Peeling Away Layers of an RFID Security System

Henryk Plötz[1] and Karsten Nohl[2]

[1] Humboldt-Universität zu Berlin
[2] Security Research Labs, Berlin

Abstract. The Legic Prime system uses proprietary RFIDs to secure building access and micropayment applications. The employed algorithms rely on obscurity and consequently did not withstand scrutiny.
This paper details how the algorithms were found from opening silicon chips as well as interacting with tags and readers. The security of the tags is based on several secret check-sums but no secret keys are employed that could lead to inherent security on the cards. Cards can be read, written to and spoofed using an emulator. Beyond these card weaknesses, we find that Legic's trust delegation model can be abused to create master tokens for all Legic installations.

1 Introduction

The "Legic Prime" RFID card is used for access control to buildings throughout Europe including critical infrastructure such as military installations, governmental departments, power plants, hospitals and airports. Despite its use in high security installations, access cards can be cloned from a distance or newly created using a spoofed master token.
The Legic Prime cards use proprietary protocols and employ simple check-sums, which have not previously been revealed. This paper discusses how the proprietary protocol and crypto functions were found using a combination of silicon reverse engineering and black box analysis. Since the cards do not employ cryptographic encryption or authentication, knowledge of the proprietary protocol alone allows for cards to be read, written to, and spoofed.
Legic's Prime technology is unique among RFID access technologies in several respects: The Prime chip is the oldest RFID card to use the 13.56 MHz band, it is the most mysterious for none of its protocol is documented, access to Legic hardware is closely guarded, and Legic cards have long been the only ones supporting trust delegation to model organizational hierarchies. We find that due to its lack of public documentation, weaknesses in the cards have gone unnoticed for two decades. The weaknesses allow for Prime cards to be fully cloned with simple equipment and for the trust delegation to be circumvented, which enables an attacker to create Legic tokens for all installations where the technology is used.
Besides for access control, Prime cards are also used as micropayment tokens in cafeterias, resorts and public transport. Money stored on the

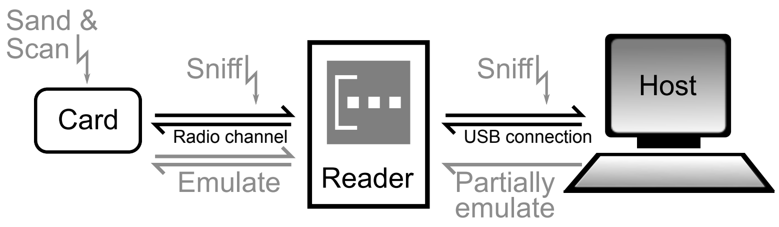

Fig. 1. Legic Prime system overview. Our analysis techniques are marked in light grey.

cards can be stolen from a distance. Legic's Prime technology must be considered insecure for its intended applications and should be replaced with cards employing peer-reviewed cryptography in open protocols.
The paper makes three main contributions by showing that:

1. Reverse engineering is possible with simple tools even for undocumented systems with multiple layers of obfuscation.
2. Legic Prime is insecure since attacks exists to clone tags and to spoof chains of trust.
3. While the trust delegation model in Legic is insecure, the same concept could be implemented securely using hash trees.

A general overview of the system that we worked with and our analysis techniques are given in Fig. 1. We partly analyzed the USB protocol and used that knowledge as a stepping stone to experimenting with the radio protocol. Concurrently silicon analysis took place on the chip embedded in the card.
The following section illustrates our two complementary approaches to reverse engineering RFID systems. Section 3 documents the Legic Prime card layout and protocol and points out several weaknesses. The concept of a trust hierarchy is introduced in Section 4 along with a discussion of why Legic's implementation is insecure while secure implementations are not hard to built.

2 RFID Reverse Engineering

Embedded computing systems such as RFID tokens often use proprietary protocols and cryptographic functions. The details of the algorithms are typically kept secret, partly out of fear that a system compromise will be easier once its operation principles are known. We found that the Legic Prime security system does not provide any inherent security beyond this secrecy.
A necessary first step in the security assessment of a proprietary system is reverse engineering of its functionality, which is achieved using one of two methods: a) Reverse engineering circuits from their silicon implementation as the more cumbersome method that is almost guaranteed to disclose the secret functions; b) Black-box analysis that can often be executed faster but requires prior information about the analysed system or lucky guesses. We employed both approaches in analyzing Legic Prime.

