Quarkslab's website

SOCIAL

- atom feed
- twitter
- github

CATEGORIES

- Android
- Android, ReverseEngineering
- Challenge
- Cryptography
- Development
- Exploitation
- Fuzzing
- Hardware
- Hardware, ReverseEngineering
- Kernel Debugging
- Life at Quarkslab
- Maths
- Obfuscation
- PenTest
- Program Analysis
- Programming
- ReverseEngineering
- Software
- Vulnerability

TAGS

RFID: New Proxmark3 Tear-Off Features and New Findings

Date Thu 19 November 2020 By Philippe Teuwen Christian Herrmann Category Hardware. Tags hardware NFC RFID proxmark3 EEPROM tear-off

Latest news from the Proxmark3 world, crunchy bits included...

Introduction

It's often easier to decide to write a blog post when it's about a security issue unveiled fully by ourselves or a tool developed entirely by ourselves. Nevertheless, we are not in an ivory tower and some tools and research are done across a much wider community. This is especially true in the Proxmark community where a little open source RFID/NFC hacking tool designed by Jonathan Westhues 13 years ago is still witnessing massive software development nowadays.

There is so much happening in the Proxmark3 world that we think this also deserves blog posts now and then, with the particularity that they build on the work of many other persons as well. I, Phil, contribute mostly on my free time, as this project is a personal hobby, but it's also a professional tool used during our trainings and CTFs, amongst other things.

It was natural to invite *Iceman* to collaborate on such blog posts as well, as an historic legend ;) and as we have been working together for a while on these matters.

For this first joint work, we will present some interesting tear-off tools and their application on EM4305 chips.

Tear-off physical concepts were introduced in the previous blog post *EEPROM: When Tearing-Off Becomes a Security Issue* [1] which we strongly recommend to read first if you're not yet familiar with the concepts.

Proxmark3 Tear-Off Features

Here is the current situation [2] of the tools in the Proxmark3 RRG repo.

ATA5577C

We already covered the command `lf t55xx dangerraw` in the previous blog post [1], implementing tear-off against ATA7755C chips.

Mikron "Ultralight"

Since then, Nahuel Grisolía (@cintainfinita) and Federico Gabriel Ukmar (@federicoukmar) published a thesis [3] about tear-off attacks against MIK640M2D (K5016XC1M2H4) [4], a variant of MIFARE Ultralight developed by Mikron. Indeed this variant didn't do anything to protect its five OTP (One-Time Programmable) blocks of 32 bits and a tear-off between internal erase and write operations is enough to set OTP bits back to zeroes. The corresponding command is `hf mfu otptear`. NXP MIFARE Ultralight tags are immune to this attack.

As a side note, other Ultralight variants such as my-d move [5] tags by Infineon and F8213 NFC-Forum Type 2 [6] tags by Shanghai Feiju Microelectronics Co also have OTP bits and no protection against tear-off but... on these tags, the default value of an erased word is `FFFFFFFF`, so it's impossible to reset bits back to zero with this method. Simply inverting the logic of the stored bits in EEPROM provided an efficient security protection against tear-off! On the other side, if such a tear-off happens by accident, you may burn all the OTPs in the word being updated... Some other cards such as ST SRI512 simply skip the erase operation on OTP bits, which is probably easier and safer.

Warning: one might attempt to tear off the MIK640M2D Lock bytes from block 2 as well and yes, it works, but a bit too well... Block 2 also contains the BCC1 byte using during the ISO14443-3 anti-collision phase as UID checksum and if it's zeroed, readers will disregard the card response.

New Generic Tear-Off Support

Since tear-off is such a new vector in the Proxmark3 world, we decided it needed a better generic support than the dedicated existing commands. To be able to experiment tear-off on more types of tags, there is now a `hw tearoff` command available to set a delay and to schedule a tear-off event during the next command supporting such tear-off. The list of commands grows every day and now you can experiment tear-off against ISO14443-A, ISO14443-B and ISO15693 raw commands as well as iClass and EM4x05 writes.

If nevertheless you want to execute a tear-off on a command not yet ready for it, adding support becomes as easy as adding a couple of lines in the code, just after the command triggering an EEPROM operation has been sent to the tag.

```
if (tearoff_hook() == PM3_ETEAROFF) {
    // tear-off occurred. Clean stuff and return
} else {
    // do as usual: read reply etc
}
```

This is how we could experiment with the EM4305 chip before writing more specific and automated tools.

Select a Target

Why the EM4305? We have to tell you, it all started by accident: a vendor was supposed to deliver FDX-A ATA5577C glass transponders but upon inspection, the transponders appeared to be EM4305 chips. Our curiosity did the rest.

