Legic Prime: Obscurity in Depth

Henryk Plötz, Karsten Nohl
ploetz@informatik.hu-berlin.de,
nohl@virginia.edu

December 28th 2009

Legic tokens are RFID access and payment cards

- Contactless smart cards at 13.56MHz
  - Legic Prime: Proprietary, marketed since 1992
  - Legic Advant: ISO compliant, marketed since 2004
- Predominantly used in access control, but payment applications exist (i.e., cafeteria)
- Can hold several applications, but this feature is rarely seen

Legic Prime

- Old card type, as old as Mifare Classic (and at least as insecure)
- Proprietary radio protocol (applied to become ISO 14443 Appendix F): „LEGIC RF“
- Proprietary ‚Legic Encryption‘
- Slow data rate ($\sim$ 10 kbit/s), comparatively high read range (supposedly up to 70 cm)
- Card types: MIM22 (outdated), MIM256 (234 bytes storage), MIM1024 (1002 bytes storage)

Legic Advant

- New card type, developed in the 2000's
- Based on ISO 14443A or ISO 15693
- 3DES or AES, also backward compatible to ‚Legic Encryption'
- Several *ATC* card types with varying sizes (15693: 128-944 bytes, 14443: 544-3680 bytes)
- Not yet analyzed by us, therefore not covered in this talk

Legic takes obscurity to the extreme

- Shrouded in a cloud of closed-ness and exclusivity
- Compared to Mifare: much harder to get cards and readers on the free market (this is on purpose)
- No documentation available beyond layer 1+2 (in rejected ISO 14443F)
- Most marketed feature and main difference to other systems: Master Token System Control

Master Token System Control

> *The powerful LEGIC Master-Token System Control (MTSC) [...] is unique in the security industry. With MTSC no sensitive passwords are needed. Instead, a special physical Master-Token [...] is used containing a unique genetic code which securely links cards and readers.* – Source: http://www.legic.com/unique_security.html

- Cards are segmented and access is regulated on a per-segment basis
- Segment access is bestowed not through the knowledge of keys or passwords but through a physical token
- The MSTC token itself is a Legic card (either Prime or Advant)

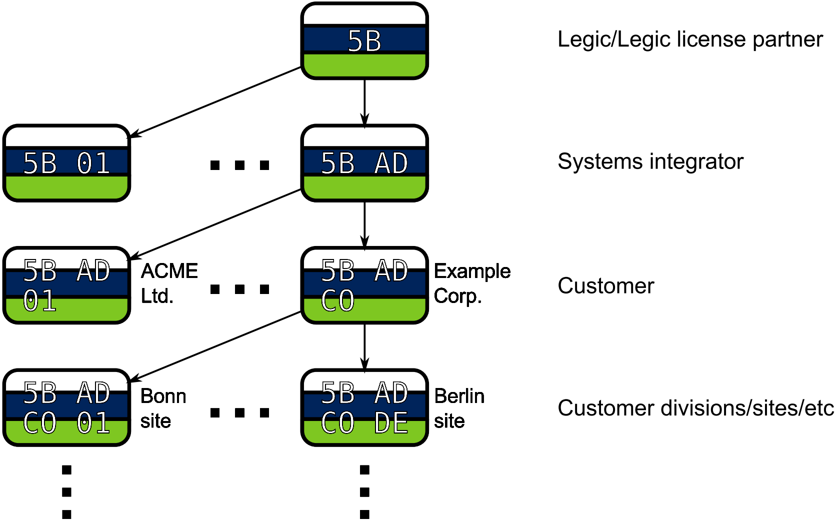
Henryk Plötz, Karsten Nohl

Segment protection

- Node identifier in the master token structure is called the stamp (or ‚genetic code')
- Segments on cards are imprinted with a stamp on creation
  - Stamp comes from the token that authorized the creation
  - Stamp can not be changed
- Optionally, segments can be „read protected"
- Readers are initialized with access rights for none/one/multiple stamps
- Card–Reader interaction:
  - Read read-protected segment and write: only if reader has access rights for that segment's stamp
  - Read non-read-protected segments: All readers can do this

MTSC Structure

- Token structure is hierarchical: a token can only create objects with higher nesting level than its own → longer stamp, but same prefix



Token Types

General Authorization Media (GAM)

Token-creating token that carries the temporary authorization to create sub-tokens

Identification Authorization Media (IAM)

Segment-creating token that carries the temporary authorization to create segments on cards

