Unwrapping the Chrysalis

M.Bond, D.Cvrcek, S.Murdoch

University of Cambridge

Abstract. We describe our experiences reverse engineering the Chrysalis-TS Luna CA3 – a PKCS#11 compliant cryptographic token. Emissions analysis and security API attacks are viewed by many to be simpler and more efficient than a direct attack on an HSM. But how difficult is it to actually "go in the front door"? We describe how we unpicked the CA3 internal architecture and abused its low-level API impersonate a CA3 token in its cloning protocol – and extract PKCS#11 private keys in the clear. We quantify the effort involved in developing and applying the skills necessary for such a reverse-engineering attack. In the process, we discover that the Luna CA3 has far more undocumented code and functionality than is revealed to the end-user, and discuss the impact of this on the security of the token.

1 Introduction

The Luna CA3 is a Hardware Security Module (HSM) manufactured by Chrysalis-ITS [2], widely used for secure storage of private keys in Certification Authorities and other cryptographic applications. The Luna CA3 (seen in figure 1) has a PCMCIA form-factor which allows it to be easily removed and locked in a safe, and has been validated to FIPS 140-1 Level 3. Its external Security API is PKCS#11 – the de facto standard for the majority of HSMs that are integrated with PKI software from vendors such as Entrust. In addition to the standard PKCS#11 API calls, the CA3 has some vendor specific functions that facilitate backup and redundancy, and secure key entry.

We were interested in examining the vendor specific functionality that Chrysalis had added to the PKCS#11 API, in particular its cloning protocol, used for back-up and availability of key material. When one of our Luna tokens malfunctioned and became non-responsive, it gave us the opportunity to consider a physical attack, with very little to lose. We thus set out to analyse the device with several goals in mind:

- To learn about reverse-engineering and to gauge the effort and resources required to attack an HSM.
- To analyse the cloning protocol and develop a method for extraction of all private keys from the device in the clear, with the co-operation of the Security Officer in charge of the device (migration of keys to another architecture or into the clear was not a *standard* feature provided by Chrysalis).
- To analyse the Luna API for security at the tamper-resistant boundary, and learn about internal HSM architecture.

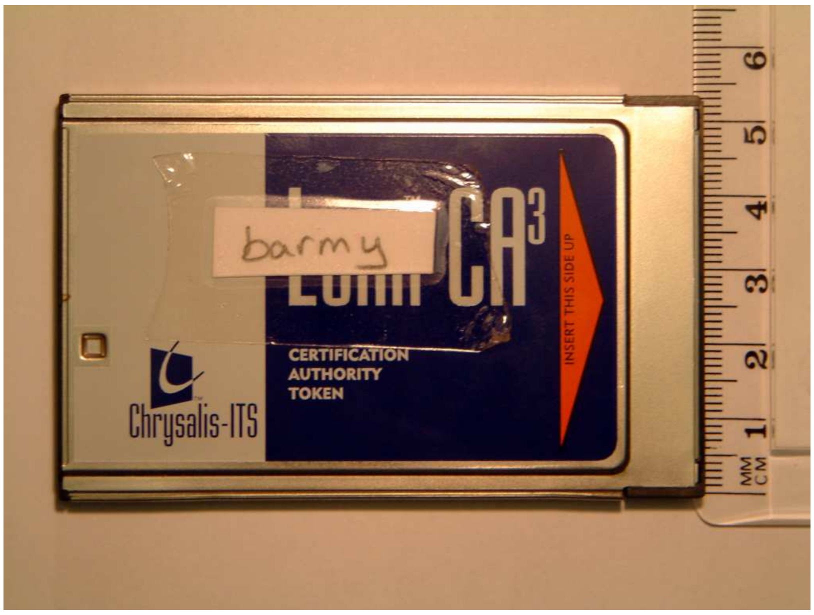

Fig. 1. The Chrysalis-ITS Luna CA3 Cryptographic Token

– As secondary consideration to assess the quality of code with respect to conventional vulnerabilities (for example buffer overflows).

This paper tells the story of how we achieved the above goals, presenting the work done chronologically to capture the flow and spirit of the challenges involved in reverse-engineering, and to work towards a quantitative picture of the effort involved. Section 2 introduces the Luna CA3 and explains the cloning protocol which was of particular interest to us. Sections 3, 4 and 5 describe how we located, understood, and eventually adapted the cloning protocol to suit our own ends. Finally, some conclusions are drawn in section 7.