EM4305

EM4305 [7] chips are manufactured by EM Microelectronic-Marin SA since 2007 and are targeting the animal identification industry.

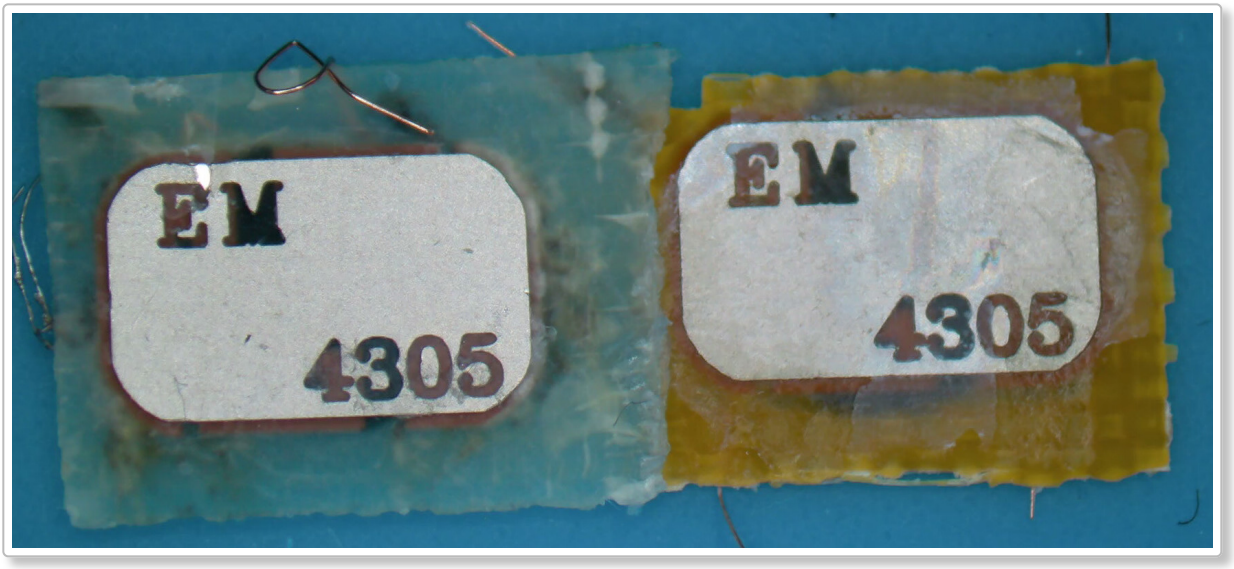
They are a bit similar to ATA5577C as you can configure them to emulate various LF ID tags and most cloners such as the one presented in the previous blog post support indiscriminately ATA5577C and EM4305 as blank tags.

They have a so-called *Protection Words* feature which fulfills the prerequisites for being an interesting target for tear-off experiments: one can trigger an EEPROM operation by changing the *Protection Words*, but without being in full control of the final value, as one is not supposed to be able to clear a protection already set.

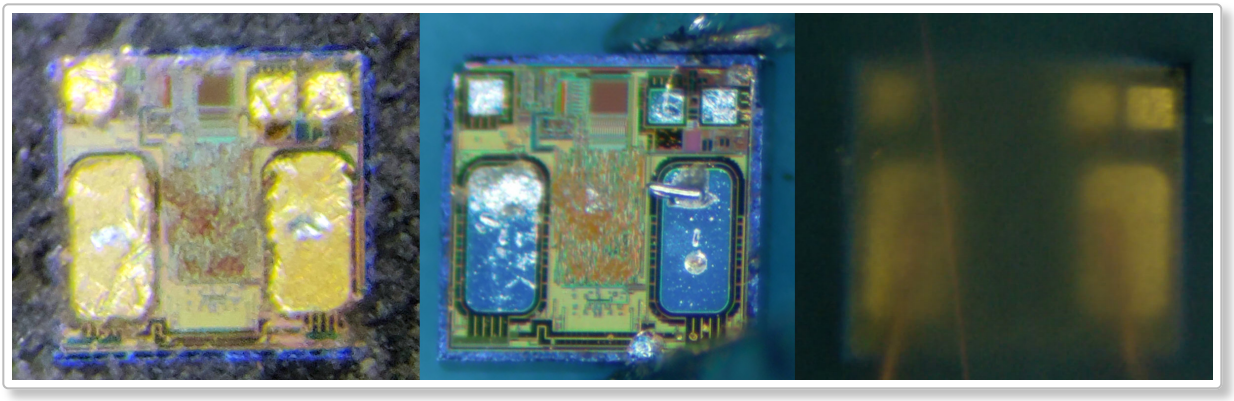
But first came the question of whether the chips we used were genuine or counterfeit, as it happens quite frequently in the RFID world. It's hard to tell for sure that they were genuine but at least we didn't see any evidence of counterfeiting:

- The various response timings were in accordance with the datasheet;
- We successfully reproduced our tests on glass transponders, key fobs, clamshells and cards from at least 6 different sources;
- We decapped a card chip and a key fob chip and compared them to a glass chip, still visible in its glass capsule, see below.