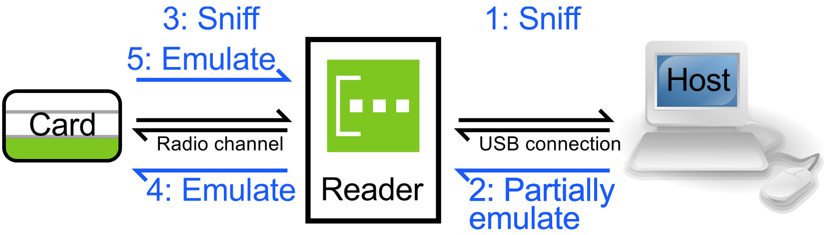
System Authorization Media (SAM)

‚Reader-creating‘ token that bestows the permanent authorization to write to existing segments on cards (and read read-protected segments)

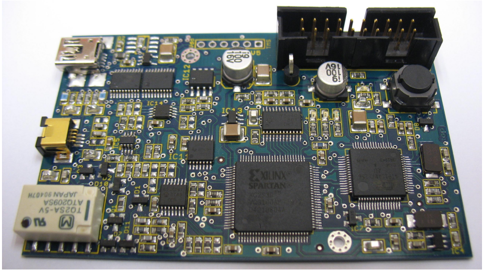
Token Sub-Types

- For the SAM (a.k.a. SAM63, a.k.a ‚Taufkarte'), which ‚launches' readers (‚*taufen*'), there is a counterpart: SAM64 (a.k.a ‚Enttaufkarte') to de-launch readers (‚*enttaufen*')
- Other types (possibly restricted to advant):
  - XAM: Permanent permission to create segments (e.g. a launching version of IAM)
  - IAM+: Restricted version of IAM, which only allows to create a given number of segments
- There are references to SAM4 ‚Parametrierkarte', which changes reader parameters. Also some systems may use other ‚SAM...' types for sneakernet purposes.

Roadmap and attack targets



- Attacks were implemented using the Proxmark3:

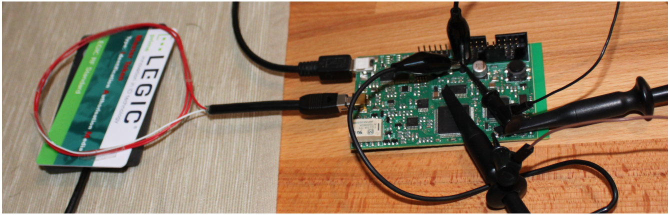


LEGIC RF

- ISO 14443 Annex F gives general parameters:
  - RWD to TAG: Pulse-pause modulation, 100% AM, off-duration: 20μs, ‚0'-bit: on-duration 40μs, ‚1'-bit: on-duration 80μs, data rate 10 kHz–16.6… kHz (data-dependent)
  - TAG to RWD: On-off-keying, load-modulation, subcarrier $f_c/64$ ($\sim$212kHz), bit-duration: 100 μs
  - Framing „defined by the synchronization of the communication"
  - No frame start/stop information for tag originated frames

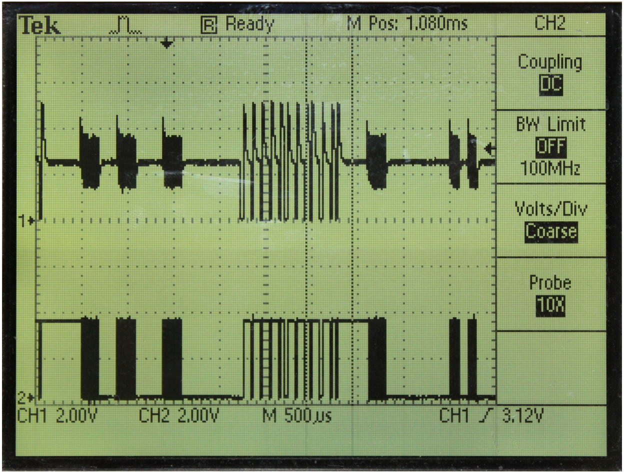
Sniffing LEGIC RF

- Sniffing with OpenPICC2 (fixed threshold, not so good) or Proxmark3 (hysteresis, much better) and oscilloscope or logic analyzer



Sniffing LEGIC RF

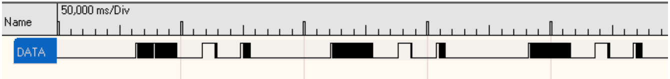
- Oscilloscope view:



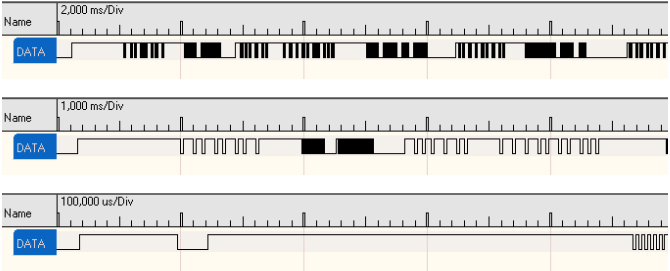
Sniffing LEGIC RF

Logic analyzer data:

- „Get UID“ type command, cycles through LEGIC RF, then ISO 14443-A, then ISO 15693:

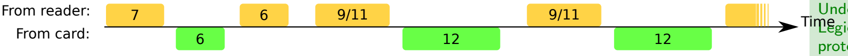


- „Get UID“ transaction consists of multiple exchanges by card and reader:



Decoding

- Custom decoder in C
- Delay between RWD command and TAG response seems to be constant, approx 330μs
- As expected: TAG-originated frames are not delimited, length unclear



- Comparing many traces yields the protocol structure:
  - Setup, once per session:
    - 7 bits from RWD
    - 6 bits from TAG
    - 6 bits from RWD
  - Repeat several times, once for each byte requested:
    - 9 bits from RWD (depending on card type: 11 bits for MIM1024)
    - 12 bits from TAG

Assumption of Encryption

- Let the 7–6–6 exchange be the ‚setup phase' and the remainder of the session be the ‚main phase'
- First 7-bit-command from RWD is more or less random, but always has first bit set, name it RAND. Assumption: IV of a stream cipher.
  - RNG is weak: a) Too small; b) 0x55 in ∼10% percent of cases
    (vs. expected 1.5%)
- For a given RAND the rest of the setup phase is identical over all cards of the same type (MIM256 and MIM1024 differ by one bit).
- Within a card type, for a fixed RAND, all reader command sequences are identical.

→ Looks like a stream cipher with weak IV from reader and no random from the card

Sniff: Get UID

UID 3e 17 44 3e			UID 3e 58 b8 79		
Src	Len	Bits	Src	Len	Bits
R	7	1010101	R	7	1010101
T	6	010001	T	6	010001
R	6	111000	R	6	111000
R	9	010010100	R	9	010010100
T	12	100010001101	T	12	100010001101
R	9	001011100	R	9	001011100
T	12	111111000110	T	12	000111101111
R	9	010100101	R	9	010100101
T	12	110011111011	T	12	111100001010
R	9	001011000	R	9	001011000
T	12	101100001000	T	12	010000101111
R	9	111101111	R	9	111101111
T	12	011001100001	T	12	001100101010

Note: These examples are synthetic and do not use the actual generator taps or CRC polynoms

Sniff: Get UID

UID `3e 17 44 3e`			UID `3e 58 b8 79`			$\bigoplus$: `00 4f fc 47`
Src	Len	Hex	Src	Len	Hex	
R	7	055	R	7	055	
T	6	022	T	6	022	
R	6	007	R	6	007	
R	9	052	R	9	052	
T	12	B11	T	12	B11	$B11 \oplus B11 = 000$
R	9	074	R	9	074	
T	12	63F	T	12	F78	$63F \oplus F78 = 947$
R	9	14A	R	9	14A	
T	12	DF3	T	12	50F	$DF3 \oplus 50F = 8FC$
R	9	034	R	9	034	
T	12	10D	T	12	F42	$10D \oplus F42 = E4F$
R	9	1EF	R	9	1EF	
T	12	866	T	12	54C	

Note: These examples are synthetic and do not use the actual generator taps or CRC polynoms

Sniff: Get UID

UID 3e 17 44 3e			UID 3e 58 b8 79			⊕: 00 4f fc 47
Src	Len	Hex	Src	Len	Hex	
R	7	055	R	7	055	
T	6	022	T	6	022	
R	6	007	R	6	007	
R	9	052	R	9	052	
T	12	B11	T	12	B11	$B11 \oplus B11 = 000$
R	9	074	R	9	074	
T	12	63F	T	12	F78	$63F \oplus F78 = 947$
R	9	14A	R	9	14A	
T	12	DF3	T	12	50F	$DF3 \oplus 50F = 8FC$
R	9	034	R	9	034	
T	12	10D	T	12	F42	$10D \oplus F42 = E4F$
R	9	1EF	R	9	1EF	
T	12	866	T	12	54C	

Note: These examples are synthetic and do not use the actual generator taps or CRC polynoms

Sniff: Get UID

UID 3e 17 44 3e			UID 3e 58 b8 79			$\oplus$: 00 4f fc 47
Src	Len	Hex	Src	Len	Hex	
R	7	055	R	7	055	
T	6	022	T	6	022	
R	6	007	R	6	007	
R	9	052	R	9	052	
T	12	B11	T	12	B11	$B11 \oplus B11 = 000$
R	9	074	R	9	074	
T	12	63F	T	12	F78	$63F \oplus F78 = 947$
R	9	14A	R	9	14A	
T	12	DF3	T	12	50F	$DF3 \oplus 50F = 8FC$
R	9	034	R	9	034	
T	12	10D	T	12	F42	$10D \oplus F42 = E4F$
R	9	1EF	R	9	1EF	
T	12	866	T	12	54C	