2 The Luna CA3

To access key material on a CA3 token, a conventional user first needs to log on using a special Luna Pin Entry Device (PED), as shown in figure 2. The user inserts a *black datakey* containing some secret authorisation key material, and also types a numeric PIN on the keypad of the PED. If the wrong key or PIN is supplied three times consecutively, the user is locked out. A *blue datakey* is held by the Security Officer, who is in charge of initialisation and fine-grain policy of the token, and can regenerate the access credentials for the user. Of particular interest was the cloning facility mentioned in the manual, which was orthogonal to the import and export mechanisms provided by PKCS#11. A destination token had to be initialised into the same 'domain' as the source, by providing a common *red datakey* containing some domain-specific secret. The security officer could then log on to each token, and use a command line utility to copy all the keys from source to target token.

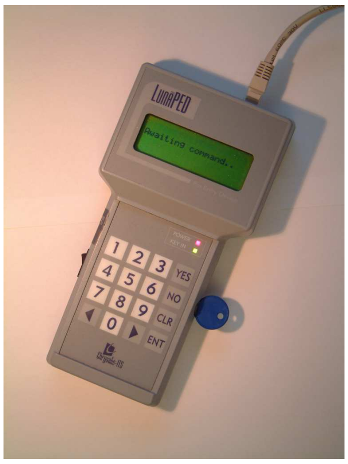

Fig. 2. The Luna PIN Entry Device (PED)

The Luna CA3 documentation made us aware that the token did not speak PKCS#11 across its tamper-resistant boundary, but a proprietary security API we called the "Luna API"; PKCS#11 translation was performed in a higher-level library. We also knew that this API had commands to support the cloning of key material between tokens for backup and redundancy purposes, and that this protocol used public key cryptography to allow the devices to exchange keys and set up a secure channel for communication of 'application keys'. We decided to use our non-functional token to learn more about the cloning protocol. Our plan sounded quite simple: *"Open up the card, reverse-engineer the flash, find the cloning protocol, extract the device keys, then impersonate a token in the cloning protocol"*.

3 Finding the Cloning Protocol

3.1 Disassembly

We disassembled the non-functional token, and to our immediate surprise found no potting material, only a red glue-like substance possibly for structural integrity or heat-dissipation. A disassembled token is shown in figure 3. The processor is a StrongARM, supported by 256KB of static ram, and twin flash chips totaling 1MB of non-volatile storage. A quicklogic FPGA was also present, having undetermined purpose. The flash chips were removed in a couple of minutes using a hot air gun to melt the solder, placed in a standard adapter, and the contents read using a reprogrammer into a binary file.