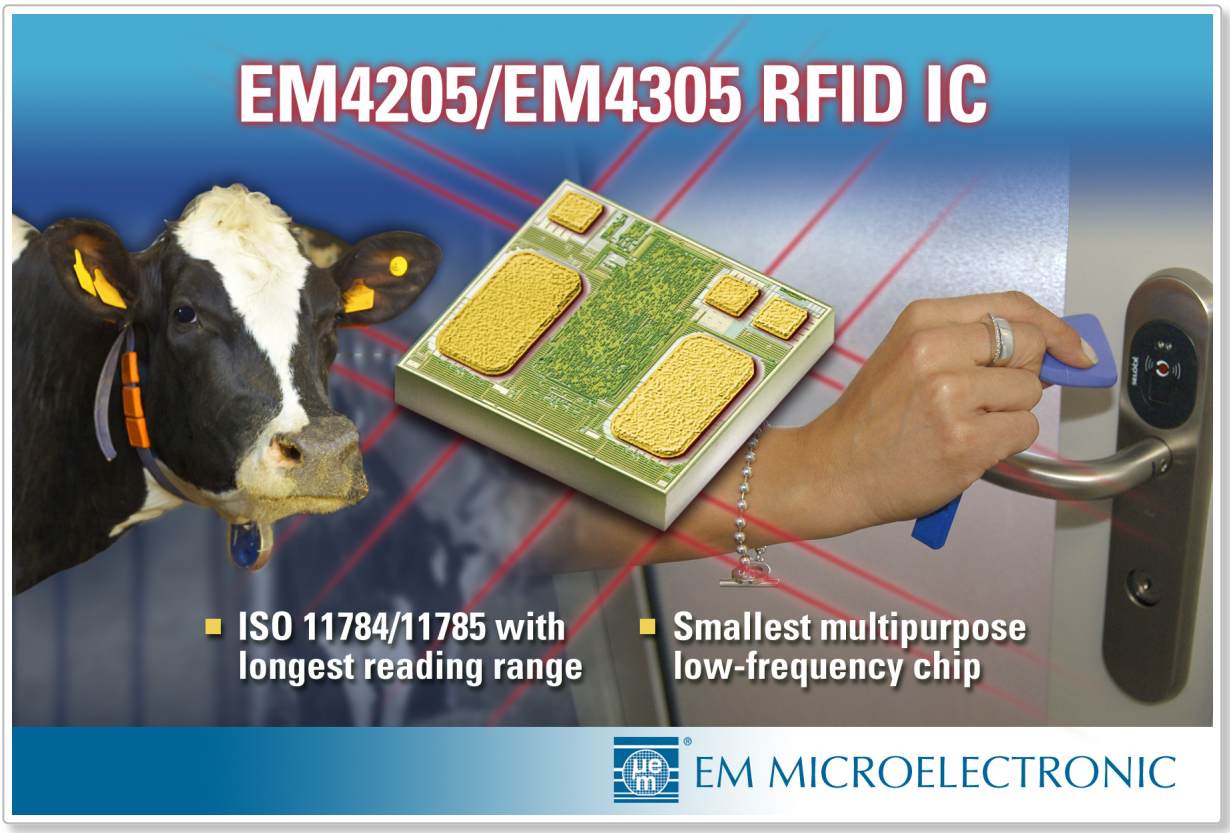
Key fob and card chips, backside of the support:



Key fob, card and glass chips:



They are all alike, sharing the same layout, and resemble quite a lot to the rendering visible on this commercial:



EM4305 Protection Words

EM4305 chips feature 16 words of 4 bytes, some of which are used as manufacturer data, UID, password, configuration block and protection words. We can read the memory of such a tag with the command `lf em 4x05_dump`.

```
[usb] pm3 --> lf em 4x05_dump
[=] Found a EM4305 tag

[=] Addr | data     | ascii |lck| info
[=] -----+----------+-------+---+-----
[=]   00 | 0000E052 | ...R  |   | Info/User
[=]   01 | 63A82630 | c.&0  | x | UID
[=]   02 |          |       |   | Password   write only
[=]   03 | 0000556C | ..Ul  |   | User
[=]   04 | 0001C258 | ...X  |   | Config
[=]   05 | 55564755 | UVGU  |   | User
[=]   06 | 95555556 | .UUV  |   | User
[=]   07 | A5A699A6 | ....  |   | User
[=]   08 | 00000000 | ....  |   | User
[=]   09 | 00000000 | ....  |   | User
[=]   10 | 00000000 | ....  |   | User
[=]   11 | 00000000 | ....  |   | User
[=]   12 | 00000000 | ....  |   | User
[=]   13 | 00000000 | ....  |   | User
[=]   14 | 00008002 | ....  |   | Lock       active
[=]   15 | 00000000 | ....  |   | Lock
```

By default, all words are readable/writable except the *Password Word* which cannot be read out and the *UID Word* which cannot be overwritten. In this example, bits in the *Protection Word* at address 14 indicate which pages are locked against write operations. *Protection Word* bits are OTP (One-Time Programmable) bits so they can be set but they cannot be cleared, so locked words will remain locked forever. The first 14 bits (`pr0` to `pr13`) represent the first 14 words. We note that the second bit `pr1` is set, locking indeed the *UID Word*. The 15th bit - called `pr14` - locks both words 14 & 15 (word 15 is also a *Protection Word* as we'll see below). If set, it forbids further modifications of the *Protection Words* themselves. So the configuration can't be changed anymore and unlocked

words will remain unlocked forever. The 16th bit - `pr15` - indicates which word is the active *Protection Word*. Remaining bits `pr16` to `pr31` are not used but are OTP too.

Word 14 being `00008002`, we get the following interpretation.

```
pr31..                      ..pr0
xxxxxxxxxxxxxxx1pppppppppppppppp
                 ^^^^^^^^^^^^^^ => pr0..pr13 locking words 0..13
                ^               => pr14 locking words 14 & 15
               ^                => pr15 = 1 for the active configuration

00000000000000001000000000000010
                             ^  => Word 1 (UID Word) is locked
               ^                => Word 14 is the active configuration
```

EM4305 Tear-Off Protection

The chip has a feature designed specifically to avoid tear-off events: *Protection Word* is shared across words 14 & 15. As seen above, the 16th bit - `pr15` - indicates which word is the active one. This explains the `00008002` value in the dump: *Protection Word 14* is the active one and protects the *UID Word*. Word 15 is zeroed and ready to be programmed with a new Protection word.

Protection Words 14 & 15 cannot be directly written with a WRITE command. To modify their content, a PROTECT command must be issued. Upon reception of such PROTECT command, the EM4305 will do the following:

- check if a password must be provided. If yes, check if a LOGIN command has been issued with a correct password;
- check which *Protection Word* is active, i.e. which has `pr15` bit set;
- write in the other *Protection Word* the new value, being the bitwise OR between the content provided in the PROTECT command and the existing content of the active *Protection Word* (as we cannot clear bits, we can only set bits);
- erase the active *Protection Word*.

For example, after issuing a PROTECT(`00000001`) we will end up with these values.

```
[=]   14 | 00000000 | ....  |    | Lock
[=]   15 | 00008003 | ....  |    | Lock       active
```

By this mechanism of shadow Protection word, we should never end up with a *Protection Word* with fewer bits set than before the command.

Abstract from the datasheet [7]:

> *The above implementation, using two physical words in a read/write EEPROM to represent a single Protection Register, was chosen as an additional security feature. This double buffered mechanism caters to the fact an EEPROM-write operation internally generates an erase-to-zero operation followed by the actual write operation. Should the operation be interrupted for any reason (e.g. tag removal from the field) the double buffer scheme ensures that no unwanted "0"-Protection Bits (i.e unprotected words) are introduced.*

Could we still do some tear-off and defeat that mechanism?

Tear-Off in Practice

Basics

Completely unnecessary paragraph as you just (re-)read the previous blog post [1] on EEPROM tearing off, right?

Changing the content of an EEPROM requires two operations: first an erase phase then a write phase. On RFID tags, EEPROM can typically be erased on a small number of bytes at once. Two possibilities: all bits of an erased block are zeroed and a write operation can only set bits, or all bits are set to 1 when erased and a write can only clear bits.

The key ingredient of a successful tear-off is to find the right timings to interrupt either the EEPROM erase phase or the write phase, depending on the specific situation and desired goal.