Note: These examples are synthetic and do not use the actual generator taps or CRC polynoms

Sniff: Get UID

UID 3e 17 44 3e			UID 3e 58 b8 79			⊕: 00 4f fc 47
Src	Len	Hex	Src	Len	Hex	
R	7	055	R	7	055	
T	6	022	T	6	022	
R	6	007	R	6	007	
R	9	052	R	9	052	
T	12	B11	T	12	B11	$B11 \oplus B11 = 000$
R	9	074	R	9	074	
T	12	63F	T	12	F78	$63F \oplus F78 = 947$
R	9	14A	R	9	14A	
T	12	DF3	T	12	50F	$DF3 \oplus 50F = 8FC$
R	9	034	R	9	034	
T	12	10D	T	12	F42	$10D \oplus F42 = E4F$
R	9	1EF	R	9	1EF	
T	12	866	T	12	54C	

Note: These examples are synthetic and do not use the actual generator taps or CRC polynoms

Sniff: Get UID

UID 3e 17 44 3e			UID 3e 58 b8 79			$\oplus$: 00 4f fc 47
Src	Len	Hex	Src	Len	Hex	
R	7	055	R	7	055	
T	6	022	T	6	022	
R	6	007	R	6	007	
R	9	052	R	9	052	
T	12	B11	T	12	B11	$B11 \oplus B11 = 000$
R	9	074	R	9	074	
T	12	63F	T	12	F78	$63F \oplus F78 = 947$
R	9	14A	R	9	14A	
T	12	DF3	T	12	50F	$DF3 \oplus 50F = 8FC$
R	9	034	R	9	034	
T	12	10D	T	12	F42	$10D \oplus F42 = E4F$
R	9	1EF	R	9	1EF	
T	12	866	T	12	54C	

Note: These examples are synthetic and do not use the actual generator taps or CRC polynoms

„Get UID“ command sequence

- Each reader-card request-response pair only transmits one byte of payload
- The UID is transmitted in order first byte, fourth/last byte, third byte, second byte (compared to the display in the GUI)
- Each response is protected by a 4 bit CRC
- A fifth byte is transmitted after the UID, this is an 8 bit CRC over the UID, stored on the card itself

Read commands

- Hypothesis: Command is 1 bit command code, 8 bit (or 10 bit) address, response is 8 bit data and 4 bit CRC
- „Get UID“ isn’t really requesting the UID, but simply reading the first 5 bytes of memory
- Hypothesis confirmed: Lowest bit (first bit transmitted) of command is command code, must not be changed; remaining 8 bits are address

Read commands (cont.)

- First command of „Get UID“ sequence is really „Read Byte 0“

Cmd	Arg								
C	x	x	x	x	x	x	x	x	Original („Read Byte 0“)
C	$\overline{x}$	x	x	x	x	x	x	x	New: „Read Byte 1“
C	x	$\overline{x}$	x	x	x	x	x	x	New: „Read Byte 2“
C	$\overline{x}$	$\overline{x}$	x	x	x	x	x	x	New: „Read Byte 3“

etc. pp.

- Timing is important. Also: Only one command per setup phase → **4 s** to read a full MIM256 card

Read commands (cont.)

- First command of „Get UID“ sequence is really „Read Byte 0“

Cmd	Arg								
C	x	x	x	x	x	x	x	x	Original („Read Byte 0“)
C	$\overline{x}$	x	x	x	x	x	x	x	New: „Read Byte 1“
C	x	$\overline{x}$	x	x	x	x	x	x	New: „Read Byte 2“
C	$\overline{x}$	$\overline{x}$	x	x	x	x	x	x	New: „Read Byte 3“

etc. pp.

- Timing is important. Also: Only one command per setup phase → **4 s** to read a full MIM256 card

ACHIEVEMENT UNLOCKED:

Access All Areas

You can now read all segments, even read protected ones

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\overline{x}$	x	x	x	x	x	x	x	x	x	x	x	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\bar{x}$	x	x	x	x	x	x	x	$\bar{x}$	x	x	x	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\bar{x}$	x	x	x	x	x	x	x	x	$\bar{x}$	x	x	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\overline{x}$	x	x	x	x	x	x	x	$\overline{x}$	$\overline{x}$	x	x	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\overline{x}$	x	x	x	x	x	x	x	x	x	$\overline{x}$	x	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\overline{x}$	x	x	x	x	x	x	x	$\overline{x}$	x	$\overline{x}$	x	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\overline{x}$	x	x	x	x	x	x	x	x	$\overline{x}$	$\overline{x}$	x	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\overline{x}$	x	x	x	x	x	x	x	$\overline{x}$	$\overline{x}$	$\overline{x}$	x	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\bar{x}$	x	x	x	x	x	x	x	x	x	x	$\bar{x}$	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\overline{x}$	x	x	x	x	x	x	x	$\overline{x}$	x	x	$\overline{x}$	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data	CRC	
x x x x x x x x	x x x x	Original (valid)
$\bar{x}$ x x x x x x x	x $\bar{x}$ x $\bar{x}$	Try (invalid)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC

- CRC in stream cipher is well known to be malleable (WEP, Mifare Classic, ...)
- With unknown CRC function, a simple approach is to brute-force the difference values for all 1-bit changes. The differences are fully additive

Data								CRC				
x	x	x	x	x	x	x	x	x	x	x	x	Original (valid)
$\overline{x}$	x	x	x	x	x	x	x	$\overline{x}$	$\overline{x}$	x	$\overline{x}$	1st Difference (valid)
x	$\overline{x}$	x	x	x	x	x	x	$\overline{x}$	x	$\overline{x}$	x	2nd Difference (valid)
x	x	$\overline{x}$	x	x	x	x	x	x	$\overline{x}$	x	$\overline{x}$	3rd Difference (valid)

⋮

Use as follows:

$\overline{x}$	x	$\overline{x}$	x	x	x	x	x	$\overline{x}$	x	x	x	Modified data (valid CRC)

Note: These examples are synthetic and do not use the actual CRC polynom

Attacking the CRC (cont.)

- After being able to freely anticipate the transport CRC, the UID-CRC can be attacked in a similar manner
- Yields two tables:
  - Transport CRC: 8 entries of 4 bits
  - UID CRC: 32 entries of 8 bits
- Gather known UID transactions for as many RANDs as possible (we managed 59 out of theoretically 64), modify responses for requested UID

Attacking the CRC (cont.)

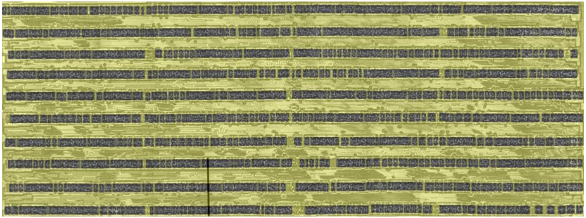
- After being able to freely anticipate the transport CRC, the UID-CRC can be attacked in a similar manner
- Yields two tables:
  - Transport CRC: 8 entries of 4 bits
  - UID CRC: 32 entries of 8 bits
- Gather known UID transactions for as many RANDs as possible (we managed 59 out of theoretically 64), modify responses for requested UID

ACHIEVEMENT UNLOCKED:

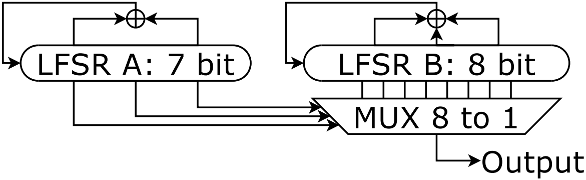
Pretender

You can now spoof arbitrary UIDs

Obfuscation function found through silicon reverse engineering

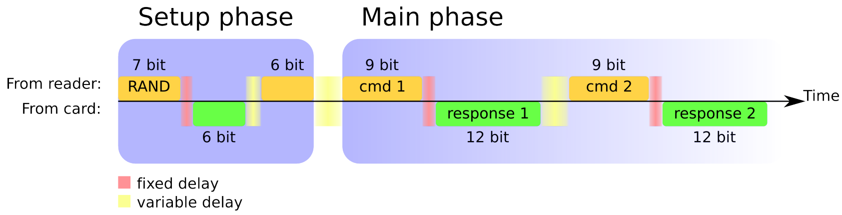


- The legic obfuscation function consists of two LFSRs
- Easily reversible, but not even needed for a state this small



A look at timing

- Until now: replay of recorded sessions with exact timing



Experiment Vary the timing before the first command

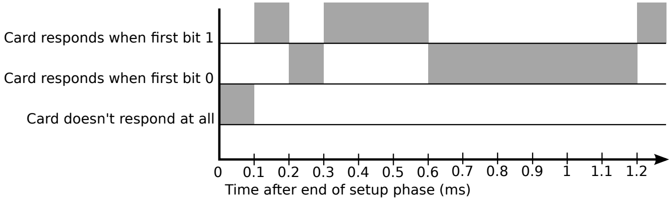
Result Card response for some delays, no card response for others

Interpretation Result of the de-obfuscation changed, which changes the command bit

Conclusion The obfuscation stream generator is continously running (at period time $\sim 100\mu s$)

Experimentally determine obfuscation stream

- Vary first bit of command at each time offset → gives first bit of obfuscation stream at that time offset