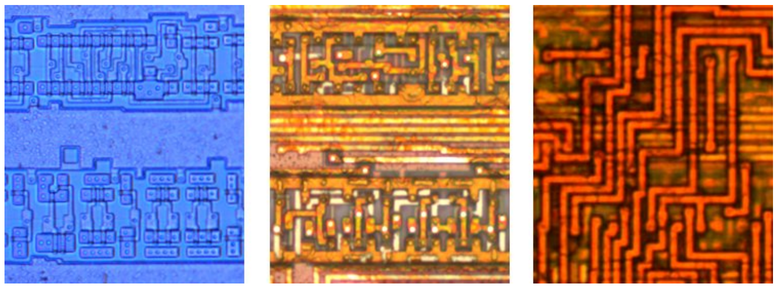

Fig. 2. Layers of a silicon chip: transistor layer, logic layer and one of the interconnect metal layers

2.1 Silicon Reverse Engineering

Disclosing secret algorithms from an RFID chip through reconstructing circuits has been demonstrated before when the Crypto-1 cipher was extracted from a Mifare Classic chip [5]. This project produced a suite of image analysis tools that work on images of the silicon chips. These images of the different chip layers are produced by polishing down the chip – a micrometer at a time – and photographing with an optical microscope. The tools then automatically detect recurring patterns in images such as those shown left in Figure 2. These patterns represent logic functions that are used as building blocks for algorithms similar to instructions in an assembly language.
The tools further support semi-automatically tracing of wires between the logic gates that disclose the chip circuit. In the case of Mifare Classic, around five hundred gates (out of a larger chip) and the connections among them were documented to disclose the encryption function [5]. In case of Legic Prime, the entire tag only consists of roughly five hundred gates, less than one hundred of which form a key stream generator. This key stream generator and part of the protocol initialization (described later in the paper) are detailed in Fig. 3. Reverse engineering the circuit of the entire digital part of the Legic Prime chip took two weeks[3].
The same reverse engineering techniques extend to any silicon chip including smart cards and trusted platform modules. The effort scales with the size of a chip; fully reverse engineering a microprocessor with millions of gates, for example, seems impractical with the current tools. However, security functions such as the on-chip encryption of smart cards are often small separated entities that can be imaged and disclosed independent from the rest of the chip. Currently the best protections against silicon reversing are randomly routed chips that mix security and other functionality, and chips of small feature sizes that require equipment more expensive than optical microscopes for imaging.

[3] The tools for silicon analysis (`Degate`) and the whole Legic Prime circuit can be downloaded at `http://degate.org/`.

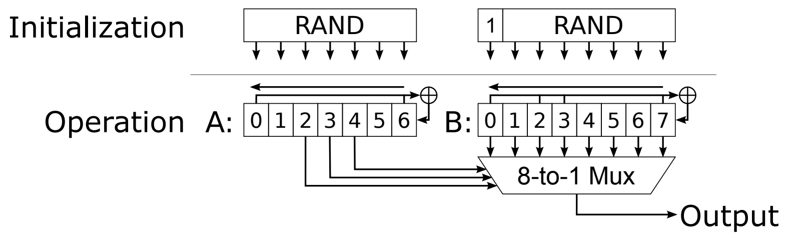

Fig. 3. The Legic Prime key stream generator and initialization function.

2.2 Black Box Analysis

Complementary to silicon reverse engineering, tests are conducted on the running system to add missing information. These two steps are roughly equivalent to disassembling and debugging in the software domain.
Given a functioning system – consisting of host software, a reader and cards – the goal of black box analysis is to learn system details by a) observing the behavior of the system on its communication channels in a passive manner, then b) trying to emulate parts of the system based on the observations from the first step, and finally c) varying the emulation by deviating from already observed behavior in order to observe new behavior. This approach was previously used for reversing the Texas Instruments DST40 cipher and protocol [4].
Our experimental setup consisted of a host PC running original Legic provided software and custom scripts. We wrote the scripts based on observations on the USB interface to replace functionality of the original software without necessarily abiding to its constraints.
As hardware we used a genuine Legic reader, a Proxmark3 RFID tool [3], and a logic analyzer. The Proxmark device was fulfilling multiple purposes: In the simplest case it is a short range RFID sniffer that outputs a demodulated carrier envelope signal on a debug test point to which we connected the logic analyzer. This setup allows for the traces of the communication between the original reader and original cards to be recorded with the stock Proxmark firmware. We could then look at the recorded waveforms on the PC for a rough visual, qualitative inspection and use custom scripts for further analysis of the data with the goal of decoding frames.
An initial survey showed that Legic had already published parts of its lower level protocols when applying for standardization as ISO 14443 Annex F [1]. While the application was rejected the material is still available online [2]. Using this information we were able to separate our trace into frames from reader and card and decode each frame to a bitstream. Later we were also able to write Proxmark3 firmware with the ability to receive and send frames, both in reader and card emulation mode. Without the ISO application document many of the modulation and encoding parameters would have had to be guessed. However, the parameter choices for the Legic Prime protocols are straight-forward (see Section 3) and could have been guessed with at most a few tries (long vs. short modulation from the reader, modulation vs. no modulation from the card).