For example, if we trigger a tear-off during a write that is setting bits, the "1" bits will be partially written, i.e. the transistor gates will be partially loaded with electrons and some bits might be seen indeed as "1", while some bits will still be seen as "0". The value for a half-written bit can even fluctuate and sometimes be seen as a "0" and sometimes as a "1".

Field Reconnaissance

So the goal will be to end up with a new valid *Protection Word* (`pr15` set) but lock bits cleared (`pr1` must be cleared).

```
pr31..                        ..pr0
00000000000000001000000000000010
=>
00000000000000001000000000000000
                               ^  => UID Word becomes unlocked
```

We have seen that the chip first writes the new value then erases the old one. There is no interesting scenario in interrupting the erase phase such as for `hf mfu otptear`. But if we interrupt the write, we may end up with fewer bits being seen as set to "1".

Several elements must be figured out for the tear-off to be successful:

1. What is the default value of an erased EEPROM word?
2. In case we tear off between write (of one word) and erase (of the other word) operations, we will end up with two valid *Protection Words*. Which one will be considered as the active one?
3. What happens if we issue a PROTECT(`00000000`) or a PROTECT that doesn't set more bits than what's already set?
4. Will the write be considered as valid only because its `pr15` bit is set or because its whole content is the expected value?
5. Is UID only protected by the `pr1` bit lock or is it read-only anyway?

Let's examine each point.

1. We have seen that it's not possible to clear bits by tear-off if the erased EEPROM state is `FFFFFFFF`. We already got the answer by looking at the dump: the inactive and erased word is `00000000`. Good (for us).
2. To discover which word gets priority, we can set a lock bit, issue a PROTECT command and tear-off before the erase. Then we test if the corresponding word is locked or not. The answer is that if both *Protection Words* are valid, word 15 is the active one. It is important to know it because if we start with an active word 15, no matter

what we manage to write into 14 and perform a tear-off, word 15 will always be the active one. So we must start our experiments from an active word 14 configuration.

3. A simple test shows that, even if unneeded, the full PROTECT cycle is performed and the *Protection Words* get swapped. It's interesting as we can swap the words at will: e.g. every time the configuration we are writing to word 15 becomes active but doesn't suit our goal, we can retry from an active word 14 by issuing such dummy PROTECT command. Good (for us).

For the last two questions, it's only by trying that we will get an answer!

Example of a configuration with two valid words, word 15 getting priority.

```
[=]   14 | 00008003 | ....  |   | Lock
[=]   15 | 00008003 | ....  |   | Lock       active
```

Strategy Development

All parameters are set, we can develop a strategy: issue a PROTECT command, perform a tear-off and hope that the `pr15` bit will be written before the `pr1` bit. (Remember: being written means being loaded above some threshold, e.g. loaded at 55%, while being not yet written means being loaded below the threshold, e.g. 45%)

1. Check which configuration is active, swap it if needed so that word 14 is the active one;
2. Issue a PROTECT(`00000000`), tear-off during writing;
3. Read both *Protection Words* to see where we stand;
    1. Did we tear too soon? (`pr15` bit of word 15 is not yet set, maybe you got a `00000000`, maybe `00000002`);
    2. Too late? (word 15 is fully written with `00008002` and active, word 14 might be intact, partially or fully erased);
    3. `00008000`? Sounds like a win :) To be confirmed;
4. If too soon, try again with longer delay;
5. If too late, swap words and try again with shorter delay. Beware that a weak `pr15` bit could tell you first that word 15 is active but when you want to swap, the chip sees it as inactive and you end up with an active word 15 after the swap. So words must be read again and a second swap must be issued if needed. It could also be that `pr1` bit is weak and even if at first sight word 15 seemed to be `00008002`, when swapping, it is seen as "0" and you end up with a word 14 containing `00008000`. But in such situation, word 15 is the active one so it doesn't help;
6. If we managed to write a word 15 with less bits than initially but with `pr15` bit set, so `00008000` if we work with a default chip, we have to commit that value because there could be weak bits with changing values. So we issue again a PROTECT, without tear-off, and we check the result. Now if word 14 contains `00008000`, we are certain we have successfully reset the `pr1` bit! Else, try again.

We have illustrated the strategy starting from a default configuration `00008002` but the same logic applies to configurations with more lock bits set.

Results

Let's put an end to the suspense: yes, we could clear the `pr1` bit to unlock the *UID Word* and... yes you can change the UID!