Note: These examples are synthetic and do not use the actual obfuscation stream

- Interpretation: The obfuscation stream generator generates a new bit approx. every 100 μs (more like 99.1 μs, might be reader-specific)

Complete break, even without a microscope: Generate arbitrary amounts of obfuscation stream by leveraging a few bits of known plaintext (optimized attack: 14 hours preparation, 4 kilobytes storage; naive attack: 4 days preparation, 80 kilobytes storage)

Combine knowledge

- Knowledge of the experimentally determined obfuscation stream allows to find the initialization for the function (brute force)
- Initialization:
  - **1st step** Load $R_a = \text{RAND}$ and $R_b = (\text{RAND} \ll 1)|1$
  - **2nd step** That's it, there's no 2nd step
    - No key input → not technically an encryption
- Can now generate obfuscation stream at any point in time
- Can send as many read commands in one single session as necessary → 0.69 s for a full dump of a MIM256

Combine knowledge

- Knowledge of the experimentally determined obfuscation stream allows to find the initialization for the function (brute force)
- Initialization:

  1st step Load $R_a = \text{RAND}$ and $R_b = (\text{RAND} \ll 1)|1$

  2nd step That's it, there's no 2nd step

  - No key input → not technically an encryption
- Can now generate obfuscation stream at any point in time
- Can send as many read commands in one single session as necessary → 0.69 s for a full dump of a MIM256

ACHIEVEMENT UNLOCKED:

Meep, meep

You can now read cards much faster

Towards an emulator

- Both the slow and the fast reader ignore the transport CRC, but for a full card emulator we need to generate the CRC
- Look at the sniffed communication (de-obfuscated):

Src	Len	Binary	Hex	Interpretation
	(setup phase omitted)			
RWD	9	1 0000 0000	001	read byte 0
TAG	12	0111 1100 1111	F3E	answer: 3e, CRC f
RWD	9	1 1000 0000	003	read byte 1
TAG	12	0111 1100 0110	63E	answer: 3e, CRC 6
RWD	9	1 0100 0000	005	read byte 2
TAG	12	0010 0010 0000	044	answer: 44, CRC 0
RWD	9	1 1100 0000	007	read byte 3
TAG	12	1110 1000 0010	417	answer: 17, CRC 4
RWD	9	1 0010 0000	009	read byte 4
TAG	12	0001 0010 0111	E48	answer: 48, CRC e

Note: These examples are synthetic and do not use the actual CRC polynom

Towards an emulator

- Both the slow and the fast reader ignore the transport CRC, but for a full card emulator we need to generate the CRC
- Look at the sniffed communication (de-obfuscated):

Src	Len	Binary	Hex	Interpretation
	(setup phase omitted)			
RWD	9	1 0000 0000	001	read byte 0
TAG	12	0111 1100 1111	F3E	answer: 3e, CRC f
RWD	9	1 1000 0000	003	read byte 1
TAG	12	0111 1100 0110	63E	answer: 3e, CRC 6
RWD	9	1 0100 0000	005	read byte 2
TAG	12	0010 0010 0000	044	answer: 44, CRC 0
RWD	9	1 1100 0000	007	read byte 3
TAG	12	1110 1000 0010	417	answer: 17, CRC 4
RWD	9	1 0010 0000	009	read byte 4
TAG	12	0001 0010 0111	E48	answer: 48, CRC e

Note: These examples are synthetic and do not use the actual CRC polynom

Towards an emulator

- Both the slow and the fast reader ignore the transport CRC, but for a full card emulator we need to generate the CRC
- Look at the sniffed communication (de-obfuscated):

Src	Len	Binary	Hex	Interpretation
		(setup phase omitted)		
RWD	9	1 0000 0000	001	read byte 0
TAG	12	0111 1100 1111	F3E	answer: 3e, CRC f
RWD	9	1 1000 0000	003	read byte 1
TAG	12	0111 1100 0110	63E	answer: 3e, CRC 6
RWD	9	1 0100 0000	005	read byte 2
TAG	12	0010 0010 0000	044	answer: 44, CRC 0
RWD	9	1 1100 0000	007	read byte 3
TAG	12	1110 1000 0010	417	answer: 17, CRC 4
RWD	9	1 0010 0000	009	read byte 4
TAG	12	0001 0010 0111	E48	answer: 48, CRC e

Note: These examples are synthetic and do not use the actual CRC polynom

CRC revisited

- A CRC is determined by four parameters: register width, polynom, initial value, final XOR
- Storage CRC is 8 bits, transport CRC is 4 bits: Easy to brute-force over the full parameter space
- If all the known inputs are of the same length, initial value and final XOR are equivalent: Fixing one to an arbitrary value gives a solution for the other
- Better than brute force: Analysis of the 1-bit differences allows direct determination of the CRC parameters