Replay. Legic Prime allows for transactions to be replayed without necessarily understanding the protocol details. This is made possible by the lack of a random number from the card side, which happily accepts any past transaction. More importantly, from an attack perspective, transactions can also be replayed to a reader since the random numbers involved are weak. Replaying a transaction has an average success probability of up to 10% and the replay can usually be tried several times (e.g. at a door reader) until accepted (door opens).

USB protocol. To generate lots of similar RF traces for comparison we first partly reverse engineered the USB protocol used between the Legic host software and the Legic reader. The reader is connected to the host using an FTDI USB-to-serial converter (with a custom vendor and product id). We used usbsnoop for Windows to record all exchanges between host and reader. We then used custom scripts to extract the serial communications stream. The general frame format is similar to other Legic readers for which we found documentation on the internet[4]: each frame has a length byte, a body and a Longitudinal Redundancy Check (LRC, simple XOR over all the bytes including the length byte). The length byte does not count itself. When analyzing the USB communication we found one command that always preceeded all other commands and seemed to return the UID of the card in the field: `02 B0 B2`. We designated this command "GET UID" and wrote custom software to repeatedly (and rapidly) send this command to the reader on its USB channel.

RF protocol: UID command. A single GET UID command will try to enumerate cards of all supported protocols in the field: LEGIC RF, ISO 14443-A and ISO 15693. Using our sniffing setup we looked at the LEGIC RF portion of this sequence to understand the general layout of the protocol. We have not investigated the remaining two protocols, which are used by Legic's Advant tags.

Even for the same card, different GET UID sequences looked very different. Each sequence starts with a 7-bit frame from the reader, which seems to be mostly random, but always has the first bit set to `1`. We designated this frame "RAND" since it looked like the initialisation vector of a stream cipher: For all transactions with the same RAND, the sequence of reader commands is identical, while the sequences of card responses are identical only for identical UIDs.

By comparing multiple traces and looking for the first and last modulation observed from the card we found the following general structure in the protocol: 7 bits from reader, 6 bits from card, 6 bits from reader, then five repetitions of 9 bits from reader, 12 bits from card. This structure was observed for a 256byte card (MIM256); a MIM1024 card receives 11 instead of 9 bits from the reader. We named the 7-6-6 part the 'setup phase' and the remainder the 'main phase'. The contents of the setup phase only depend on the RAND frame. The remainder is always identical (within a card type; a MIM1024 card has one bit flipped in the 6-bit card response). So far the protocol looked like an authenticating stream cipher, with weak initialization vector from the reader. The ran-

[4] `http://www.rfid-webshop.com/shop/product_info.php/info/p318_LEGIC-Plug---Play-module.html/`

dom numbers from the readers are not only short but also statistically biased: `0x55` appears in roughly 1 out of 10 tries. Here, and in the following discussion, frames are represented as single integers transmitted in LSBit-first order.

Since the GET UID sequence must contain the UID of the card we XORed two traces from cards with different UID (but same RAND) to learn about the order of transmission. We found that the first card response in the main phase contained the first byte of the UID in the lower 8 bits, and something else (probably a checksum) in the higher 4 bits. The second card response contained the second byte of the UID, and so on[5]. Since there are only four bytes to the UID, but five data transmissions in the main phase for GET UID, we assumed that the fifth transmission would be some kind of checksum, most likely a CRC-8, which we called the storage CRC.

In order to further test our hypotheses and learn more about the system we implemented a card and a reader emulator which could replay previously recorded frames. In the first attempts we replayed the sniffed frames verbatim. This was made possible by the absence of any random number from the card –a reader emulator can completely play back a recorded transaction– and a weak random number from the reader –a card emulator can completely replay certain transactions with $\sim 10\,\%$ probability. Completely replaying frames worked flawlessly (as long as the timing of the original trace was followed precisely), so next we experimented with changing single bits in the main phase of the replayed traces, without touching the setup phase.

Flipping a single bit in the card responses in the main phase would make the reader abort the session, clearly indicating the presence of a checksum. Since we already knew that the data is transmitted in the lower 8 bits, the checksum must be in the high 4 bits, most likely some form of CRC-4, which we called the transport CRC. Since there are only 16 possibilities we opted for a quick brute force approach: Flip one bit in the data section, then try all 16 flip variants on the CRC section to find the variant where the reader would not abort the session. Using this approach we found a table with 8 entries of 4 bits each that would allow us to correctly fix the CRC for an arbitrary change of the data section, given a trace with correct transport CRCs. This approach works since CRCs are linear: $\mathrm{CRC}(a \oplus b) = \mathrm{CRC}(a) \oplus \mathrm{CRC}(b)$, under some preconditions (see [7]).