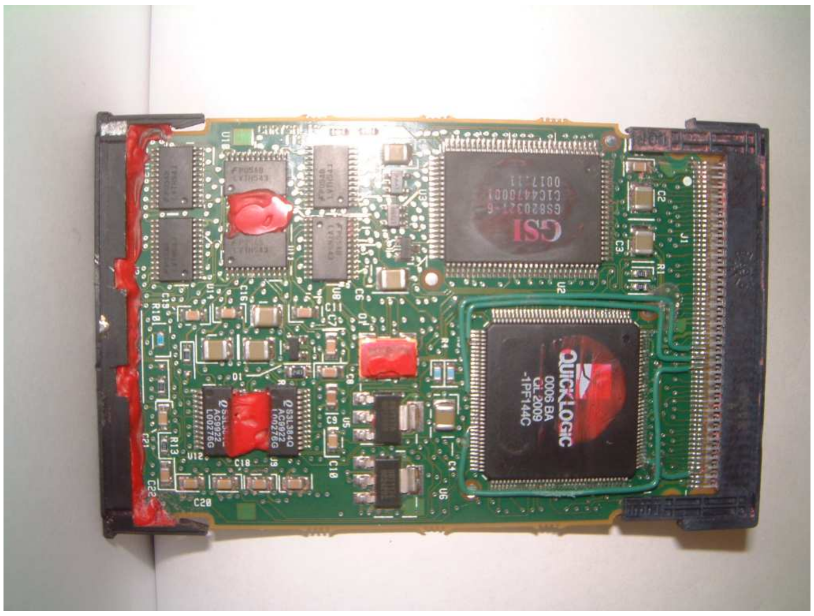

Fig. 3. A Luna CA3 token disassembled

3.2 Making sense of the code

We used the commercial tool IDA (the Interactive DisAssembler) [3] to reverse-engineer the binary. About 300KB of the binary was 32-bit ARM code, 500KB high-entropy data with some fairly regular structure visible, and the remainder was blank. An initial automatic analysis using IDA split the code into subroutines, of which there were over one thousand – the disassembly listing totaled approximately 1700 pages. In order to discover the details of the Luna API functions, both to understanding the cloning process and to analyse their security, we needed to identify the command dispatcher. Our first step towards this was to classify functions, and to chart their flow. One limitation was that IDA's graphing tool could not display conditional or calculated jumps in its diagrams. It quickly became clear that the naming of functions should be carefully thought out, so as to be of maximum help to the reverse-engineer. A typical name viewed in the disassembly might be:

```
C5D2_30SER_BLEV_JANE_do_something_sub1
```

Subroutines were first named simply by their characteristics, and using their address for unique identification (this was IDA's default option). `C5` would represent that the subroutine was called from five locations, and `D2` that its address appeared in data (presumably jump-tables) in two locations. Classification of subroutines into groups was also useful: unusual constants returned in R0 after a subroutine were taken to be error codes, and it quickly became clear that there was human logic behind their values. `30SER` represented that error codes of the form `0x0030XXXX` were commonly returned by that code. Secondly, when a subroutine was called only once, or just a few times from the same parent, it could be conceptually grouped together as part of its caller.

We reasoned that the command dispatcher would be easily identifiable through its high fanout and probable use of case switches (implemented in ARM machine code as a shifted add of an index to the Program Counter register).

```
ADDLS         PC, PC , R0, LSL\#2 ; switch 0xC ways
```

47 different switch statements were found ranging between five and more than twenty cases. We found a page in the appendix of a Luna Security Policy document listing the access permissions required for different Luna API commands – but most important to us revealing their names, and classifying them into modules. However, none of the subroutines with switch statements had both a neat correspondence between the number of cases in the switch and the number of commands in a module, *and* and appropriate fan-out of subroutines.

Working within a team, and trying to keep track of the vast number of unknowns took its toll. Though all the routines were characterised, the functionality of very few had been identified – after several weeks only a couple of subroutines that adjusted parity on DES key material had been identified. The stumbling block was trying to hold a mental image of the interactions between the vast array of subroutines, and graphing was rather ineffective when viewing the entire program structure. We soon realised the importance of naming functions – not just by characteristic and discovered purpose, but also with arbitrary but memorable names, such as common human first names. Binding partial knowledge of functionality to names – FRED, GEORGE, JANE or JULIET – made human memory much better, and aided team discussion (even if it did make for rather bizarre conversation). We soon developed our own two golden rules of reverse-engineering:

– Do what you can.
– Give everything a name.

The command dispatcher was eventually found (and identified to be 'LUCY') through persistence and fresh re-study of the same switch code. A suite of dispatchers were found, corresponding to each module, but (crucially) with very different implementations of the command code processing – this revealed our first clue to the mindset of the designers: modules were written by different authors, with little supervisory architectural guidance that would have yielded consistent implementations of the command decoders. Later on, it transpired that second-guessing the mindset of the designers would become an important reverse-engineering skill. In the mean-time, 90% of the commands were successfully identified and named according to the Luna API command page, including those corresponding to the cloning protocol.

4 Understanding the Cloning Protocol

We now had the code for three commands:

– `LUNA_CLONE_AS_TARGET_INIT`
– `LUNA_CLONE_AS_TARGET`
– `LUNA_CLONE_AS_SOURCE`

but as the lower-level resource subroutines beneath the commands were not at all annotated, the code alone did not reveal much about the protocol. We prioritised the understanding of the protocol (as the protocol design would have a profound impact upon attack strategies for cloning), and decided to intercept communications between the source and target cards, and use the captured data to try and understand the data flows within the protocol, as well as illuminate the processing performed in the ARM code. To this end, we used a trivially modified PCMCIA extender card costing about $100 to present the bus pins of one card for interception, and hooked up a Tektronix TLA711 logic analyser.

