Writing your own data to a Disney Infinity NFC toy

As discussed in the case study, by knowing the algorithm used to set the read/write password (key A), we can interoperably read/write our own data to an Infinity NFC toy. Read New interoperability for Disney Infinity NFC toys for details on the algorithm, and a demonstration video showing it in use.

By request, this page documents a workflow similar to the demo video, using standard software available on any Mac or Linux computer.

> **Use at your own risk!**
>
> Consider writing your own data only to toys you no longer wish to use with the game.

Prerequisites

You'll need libnfc. The last stable release, 1.7.1, is fine. Linux users should be able to install it using their package manager. Mac users may be able to use macports or homebrew, or compile it natively.

You'll also need a libnfc-supported NFC reader. These examples used the Identiv SCL3711.

You'll need the `infsha.py` Python 2 program, which is an example implementation of the key generation algorithm, listed in New interoperability for Disney Infinity NFC toys.

We'll be working in hexadecimal, which is easier for people to read, but the tools use the "MIFARE Dump" format. Download the `eml2mfd.py` and `mfd2eml.py` programs from the MIFARE Classic Tool source control so we can convert data back and forth.

You may also want the `emlinsert.py` Python 2 program, which makes it easier to create the file necessary to write custom data to an NFC tag, listed at the bottom of this page.

Identifying an Infinity NFC toy

Here's the output of the libnfc standard tool, `nfc-list`, when run against the "Kanan Jarrus" figure used in the demo. (This is the same as for any MIFARE tag.)

```bash
securitoy$ nfc-list
nfc-list uses libnfc 1.7.1
NFC device: SCM Micro / SCL3711-NFC&RW opened
1 ISO14443A passive target(s) found:
ISO/IEC 14443A (106 kbps) target:
    ATQA (SENS_RES): 00  44
       UID (NFCID1): 04  5d  60  ba  4f  48  80
      SAK (SEL_RES): 09
securitoy$
```

An ATQA of `00 44` with an SAK of `09` means a Disney Infinity NFC toy.

The UID of this figure is `04 5d 60 ba 4f 48 80`.

Generating the keys for an Infinity NFC toy

Pass the UID to the `infsha.py` program:

```bash
securitoy$ ./infsha.py 045d60ba4f4880
88f8a009843d
securitoy$
```

That is the key A used for all five sectors, necessary to read or write the Infinity NFC toy, generated algorithmically, instead of sniffing.

Reading the data from an Infinity NFC toy

To use this key with the libnfc MIFARE tools, it needs to be in a MIFARE Dump format. The sample algorithm implementation can output a hexadecimal, Proxmark emulator format (EML), which we can convert.

```bash
securitoy$ ./infsha.py 045d60ba4f4880 -eml
045d60ba4f4880000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
88f8a009843d0000000088f8a009843d
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
88f8a009843d0000000088f8a009843d
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
88f8a009843d0000000088f8a009843d
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
88f8a009843d0000000088f8a009843d
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
88f8a009843d0000000088f8a009843d
securitoy$
```

That is the same key A (also B), plus the UID, written "in place", where they'd be in a hexadecimal dump of the tag.

Save that to `kanan-keys.eml` and convert it to a MIFARE Dump file using `eml2mfd.py`.

```bash
securitoy$ ./infsha.py 045d60ba4f4880 -eml > kanan-keys.eml
securitoy$ ./eml2mfd.py kanan-keys.eml kanan-keys.mfd
securitoy$
```

Read the contents of the tag using the libnfc standard tool `nfc-mfclassic`, using the generated keys, saving it into `kanan-dump.mfd`. (This is the same as for any MIFARE tag.)

```bash
securitoy$ nfc-mfclassic r a kanan-dump.mfd kanan-keys.mfd
NFC reader: SCM Micro / SCL3711-NFC&RW opened
Found MIFARE Classic card:
ISO/IEC 14443A (106 kbps) target:
    ATQA (SENS_RES): 00  44
       UID (NFCID1): 04  5d  60  ba  4f  48  80
      SAK (SEL_RES): 09
Guessing size: seems to be a 320-byte card
Reading out 20 blocks |....................|
Done, 20 of 20 blocks read.
Writing data to file: kanan-dump.mfd ...Done.
securitoy$
```