With this table we were able to send arbitrarily modified card responses (based on our initial guess to what the responses would mean) that were accepted by the reader. The reader would still not accept the complete UID, because of the checksum in the fifth byte. We attacked this byte in a similar manner, yielding a table of 32 entries with 8 bits each, allowing to spoof an arbitrary UID in the GET UID sequence. This validated our

[5] Different versions of the official host software display the UID in different formats. Older versions display the UID in order first byte, fourth byte, third byte, second byte. This seems to relate to a structure in the UID: The first byte is a manufacturer code, and the remaining three bytes are treated as a LSByte-first integer. We use the transmission and storage order in this paper.

assumptions on the meaning of all the parts of the card responses and further provides known keystream for all the UID data bytes.

From the CRC tables we could derive the used CRC polynomial: Since the CRC is over known data with only a single bit set, the differences between the different entries in the table differ only by the amount of shifts in the CRC calculation. Whenever a 1-bit is shifted out, the CRC polynom is XORed onto the state. By looking for these two properties in our tables we found the transport CRC polynomial to be `0xc` and the storage CRC polynomial to be `0x63` (but with a reversed shift direction).

RF protocol: reading memory. Based on the simplicity of the protocol so far observed we formed the following hypothesis about the reader commands: The GET UID sequence is not really requesting the UID (e.g. such as anticollision ISO 14443) but simply reading the first 5 bytes of memory. Each reader command in the GET UID sequence is a "read byte x" command. Since the command is 9 bits, and 8 bits are necessary to address all 256 bytes on a MIM256 card, that leaves 1 bit for the command code, namely "read" in this case.

We verified this assumption by replaying modified frames with correct timing: changing the first bit of the reader command would make the card never respond, while changing any other bit or bit combination would always lead to a response. This confirmed three things:

- The first bit is the command code and only a correct command code will lead to a response. The other command, presumably "write" must have another frame format.
- The remaining bits (8 for MIM256, 10 for MIM1024) are the address.
- The entire memory space can be read, with no restrictions (there is always a response, no matter the address).

Using the recorded first command of the GET UID sequence (which we now know is "read byte 0"), known keystream from its response and by changing the address in that command, we were able to completely dump the contents of any card; including unused and deleted segments.

RF protocol: key stream obfuscation. Next we sought to understand a phenomenon that we encountered early on in the implementation phase of the reader emulator: Not exactly following the timing of the recorded traces would sometimes make the card not respond. We concluded that the cipherstream generator must be continuously running, after the end of the setup phase, and replaying a recorded frame with some offset against the recorded timing would lead to a completely different frame on the card side, after decoding. In the instances where the card would not respond this changed frame was invalid. Since we know that the card will always respond to a command with a 'read' command code we could assume that the non-responding cases were those where the command code bit was received different from the 'read' command code.

Following up on the assumption of the continuously running cipherstream generator we leveraged existing known keystream to find new known keystream. First we determined the clock of the generator to be around 99.1 μs, which approximately matches the bit duration in card originated frames. We did this by a simple sweep over the possible delays before the first command in the main phase, and then sending a fixed command (all `1`). Since the card responds if, and only if, the first bit of the received

and decoded command is a correct read command code we could determine the current output of the keystream generator at the start of the command frame: Using our known keystream we found that the 'read' command code is `1`, so when we got a card response for some delay value we knew that the cipherstream at that point started with `0`.
By repeating this experiment for many different delay values (while always powering the card down between two trials) we got a time series that clearly showed the cipherstream output (with transitions only every $99.1\,\mu$s). Using this method it is possible to use the card as an oracle to generate cipherstream for any setup phase. From reverse engineering the silicon chip we knew that the stream generator has only 15 bits of state, so the stream repeats after $2^{15} - 1 = 32767$ bits. Reconstructing the complete stream from the card responses would take approximately 14 hours (due to the wait times incurred by powering down the card). This key stream can then be used to fully emulate a card or a reader without knowledge of the key stream generator, which we gained from the silicon chip.
By this point it is also clear that there is no key input to the stream generator since the recorded stream is portable between any card and any reader. We will no longer refer to it as an encryption (which it is not due to a lacking key) but only obfuscation function. Other radio protocols such as Bluetooth have similar mechanisms to enhance physical radio properties, called whitening.