So, starting from a blank EM4305 tag, we have now an EM4305 fully unlocked, with writable UID, a kind of equivalent to the UID-writable Chinese clones of MIFARE tags.

```
[usb] pm3 --> lf em 4x05_write 1 deadbeef
[=] Writing address 1 data DEADBEEF
[usb] pm3 --> lf em 4x05_dump
[=] Found a EM4305 tag

[=] Addr | data     | ascii |lck| info
[=] -----+----------+-------+---+-----
[=]   00 | 0000E052 | ...R  |   | Info/User
[=]   01 | DEADBEEF | ....  |   | UID
[=]   02 |          |       |   | Password   write only
[=]   03 | 00009528 | ...(  |   | User
[=]   04 | 0001C258 | ...X  |   | Config
[=]   05 | 55564755 | UVGU  |   | User
[=]   06 | 95555556 | .UUV  |   | User
[=]   07 | 95A599A6 | ....  |   | User
[=]   08 | 00000000 | ....  |   | User
[=]   09 | 00000000 | ....  |   | User
[=]   10 | 00000000 | ....  |   | User
[=]   11 | 00000000 | ....  |   | User
[=]   12 | 00000000 | ....  |   | User
[=]   13 | 00000000 | ....  |   | User
[=]   14 | 00008000 | ....  |   | Lock       active
[=]   15 | 00000000 | ....  |   | Lock
```

What is the success rate?

By experiment, about 85% of the tags we tried, starting from a default configuration, could be unlocked. Sometimes in a dozen seconds, sometimes in a few minutes, trying different positions on the antenna.

One could expect the probability to be 50% as either `pr15` bit gets always written before `pr1` bit (protecting the *UID Word*), or always after `pr1` bit, for a given tag. The reality is that it's a process comprising stochastic fluctuations, so e.g. after a tear-off one bit will be filled at 45%+-6% and the other at 55%+-6%, which means that by repeating the experiment many times, at some point one can be filled at 51% and the other one at 49%. If the bias between our two bits is strong and unfavorable, we will be unable to unlock the tag.

We also noticed that there is more variability when the tag is put at a greater distance from the reader. The write process is slightly slower as the power budget is smaller and we had success on tags which we were unable to unlock when put directly on the reader.

If we start from a configuration with more words locked (e.g. `0000BFFF`), we can hope to clear a few bits but it will be hard to clear them all as some will probably always be written before the `pr15` bit.

The results are specific to each tag but they are quite reproducible.

New Command Ahead!

The strategy was designed firstly in a Lua script for fast prototyping, then implemented in a native Proxmark3 command automating all the process to quickly find the right timing: `lf em 4x05_unlock`, using internally the new generic tear-off functionality we mentioned at the start.

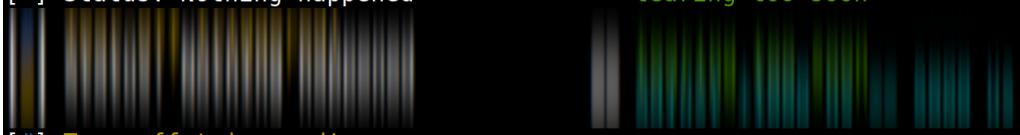
The output being rather long, here is the beginning and the end of one execution:

```
[usb] pm3 --> lf em 4x05_unlock
[=] -------------- EM4x05 tear-off : target PROTECT ----------------------

[=] initial prot 14&15 [ 00008002, 00000000 ]
[=]     automatic mode [ enabled ]
[=]    target stepping [ 2000 ]
[=] target delay range [ 2000 ... 6000 ]
[=]       search value [ 00008002 ]
[=]        write value [ 00000000 ]
[=] ------------------------------------------------------------------------


[=] press 'enter' to cancel the command

[=] -------------- start ----------------------

[#] Tear-off triggered!
[=] Status: Nothing happened            => tearing too soon

[#] Tear-off triggered!
[=] Status: 15 ok, 14 not yet erased    => tearing too late
[=] Tried 5 times, soon:1 late:4        => adjust -1 us >> 4052 us
[#] Tear-off triggered!
[=] Status: 15 bitflipped and active    => SUCCESS?:  14: 00008002  15: 00008000
[=] Committing results...
[=] Status: confirmed                   => SUCCESS:   14: 00008000  15: 00000000
[=] ---------------------------- exit ----------------------------------


[+] time in unlock 50 seconds

[=] Old protection word => 00008002
[=] Bitflips:  1 events => .......x
[=] New protection word => 00008000

[=] Try `lf em 4x05_dump`
```

Countermeasures

It's hard to tell if unlocking the EM4305 UID has really any security impact. Designing an access control system solely based on a fixed ID is known to be flawed anyway and could already be defeated by the various Proxmark3 simulation capabilities.

