com/shop/product/token2-otpc-p2-programmable-card-with-restricted-time-sync-nonbranded`, 2021 2

[7] NFC Burner app for Windows, Token2, `https://www.token2.com/site/page/windows-nfc-burner`, 2021 3

[8] NFC Burner 2 (Android app), Token2, `https://play.google.com/store/apps/details?id=com.token2.nfcburner2`, 2021 3

[9] AN10927M – IFARE product and handling of UIDs, NXP Semiconductors, `https://www.nxp.com/docs/en/application-note/AN10927.pdf`, 2021 5

[10] Smart card application protocol data unit, Wikipedia, `https://en.wikipedia.org/wiki/Smart_card_application_protocol_data_unit`, 2021 4

[11] Twitter account of Philippe Teuwen, Twitter, `https://twitter.com/doegox`, 2021 8

[12] PROTECTIMUS TOTP BURNER, Protectimus Solutions LLP, `https://play.google.com/store/apps/details?id=com.protectimus.totpburner.nfc&hl=en_US&gl=US`, August 28, 2020 22

[13] FEITAN OTP display cards, FEITAN Technologies Co., `https://play.google.com/store/apps/details?id=com.protectimus.totpburner.nfc&hl=en_US&gl=US`, 2021 25

[14] PROTECTIMUS SLIM NFC Lua script for Promark3 Matthias Deeg, SySS GmbH, `https://github.com/SySS-Research/protectimus-slim-proxmark3`, 2021 26

[15] SySS Security Advisory SYSS-2021-007, Matthias Deeg, SySS GmbH, `https://www.syss.de/fileadmin/dokumente/Publikationen/Advisories/SYSS-2021-007.txt`, 2021 31

[16] To the Future and Back: Attacking a TOTP Hardware Token (Protectimus SLIM NFC), Matthias Deeg, SySS GmbH, `https://www.youtube.com/watch?v=C0pM6TIyvXI`, 2021 31

[17] SySS Pentest TV, SySS GmbH, `https://www.youtube.com/SySSPentestTV`, 2021 31