Convert the MIFARE Dump to hex to view it, using `mfd2eml.py` and the standard Unix tool `cat`.

```bash
securitoy$ ./mfd2eml.py kanan-dump.mfd kanan-dump.eml
securitoy$ cat kanan-dump.eml
045d60ba4f4880894400c20000000000
4932cc857e7c84485cea496d77f84183
162451ae6b03509e5663b3f648d16298
88f8a009843d17878e0088f8a009843d
b5452d09d628248c92c9f6910e87e435
b5452d09d628248c92c9f6910e87e435
00000000000000000000000000000000
88f8a009843d7787880088f8a009843d
0994ca9e9bcd2eba9e337f0b0b271862
b92ff78dd1af8f7cfaed9f8c3ee42f76
00000000000000000000000000000000
88f8a009843d7787880088f8a009843d
49eb1e6683681b27c026626f614f5131
49eb1e6683681b27c026626f614f5131
00000000000000000000000000000000
88f8a009843d7787880088f8a009843d
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
88f8a009843d7787880088f8a009843d
securitoy$
```

(This is similar to the data we saw on the toy in the case study.)

Extract the writable, 192 bytes from the hex output and evaluate it, using the standard Unix `sed`, `xxd` and `file` tools.

```bash
securitoy$ sed -n -e 5,7p -e 9,11p -e 13,15p -e 17,19p kanan-dump.eml | xxd -r -ps > kanan-192.bin
securitoy$ file kanan-192.bin
kanan-192.bin: data
securitoy$
```

Since this is the encrypted gameplay data from the toy, it's just "data", and we can't do anything further with it.

Writing our own data to an Infinity NFC toy

Create your own data to write, up to 192 bytes, and save it. For example, this is similar text to the demo video, saved as `nfctoys-inf.txt`.

```
nfc.toys

You bought the toys, including the NFC chip inside!

Put your own data on your own toys!

Presented at HOPE 2018 in NYC, July 22.

nfc.toys
```

Convert your data to hex, in a similar structure to the Proxmark emulator format, using the standard Unix `xxd` tool.

```bash
securitoy$ xxd -c 16 -l 192 -ps nfctoys-inf.txt
6e66632e746f79730a0a596f7520626f
756768742074686520746f79732c2069
6e636c7564696e6720746865204e4643
206368697020696e73696465210a0a50
757420796f7572206f776e2064617461
206f6e20796f7572206f776e20746f79
73210a0a50726573656e746564206174
20484f5045203230313820696e204e59
432c204a756c792032322e0a0a6e6663
2e746f7973
securitoy$
```

Open up the `kanan-keys.eml` file, replace the `00000000000000000000000000000000` blocks with your hex content, and save it as a new file, such as `kanan-new.eml`. Remember that sector 0 is write-only, so you're starting on line 5 (block 4, sector 1), then line 6, line 7, skipping line 8 (the sector trailer with the key A), and so on. For lines that aren't 32 characters long, pad them out with zeros. (This is the same as for any MIFARE tag.)

```
045d60ba4f4880000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
88f8a009843d0000000088f8a009843d
6e66632e746f79730a0a596f7520626f
756768742074686520746f79732c2069
6e636c7564696e6720746865204e4643
88f8a009843d0000000088f8a009843d
206368697020696e73696465210a0a50
757420796f7572206f776e2064617461
206f6e20796f7572206f776e20746f79
88f8a009843d0000000088f8a009843d
73210a0a50726573656e746564206174
20484f5045203230313820696e204e59
432c204a756c792032322e0a0a6e6663
88f8a009843d0000000088f8a009843d
2e746f79730000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
88f8a009843d0000000088f8a009843d
```

You can also use the `emlinsert.py` Python 2 program to do the same thing automatically.

```bash
securitoy$ xxd -c 16 -l 192 -ps nfctoys-inf.txt > nfctoys-inf.eml
securitoy$ ./emlinsert.py kanan-keys.eml nfctoys-inf.eml > kanan-new.eml
securitoy$
```