3 Legic Prime Protocol

The rejected ISO 14443 annex F describes the lower layer radio protocol of Legic RF: Reader to card is 100% ASK with pulse-pause-modulation (1-bit is $100\,\mu$s, 0-bit is $60\,\mu$s, pause is $20\,\mu$s), card to reader is load modulation on a $f_c/64$ ($\approx 212\,$kHz) subcarrier with bit duration $t_{\mathrm{bit}} = 100\,\mu$s (subcarrier active means 1-bit). The annex does not specify the framing of card originated frames, merely stating that it is "defined by the synchronization of the communication". From our observations we found this to mean that the card frame starts at a fixed time after the reader frame (this time was measured to be $\sim 330\,\mu$s, which approximately equals 3 t_{bit}) and that the reader must know in advance how many bits the card will send, since there is no explicit frame end indication. Most notably this also means that not sending a frame (e.g. due to no card present, or card removed) is indistinguishable from sending a frame of only 0-bits.
The protocol consists of two phases: setup phase and main phase. The setup phase starts with the reader sending an *initialization frame* RAND of 7 bits (in LSBit-first order) with the lowest bit set to 1. At this point the obfuscation stream generator is started by setting $\mathrm{LFSR}_{\mathrm{A}}$:=RAND and $\mathrm{LFSR}_{\mathrm{B}}$:= (RAND$\ll$ 1) | 1 and all further communications are XORed with the current generator output. When no frame is being transmitted the generator generates one new bit every t_{bit}. When a frame is transmitted the generator is clocked with the data bit clock (this especially applies to reader originated frames which can have bit durations

that are smaller than t_{bit}). We found the generator initialization by trying different obvious variants of assigning RAND to the generator registers and comparing the output to the known obfuscation stream output from the previous section. Knowledge of the generator and initialization made it possible to completely deobfuscate all recorded traces of communication between the official Legic software, reader and card and observe all the remaining protocol specifics.
The card responds to the RAND frame, after a wait time of 3 t_{bit}, with an obfuscated *type frame* of 6 bits. This frame is either `0xd` for MIM22, `0x1d` for MIM256 or `0x3d` for MIM1024. The reader must wait at least one t_{bit} before sending its obfuscated *acknowledgment frame* of 6 bits. This is `0x19` for MIM22 and `0x39` for MIM256 and MIM1024. After this frame is sent the setup phase is complete and the main phase starts.
In the main phase the reader can send commands at any point in time. Each command has an address field of either 5, 8 or 10 bits (for MIM22, MIM256, or MIM1024 respectively). The following discussion only covers the 8-bit-address case, which is the most common card variant.
There are two types of commands: Read and Write. A read command consists of one bit command code `1`, followed by the address. After a waiting time of 3 t_{bit} the card will respond with 12 bits: The first 8 bits are the data byte from the transponder memory at the given address (LSBit first), the next 4 bits are the transport CRC-4. The transport CRC-4 is calculated with polynomial `0xc`, initial value `0x5`, and is calculated over the command code, address and data byte.
A write command consists of the command code `0`, the address, 8 bit data and 4 bit transport CRC-4. The transport CRC is calculated as above (just that the command code is now 0). If the write is successful, the card will respond with an ACK: a single unobfuscated `1`-bit. The time until the ACK can vary, but will be approx. 3.5 ms.

3.1 Card Layout

The memory space of a Legic Prime transponder is separated into three distinct physical zones: the UID (with its CRC), which is read-only, the decremental field (DCF), which, taken as a little endian integer, can only be decremented, and the remainder of the card, which can be freely written to. The entire memory space can always be read from. There are two variants for the logical organization of the card's payload data: unsegmented media (with master token being a special case), which contain exactly one segment, and segmented media, which can contain multiple segments. The protection features (on a reader firmware level) for both kinds of segments are essentially identical, and the headers are very similar. For this reason the rest of the paper will only consider the segment headers on segmented media, which now make up most of the market, and will cover master token as a special case of unsegmented media.
The general layout of a segmented Legic Prime medium is shown in Fig. 4. The remainder of the card that is not shown in the figure, starting at byte 22, contains the payload segments.
The payload area is obfuscated with the CRC of the UID: All bytes, beginning with byte 22, are XORd with $\mathrm{CRC}_{\mathrm{UID}}$ (address 4). Within

UID_0	UID_1	UID_2	UID_3	CRC_{UID}	DCF_{lo}	DCF_{hi}	`9F`
`FF`	`00`	`00`	`00`	`11`	BCK_0	BCK_1	BCK_2
BCK_3	BCK_4	BCK_5	CRC_{BCK}	`00`	`00`		
payload							

⋮

Fig. 4. Legic Prime card layout of segmented medium, containing a unique identifier (UID), decremental field (DCF), and a segment header backup (BCK) with its own CRC

the payload area the different segments are stored consecutively and each segment starts with a five byte segment header. This header consists of