Transport CRC

- Differently sized commands (9 bit for MIM256, 11 bit for MIM1024) allows to disambiguate initial value and final XOR
- Result: transport CRC is made over the full command and the full payload of the response:

From reader		From card	
1	Address	Data	CRC
	8/10 bits	8 bits	4 bits

Transport CRC

- Differently sized commands (9 bit for MIM256, 11 bit for MIM1024) allows to disambiguate initial value and final XOR
- Result: transport CRC is made over the full command and the full payload of the response:

From reader		From card	
1	Address	Data	CRC
	8/10 bits	8 bits	4 bits

ACHIEVEMENT UNLOCKED:

Chameleon card

You can now spoof arbitrary card contents

Write commands

- Write commands are 21 bit for MIM256 and 23 bit for MIM1024
- Contains command code („0“), 8/10 bit address, 8 bit data, 4 bit CRC
- Same CRC as for read commands, calculated over the full 17/19 bits

0	Address	Data	CRC
	8/10 bits	8 bits	4 bits

- Card acknowledges with a single „1“-bit, after 3.6 ms
- Obfuscation stream is unaffected by ACK

Write commands

- Write commands are 21 bit for MIM256 and 23 bit for MIM1024
- Contains command code („0“), 8/10 bit address, 8 bit data, 4 bit CRC
- Same CRC as for read commands, calculated over the full 17/19 bits

0	Address	Data	CRC
	8/10 bits	8 bits	4 bits

- Card acknowledges with a single „1“-bit, after 3.6 ms
- Obfuscation stream is unaffected by ACK

ACHIEVEMENT UNLOCKED:

Prolific Writer

You can now write to cards

Sniffing a master token load

- Analysis of a „load IAM“ or „launch reader“ process reveals:
  - UID is read, UID-CRC is read
  - Bytes 6 and 5 are read (in that order)
  - Byte 7...(7+stamp length) are read
  - Byte 21 is read
- Launch process takes a long time, ∼15 s, providing the illusion that something profound is happening (key-derivation? lengthy EEPROM reprogramming?)
  - On the radio channel, byte 4 (UID-CRC) is read every 1 s, to ping whether the card is still there

Reconstruct token contents from sniff

- Combining the address/data information from a sniff, the following structure of an IAM is revealed:

Address	Data							
0	3e	3e	44	17	48	2f	f8	04
8	5b	ad	c0	de				
16						0e		

Note: These examples are synthetic and do not use the actual CRC polynom. Also, the stamp is fake. Obviously.

Copy/emulate token

- Naive transfer to physical card not successful
- Bytes 5 and 6 behave strange when writing → can only be decremented
- Complete emulation is successful
- Playing with the emulated card reveals: Byte 21 is a CRC, secures UID and stamp
- Exhaustive search over the CRC byte enables emulation of an IAM for different stamps

Henryk Plötz, Karsten Nohl

CRC, again

- Analysis of CRC bit differences reveals: same CRC polynom as for the UID
- Further analyses find a common set of parameters for the UID CRC and the master token CRC
  - Disambiguates initial value/final XOR
  - Master token CRC is calculated over:
    1. UID, bytes 0 thru 3
    2. Bytes 6 and 5
    3. Byte 7
    4. Stamp, bytes 8 thru (8+(stamp length)-1)
- Can now emulate IAM and SAM for arbitrary stamps of length 4

S(tamp s)ize matters

Now that we can generate the master token CRC, let's play with the different bytes:

- Byte 5 seems to control the token type
- Byte 6 seems to control the stamp length, in coordination with byte 7
- Byte 7 is 0x04 for the IAM and 0x44 for the SAM (both of stamp length 4)

S(tamp s)ize matters (cont.)

- Wrong values for byte 6 tend to freak out the software: differing error messages, exceptions, crashes or the mute pretense that the card is empty
- Lucky accident: Set byte 7 to 0x00, byte 6 to 0xfc and we got ourselves an IAM of stamp length 0

S(tamp s)ize matters (cont.)

- Wrong values for byte 6 tend to freak out the software: differing error messages, exceptions, crashes or the mute pretense that the card is empty
- Lucky accident: Set byte 7 to 0x00, byte 6 to 0xfc and we got ourselves an IAM of stamp length 0

ACHIEVEMENT UNLOCKED:

Uber-IAM

You can now create and read arbitrary segments

Master Token contents

- Byte 7 is RD/WRP/WRC
  - Low nibble controls the stamp size
  - High nibble controls the stamp size for the launch process
- Byte 5 is token type: MSBit controls whether the token can create sub-tokens (OLE), remaining 7 bits are:

  0x00–0x2f IAM
  0x30–0x6f SAM
  0x70–0x7f GAM
- Byte 6 is the organisational level? Must be 0xfc - (stamp length)

Master Token contents

- Byte 7 is RD/WRP/WRC
  - Low nibble controls the stamp size
  - High nibble controls the stamp size for the launch process
- Byte 5 is token type: MSBit controls whether the token can create sub-tokens (OLE), remaining 7 bits are:

  0x00–0x2f IAM
  0x30–0x6f SAM
  0x70–0x7f GAM
- Byte 6 is the organisational level? Must be 0xfc - (stamp length)

Achievement Unlocked:

???