Convert the EML file to MIFARE Dump using `eml2mfd.py`, then write the contents of the file using the libnfc standard tool `nfc-mfclassic`, using the generated keys. (This is the same as for any MIFARE tag.)

```bash
securitoy$ ./eml2mfd.py kanan-new.eml kanan-new.mfd
securitoy$ nfc-mfclassic w A kanan-new.mfd kanan-keys.mfd
NFC reader: SCM Micro / SCL3711-NFC&RW opened
Found MIFARE Classic card:
ISO/IEC 14443A (106 kbps) target:
    ATQA (SENS_RES): 00  44
       UID (NFCID1): 04  5d  60  ba  4f  48  80
      SAK (SEL_RES): 09
Guessing size: seems to be a 320-byte card
Writing 20 blocks |xxfailed to write trailer block 3
x...failed to write trailer block 7
x...failed to write trailer block 11
x...failed to write trailer block 15
x...failed to write trailer block 19
x|
Done, 12 of 20 blocks written.
securitoy$
```

Read the data back out to verify it, using `nfc-mfclassic`, `mfd2eml.py`, `sed`, `xxd`, `file`, and `cat`. (This is the same as for any MIFARE tag.)

```bash
securitoy$ nfc-mfclassic r a kanan-verify.mfd kanan-keys.mfd
NFC reader: SCM Micro / SCL3711-NFC&RW opened
Found MIFARE Classic card:
ISO/IEC 14443A (106 kbps) target:
    ATQA (SENS_RES): 00  44
       UID (NFCID1): 04  5d  60  ba  4f  48  80
      SAK (SEL_RES): 09
Guessing size: seems to be a 320-byte card
Reading out 20 blocks |....................|
Done, 20 of 20 blocks read.
Writing data to file: kanan-verify.mfd ...Done.
securitoy$ ./mfd2eml.py kanan-verify.mfd kanan-verify.eml
securitoy$ sed -n -e 5,7p -e 9,11p -e 13,15p -e 17,19p kanan-verify.eml | xxd -r -ps > kanan-verify.bin
securitoy$ file kanan-verify.bin
kanan-verify.bin: ASCII English text
securitoy$ cat kanan-verify.bin
nfc.toys

You bought the toys, including the NFC chip inside!

Put your own data on your own toys!

Presented at HOPE 2018 in NYC, July 22.

nfc.toys
securitoy$
```

Restore the original contents of the toy using `nfc-mfclassic`. (This is the same as for any MIFARE tag.)

```bash
securitoy$ nfc-mfclassic w A kanan-dump.mfd kanan-keys.mfd
NFC reader: SCM Micro / SCL3711-NFC&RW opened
Found MIFARE Classic card:
ISO/IEC 14443A (106 kbps) target:
    ATQA (SENS_RES): 00  44
       UID (NFCID1): 04  5d  60  ba  4f  48  80
      SAK (SEL_RES): 09
Guessing size: seems to be a 320-byte card
Writing 20 blocks |xxfailed to write trailer block 3
x...failed to write trailer block 7
x...failed to write trailer block 11
x...failed to write trailer block 15
x...failed to write trailer block 19
x|
Done, 12 of 20 blocks written.
securitoy$
```

emlinsert.py

```python
#!/usr/bin/python

## emlinsert.py - Insert hex data into other hex data
##
## Written in 2018 by Vitorio Miliano
##
## To the extent possible under law, the author has dedicated all
## copyright and related and neighboring rights to this software to
## the public domain worldwide.  This software is distributed without
## any warranty.
##
## You should have received a copy of the CC0 Public Domain
## Dedication along with this software.  If not, see
## <http://creativecommons.org/publicdomain/zero/1.0/>.

import os, sys

if __name__ == '__main__':
    if not len(sys.argv) > 2:
        raise ValueError('usage: {} MFDFILE DATAFILE'.format(sys.argv[0]))

    if not os.path.exists(sys.argv[1]) or not os.path.exists(sys.argv[2]):
        raise ValueError('usage: {} MFDFILE DATAFILE'.format(sys.argv[0]))

    MFDFILE = []
    with open(sys.argv[1]) as f:
        for line in f:
            MFDFILE.append(line)

    DATAFILE = []
    with open(sys.argv[2]) as f:
        for line in f:
            DATAFILE.append(line)

    blocks = [x for x in range(4, len(MFDFILE)) if (x+1) % 4]

    NEWHEX = []
    b = 0
    for idx, x in enumerate(MFDFILE):
        if idx in blocks:
            if b < len(DATAFILE):
                NEWHEX.append(DATAFILE[b].strip() + (32-len(DATAFILE[b].strip()))*'0' + '\n')
            else:
                NEWHEX.append('0'*32+'\n')
            b = b + 1
        else:
            NEWHEX.append(x)

    print ''.join(NEWHEX).strip()
```