- byte 0: lower byte of segment length, segment length includes the 5 bytes for the segment header
- byte 1, bits 0-3: high nibble of segment length
- byte 1, bit 6: segment valid flag, if this flag is not set, the segment has been deleted
- byte 1, bit 7: last segment flag, if this bit is set, no more segments are following
- byte 2: WRP, “write protection”
- byte 3, bits 4 through 6: WRC, “write control”
- byte 3, bit 7: RD, “read disabled”
- byte 4: CRC over the segment header

The different protection features are implemented in the firmware of all official Legic readers. To this end the data portion of a segment usually starts with the stamp of that segment, with the length of the stamp contained in the WRC field. A reader will compare this stamp to an internal database of stamps that it is authorized to operate on and then behave accordingly: write access is only allowed if the reader is authorized for that stamp. If the RD flag is set and the reader is not authorized for the stamp, then it will not allow any read access to the segment (including to the stamp). A reader will never allow write access to the bytes protected by the WRP field. A reader emulator can ignore all of these rules.

When writing a segment header, the official readers follow a special backup procedure to ensure that the segment structure cannot be corrupted by prematurely removing the card. Before changing a segment header which is included in the existing segment chain (e.g. all segment headers up to and including the first header that has the ‘last’ flag set), the complete header, including the CRC, is copied to BCK_1 through BCK_5, BCK_{CRC} is set to the CRC over BCK_0 through BCK_5 (note that BCK_0 has not been written yet). Then BCK_0 is written: bit 0 through 6 contain the number of the segment header (with 1 being the first segment header, this limits a card to maximal 127 segments) and bit 7 is a ‘dirty’ flag, which is set. Only after the backup area has been written will the actual segment header be changed. As soon as the new header is completely written, the backup area is invalidated by clearing the ‘dirty’ flag. Should the write to the segment header be interrupted, a reader will notice the flag next time when the card is presented and restore the the original segment header and then clear the flag. This procedure guarantees that any single change to any header is always handled atomically.

3.2 Weaknesses

Most protection functions are implemented in the firmware of the official Legic readers. The only hard protections in the card are the read-only state of bytes 0 through 4 (UID) and the decrement-only logic of bytes 5 and 6. Everything else on the card is freely read- and writable with a custom reader that ignores the protection flags. For all applications that do not explicitly check the UID it is possible to directly copy data from one card to another, as long as one fixes the payload obfuscation and CRCs (which both depend on the UID). Moreover there is no keyed authentication involved anywhere, so a clean dump and full emulation of cards are possible to trick even application that do check for the UID.
Since no keys are involved it is then also possible to spoof new, non-existing cards, including master tokens. Spoofed master-tokens can simplify attacks since with them it is not necessary to reverse engineer the complete payload format of an application: one can simply use an existing, official reader for that application and make it believe that it is authorized to work on the cards to be attacked.
Legic Prime poses an unusually large skimming risk, since, unlike most other card types, there is no read protection. A reader can read any card that is in its range. Because Prime was developed with read range in mind and uses very low power due to its simplicity, skimming ranges above what is common with ISO 14443 should be possible. The manufacturer variously states up to 70 cm read range for their official readers.
We also observed that the reader would usually only do the absolutely minimum changes necessary to modify the linked segment structure. For example when deleting a segment it will only clear the ‘valid’ bit in the segment header and not clear the segment payload. Also the backup area is never cleared after use. This means that a custom reader can gather much more data from a card than an official Legic reader: Most ‘deleted’ segments will still be intact and through the backup area there is a trace that shows which segment header was changed last.

4 Legic Trust Delegation

4.1 Card Hierarchy Concept

Legic systems are designed as a replacement for mechanical locking systems and implement not only their functional properties but also the organizational concepts of locking systems. The cards are organized in a hierarchy that represents the distribution path from Legic to the customer as well as the permission hierarchies within the customer’s organization.

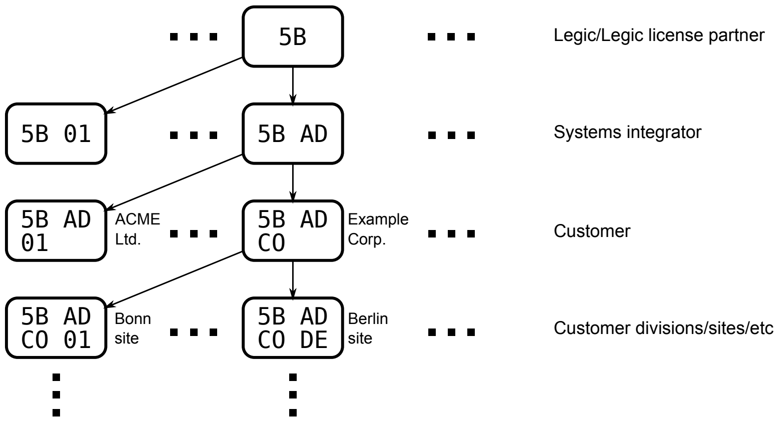

Fig. 5. Legic card trust delegation: The length of a card's stamp encodes its level in the hierarchy.