Maybe it could impact the pigeon racing mode, where part of the FDX-B ID can be locked (words 5 & 6) and part can be left open (word 7), as one could unlock and rewrite the ID. But in this scenario it would be easier just to replace the pigeon ring with another one anyway.

Locking the Protection Words?

What countermeasures are possible, besides changing the hardware implementation?

One could lock the *Protection Words* themselves, by setting the `pr14` bit. In this case, it's not possible to set new bits with the PROTECT command and for any attempt, the tag will return an error preamble to refuse the command.

Yes but...

The funny thing is that, if we read back words 14 & 15 after such denied PROTECT command, the words got nevertheless swapped!

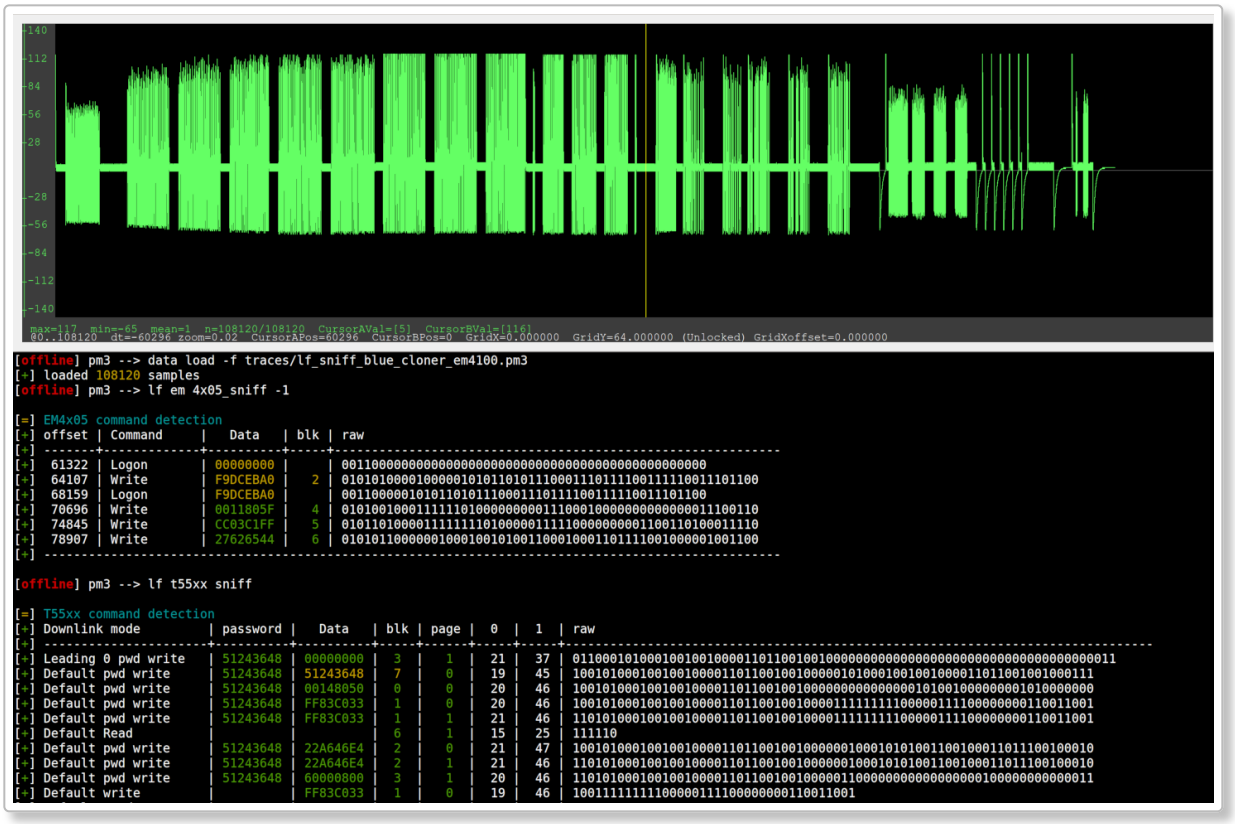
It means that the *Protection Words* update sequence took place regardless of the tag response. It's not possible to set new bits but it's still possible to run the tear-off strategy and to clear lock bits! Including possibly the `pr14` bit.

Adding a Password?