Gratuitous GAM

You can now create GAMs with stamps of 2 bytes or longer

Extent of pwnage

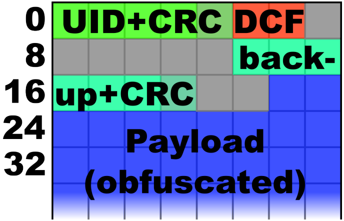
- Can create IAMs and SAMs for arbitrary stamps of arbitrary lengths (including 0!)
  - If the SAM should launch readers, its stamp length must be at least 1
  - Uber-IAM allows full read and creation access to arbitrary stamps
- Can create GAMs for arbitrary stamps of length 2 or higher
  - The software seems to specifically lock out shorter GAMs, pretends the card is empty

Comprehending card contents

- Reverse engineering card contents not necessary for the standardized types (e.g. cash, access, biometric): Simply use the regular software together with the Uber-IAM
- Otherwise, if available, use csg files (legic segment definition) to aid in interpretation
- Data on the card is further obfuscated: All payload bytes are XORed with some value.
  That value is the CRC of the UID (which is also stored on the card)
  → Obscurity In Depth

Card format

- 4 bytes UID + 1 byte CRC
- 2 bytes decremental field (DCF), is 0x60 0xea for all cards that aren't master token
- 6 bytes unknown/unused/fixed, might be a version identification, possibly related to old unsegmented cards
- 6 bytes segment header backup area + 1 byte CRC
- 2 bytes unknown/unused
- remainder: obfuscated payload



Segment format

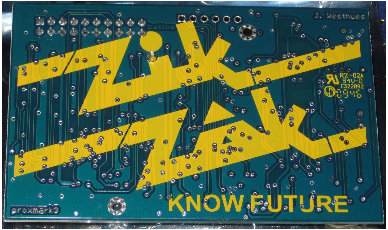
- Segment header is 4 bytes + 1 byte CRC
  - 1st byte: lower byte of segment length (including header)
  - 2nd byte, lower nibble: high nibble of segment length
  - 2nd byte, high nibble: flags: 0x8 == last segment flag, 0x4 == segment valid flag (if flag is not set, the segment is deleted)
  - 3rd byte: WRP, length of write protected area of the segment. Always includes the stamp length
  - 4th byte, bits 4 thru 6: WRC
  - 4th byte, MSBit: RD, read protection
- Segment header write procedure:
  - Save old segment header to backup area
  - First byte of backup area := 0x80 (‚dirty') | segment number
  - Write new segment header
  - Clear dirty flag in backup area

Root Security Issues

- No keys, (no key management, no card authentication, no reader authentication)
  - → Spoofing, skimming
  - → Segments can be created out of thin air
  - → Master token can be created out of thin air
- No authorisation necessary for master token use, master token not inherently necessary for segment creation
  - → Master token clonable

Proxmark3 allows pen-testing Legic systems

- We release today: Legic Prime reader emulation
  - Test whether an access cards is Legic Prime (or HID, Mifare Classic) and hence vulnerable
  - Test whether private data is stored on the card (including in read-protected segments)
- **Proxmarks are available at 26C3**; look for the green laser in the basement



- We do not release: Card emulation, full protocol
  - Reverse-engineering these components is not hard
  - Therefore: Upgrade ASAP.

Please upgrade, just not to HID!

- Several RFID cards have been publicly broken over the past years: Mifare Classic, NXP Hitag2, Legic Prime
- Meanwhile, **HID Prox** – the card with the least security – still has a reputation of being secure
- Let us recap:
  - HID Prox cards can be read and emulated with a $20 device (c.f. proxpick.com)
  - Reading distance is at least 20cm
  - No crypto, no obfuscation, no protection; but: good lawyers

Conclusions and Outlook

- Even multi-level obfuscation does not prevent reverse-engineering
- Access cards at the very least need inherent protection in form of good crypto and secret keys
- Legic Prime analyzed head to toe
  - No actual, inherent security found
  - Advertised range ~70 cm & card completely unprotected against skimming → more significant break than with Mifare Classic
- Once again: Security by obscurity does not work

The End

?|!

Henryk Plötz–ploetz@informatik.hu-berlin.de
Karsten Nohl–nohl@virginia.edu