Cards that are higher in this virtual hierarchy can produce all cards below them. For example, a distributor can generate master tokens for all of its customers, even after the system was fully deployed; and Legic itself can generate tokens for all distributors. By delegation of trust, Legic can clone any card in existence or create new cards for any system since all cards are part of the same trust tree. While it is arguable whether a single company should have this level of control over its customers' systems, the trust tree implementation of Legic Prime allows for more concerning attacks: Anybody can move to the highest level of the tree thereby gaining control over all Legic Prime systems.

4.2 Legic Prime Implementation

Legic Prime distinguishes at least three types of master token:

- General Authorization Media (GAM), to create further master tokens
- Identification Authorization Media (IAM), to create segments on cards
- System Authorization Media (SAM), to transfer read/write authorizations to readers

Each node in the trust delegation hierachy has an identifier which is called the stamp (sometimes also "genetic code"). When creating a child node at least one byte is appended to the stamp, so nodes that are farther down in the hierarchy have longer stamps, see Fig. 5 for an example hierarchy[6]. Authorization decisions are made by a prefix match: A reader that is authorized to read segments with stamp `5B AD` is also authorized to read segments with stamp `5B AD C0`.

[6] The stamp `5B AD C0 DE` was chosen as a fictional example. Any resemblance to any real-world stamps is purely coincidental.

Creation of sub-tokens is controlled by another bit: Organization Level Enable (OLE). Only when this control bit is set is a master token authorized to create sub-tokens. A GAM (with OLE=1) can create any non-GAM master token with the same stamp as its own, or any master token with a stamp that is longer than its own and has the same prefix. IAM and SAM (both with OLE=1) can create IAMs and SAMs, respectively, with a stamp that is longer than their own and has the same prefix.

A master token is a special case of an unsegmented Legic medium which uses 22 bytes of data (exactly fits a MIM22 card, but can be written to larger media). The token type is encoded in the DCF, which has two main consequences: a master token cannot at the same time be a normal segmented medium and there are certain restrictions when writing master tokens to original Legic cards. For easier experimentation we performed the analysis of the master token structure with our card emulator which was implemented on the Proxmark and which allowed free change of the DCF. We first emulated the verbatim contents of a real master token and then performed incremental changes to the data, followed by a read using the official Legic software, in order to determine the meaning of the different fields.

For master tokens the header is interpreted as follows:

- $\mathrm{DCF}_{\mathrm{lo}}$ (byte 5), bit 7: OLE, organization level enable flag
- $\mathrm{DCF}_{\mathrm{lo}}$, bits 0 through 6: token type: `0x00-0x2f` IAM, `0x30-0x6f` IAM, `0x70-0x7f` GAM
- $\mathrm{DCF}_{\mathrm{hi}}$ (byte 6): must be `0xfc` - (stamp length), indicates level in the hierarchy
- byte 7, bits 0 through 3: WRP, contains the stamp size
- byte 7, bits 4 through 6: WRC, on SAM cards contains the number of stamp bytes that will be stored in the internal authorization database
- byte 7, bit 7: RD, not set for master token
- bytes 8. . .: stamp, variable length
- byte 21: CRC-8 over $\mathrm{UID}_{0\ldots3}$, $\mathrm{DCF}_{\mathrm{hi}}$, $\mathrm{DCF}_{\mathrm{lo}}$, byte 7, byte 8. . .

Note that this is the same general format as with unsegmented non-master token media (though we've only seen one such medium to date). For all non-master token the highest DCF value observed was `0xEA60` which is less than the minimal DCF value for a master token (`0xF000` for an IAM in level 12), so this field alone gives the distinction between master and non-master tokens. There might be other interpretations for the DCF: We didn't perform a comprehensive search over the DCF space, since the Legic software is rather picky and crashes when it encounters an unexpected DCF value.

The multiple possible values for the same master token type in the same hierarchical level can perform different functions. Setting $\mathrm{DCF}_{\mathrm{lo}}$, bits 0 through 6 to `0x31` gives a SAM63 in Legic lingo, while setting it to `0x30` gives a SAM64. SAM63 is also known as "Taufkarte", or launching card, while SAM64 is an "Enttaufkarte", or delaunching card. This refers to the processes of storing and deleting the authorization for a stamp in a reader which are known as "taufen"/launch and "enttaufen"/delaunch, respectively.