We believe the best way to avoid tear-off against *Protection Words* on EM4305 is to enable the password for WRITE/PROTECT commands. This can be done by setting the *Write Login* bit `co20` in the configuration word (word 4). Once the *Write Login* bit is enabled, a PROTECT command without a password is denied and the *Protection Words* are not swapped, so no doubtful EEPROM write, this time. Once the password is activated, a LOGIN command must be performed.

Obviously, the password must be changed. Keep in mind we added `lf em 4x05_chk` to perform a dictionary attack and `lf em 4x05_brute` to perform a bruteforce attack at a rate of about 50 passwords per second. Once the password is found, it can be provided to `lf em 4x05_unlock` to perform the required LOGIN command before PROTECT. E.g. we managed to unlock cloned HID cards.

Such passwords are typically collected by sniffing a reader while it's logging in onto the tag, as the password travels in clear. Recently, @mwalker added `lf em 4x05_sniff` and `lf t55xx sniff`, which greatly helps decoding such sniffed commands. Here is an example of the blue cloner mentioned in the previous blog post [1], while it's busy cloning an EM4100 ID into an ATA5577C (first half of the graph, on the left of the yellow cursor) then into an EM4305. Typically these cloners emit both write sequences blindly without trying to understand which type of tag is presented.



Conclusion

The shipment of a few EM4305 by mistake led to an unexpected but interesting adventure in the possibilities offered by the EEPROM tearing off method.

Being able to change the UID of an EM4305 is probably not very useful *per se*. But the whole exercise of understanding how a tear-off protection is designed and finding ways to defeat it is rewarding. It helps to better understand the subtle implications of tear-off physical principles and prepares the ground for more complex attacks to come.

It was also the occasion to design a more generic approach in the Proxmark3 code, to enable fast prototyping of tearing-off experiments on the whole range of LF and HF tags supported by the Proxmark3, followed by the implementation of a specific automated command targeting the EM4305 protection mechanism.

There is no more excuses not to try our tear-off tools against any type of tag supported by the Proxmark3! See also the *Tearing-off Cheatsheet* here below.

-- @doegox & @herrmann1001

PS: Many thanks for colleagues and friends who proofread this article.

Tearing-Off Cheatsheet

Tearing-Off Commands in Proxmark3 RRG (11/2020)	
Chip/Standard	Command
ATA5577C	`lf t55xx dangerraw`
MIK640M2D	`hf mfu otptear` (semi-automated)
EM4x05	`lf em 4x05_unlock` (automated)
EM4x05	`hw tearoff` combined with `lf em 4x05_write`
EM4x50	`hw tearoff` combined with `lf em 4x50_write` or `lf em 4x50_write_password`
ISO14443A	`hw tearoff` combined with `hf 14a raw`
ISO14443B	`hw tearoff` combined with `hf 14b raw`
ISO15693	`hw tearoff` combined with `hf 15 raw`
iClass	`hw tearoff` combined with `hf iclass wrbl`
...	Add easily your commands to the list with `tearoff_hook()` !

[1] *(1, 2, 3, 4)* https://blog.quarkslab.com/eeprom-when-tearing-off-becomes-a-security-issue.html
[2] At the time of writing, we used commit 29b3477b. Beware syntax might have changed in later commits.
[3] A copy is available at http://proxmark.org/files/Documents/13.56%20MHz%20-%20MIFARE%20Ultralight/PFI%20-%20Federico%20Gabriel%20Ukmar%20LU1052979%20-%20Nahuel%20Grisolia%20LU1038395%20-%20Ingenier%c3%ada%20Inform%c3%a1tica.pdf
[4] https://en.mikron.ru/products/rfid-chip-inlays-maps/rfid-chips/product/rfid-chip-mik1312ed/
[5] https://www.infineon.com/cms/en/product/security-smart-card-solutions/contactless-memories/my-d-move-and-my-d-move-nfc/
[6] http://www.nfcic.com/index.php?_m=mod_product&_a=view&p_id=102
[7] *(1, 2)* https://www.emmicroelectronic.com/product/lf-animal-access-ics/em42054305

Comments

Christian Henn • 5 months ago

Hi,

i am on Iceman 11.02.2021
Tried the lf em 4x05 unlock on my 4305 card but it seems to not work

[usb] pm3 --> lf em 4x05 unlock
[=] ---------- EM4x05 tear-off : target PROTECT ----------

[=] initial prot 14&15 [0001805F, 00000000]
[=] automatic mode [enabled]
[=] target stepping [2000]
[=] target delay range [2000 ... 6000]
[=] search value [0001805F]
[=] write value [00000000]
[=] ----------

@doegox ➚ Christian Henn • 3 months ago

there was indeed an error introduced in the repo when fixing unrelated compilation warnings