4.3 Weaknesses

Due to the way the DCF field is used, an existing master token cannot be freely modified and indeed can only be changed into a lower level master token. However, no such restriction applies when writing to a fresh card (or when using a card emulator). On a new card the DCF field is set to `0xFFFF` so it is possible to write any master token to such a card. Within the normal Legic system there are protections in the reader firmware that prevent creating a master token without the proper authorization, but no such mechanism applies when using a custom reader.

This means that master tokens can be freely copied to empty cards, and indeed freely generated for any desired stamp value. We also found that the prefix match for IAM and SAM is unrestricted: We have created an IAM of stamp length 0, which will prefix-match any stamp value. This Uber-IAM can authorize an official reader to read and write segments with arbitrary stamps. It is, however, not possible to launch a reader with a SAM with stamp length 0, since WRC must be ≥ 1 for a launch process to take place. Also GAMs with stamps shorter than 2 bytes seem to be specifically locked out in the host software: when trying to load such a GAM, the software will stall for a few seconds and then pretend that the card was empty.

The problem that master tokens can be cloned is inherent in the work flow of the Legic system (and therefore most likely also present in Legic Advant): In order to use a master token to create a new segment, first the master token (an IAM or GAM) is presented to the reader, which then stores the authorization information internally. Afterwards the reader will allow, until a configurable timeout occurs, to create segments on normal cards with this stamp or a longer stamp. The master token is not inherently necessary for the segment creation: by the time the segment is actually being created, the master token can long be back in a safe. This clearly shows that the complete 'essence of being' of that master token has been transferred into the reader, which means it's possible to read out and store all the data that is necessary to perform the functions that the master token allows. In the case of Legic Prime this only includes the stamp value and token type, but even if there was cryptographic key material in the master token, as might be the case with Legic Advant, it must be exportable. This export process cannot use any strong authentication, since the reader and the token share no previous association with each other, and the user is not prompted for a PIN or similar.

4.4 Improvement Potential

The concept of organizing access credentials in a trust hierarchy is not flawed in itself. In fact, the same idea is used very successfully –and securely– in virtual credential systems such as Microsoft's Active Directory service. For an implementation of trust delegation to be secure, the delegation process has to be one-way in the sense that only higher level entities are trusted. Legic's current implementation clearly is two-way as

stamps can be shortened and lengthened at will, thereby moving up and down the trust hierarchy.
A sensible system would use a one-way function such as a cryptographic hash function to assure that the access credentials of tokens cannot be elevated. The stamp of a lower level card would be created as

$$\text{stamp}_{\text{child}} = \text{Hash}\left(\text{stamp}_{\text{parent}}, \text{metadata}_{\text{child}}\right)$$

where the metadata is used to distinguish several child cards. Going one step beyond the functionality of the Legic Prime system, these stamps would not be exchanged in cleartext but rather used as secret keys in a strong encryption function such as 3DES or AES, which are available on several modern RFID tags. The idea of using hash trees for authentication hierarchies has already been discussed as early as 1988, in [6].

5 Conclusion

Systems must not rely on obscurity but should rather employ cryptography as a base for security functions. The Legic Prime security chain breaks in two places where cryptography is missing. First, cards and readers cannot authenticate each other, which allows an attacker to assume either role and read, write, or spoof cards and readers. Secret keys and a simple encryption or hash function would mitigate these problems. The second place where the lack of cryptography enables attacks against Legic systems is their unique trust delegation model. Since no secret information exists that could distinguish higher permission from lower levels, any of the levels can be spoofed. The idea of having secure trust delegation, however, models many organizations' needs very well, which may have contributed to the popularity of Legic cards. Creating a system with secure delegation features is left for further research and development.

References

1. Article "ISO14443" in the openpcd wiki, section "LEGIC RF", revision as of 00:32, 6 september 2010, `http://www.openpcd.org/index.php?title=ISO14443&oldid=193#LEGIC_RF`
2. ISO 14443 Part 2 Amendment 1, dRAFT 2nd P-DAM BALLOT TEXT
3. PROXMARK III community, `http://www.proxmark.org/`
4. Bono, S., Green, M., Stubblefield, A., Juels, A., Rubin, A., Szydlo, M.: Security analysis of a cryptographically-enabled RFID device. In: USENIX Security Symposium (2005)
5. Nohl, K., Evans, D., Starbug, Ploetz, H.: Reverse-engineering a cryptographic RFID tag. In: USENIX Security Symposium (2008)
6. Sandhu, R.S.: Cryptographic implementation of a tree hierarchy for access control. Information Processing Letters 27(2), 95–98 (February 1988), `http://dx.doi.org/10.1016/0020-0190(88)90099-3`
7. Stigge, M., Plötz, H., Müller, W., Redlich, J.P.: Reversing crc–theory and practice (2006), `http://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2006-05/SAR-PR-2006-05_.pdf`