Source	*Target*
`LUNA_FIND_OBJECTS`	`LUNA_FIND_OBJECTS`
`LUNA_GET (slot 0xE)`	`LUNA_DESTROY_OBJECT`
`LUNA_GET (slot 0xF)`	`FIND_OBJECTS`
`LUNA_CLONE_AS_SOURCE`	`LUNA_GET (slot 0xE)`
`LUNA_GET (slot 0xE)`	`LUNA_GENERATE_KEY`
`LUNA_CLONE_AS_SOURCE`	`LUNA_SET_UP_MASKING_KEY`
`LUNA_GET (slot 0xE)`	`LUNA_DESTROY_OBJECT`
	`LUNA_GENERATE_KEY_W_VALUE`
	`LUNA_CLONE_AS_TARGET_INIT`
	`LUNA_CLONE_AS_TARGET`
	`LUNA_GET (slot 0xE)`

Fig. 4. Commands called on source and target during cloning

There was no traffic on the bus whilst the device was idle. When triggered, the host would assert 16 bytes of data on the bus lines in each clock cycle, and the device would reply with the constant ‘`FTSI`’, followed by the response data. We wrote some scripts in python to process the logic analyser traces, spot the

FTSI signal and format the data in human-readable form. Interception of the bus data proved very fruitful, especially once the command codes had all been identified. It seemed that the cloning process was driven by the host PC, and was repeated for each key copied. We discovered that the cloning process contained more than just the challenge and response that was implied by the three cloning commands. A list of commands performed during cloning is shown in figure 4. In particular numerous calls to `LUNA_GET` with various parameters were observed, and it soon became clear that the data returned from a `LUNA_GET` with parameter `0xF` was fed as an argument to `LUNA_CLONE_AS_TARGET_INIT`. The outputs of this command were passed to `LUNA_CLONE_AS_SOURCE`, whose data was returned to `LUNA_CLONE_AS_TARGET` – all in all three message passings.

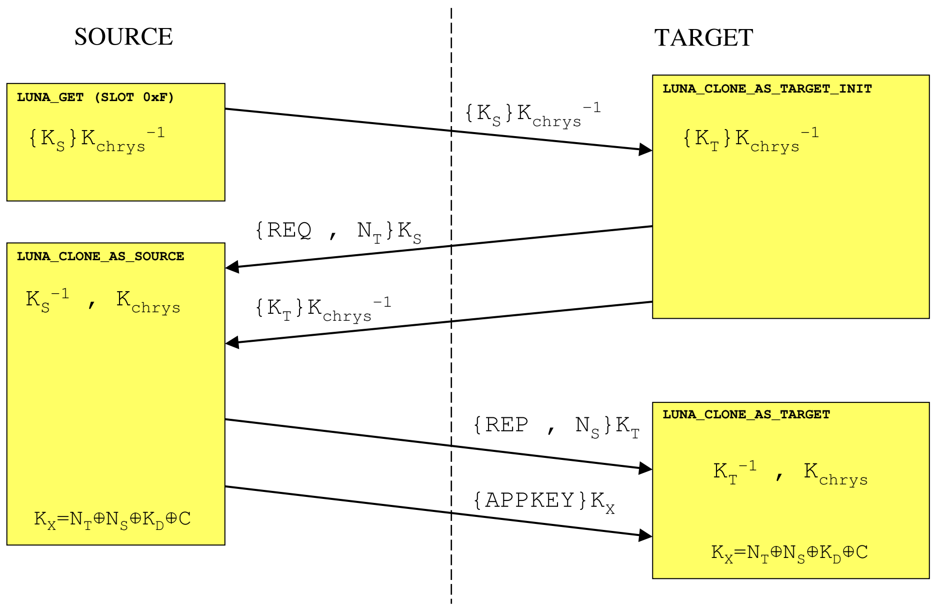

Fig. 5. The cloning protocol

In order to understand the protocol better, the focus now had to shift to understanding the format of this data, and following through its interpretation using the ARM disassembly of the cloning commands. In parallel we worked on annotation of the cryptographic subroutines in the device, which progressively illuminated the structure of the protocol. After a week or so our high-level guesses about the protocol had been corroborated. Figure 5 shows the protocol in normal operation.

There are three distinct exchanges, though the latter two consist of two messages each. All tokens store internally a public key belonging to Chrysalis, which is used to sign certificates attesting to the correspondence between a public key and a real-life Luna CA3 token having a particular serial number. In the first message, the target is sent a certificate held by the source, attesting to its public key, K_S, where K_{chrys}^{-1} is a signature under Chrysalis' private key.

In the second step, the target verifies the certificate, and then generates and sends a nonce N_T encrypted under the public key of the source token. It also forwards its own certificate.

In the third step, the source verifies the authenticity of the target, then generates and sends a nonce of its own N_S. It then creates a one-time symmetric transport key using both nonces and the secret data from the red 'domain' datakey (which is provided to the devices during initialisation). This symmetric transport key is used to encrypt the application key (i.e. the particular PKCS#11 key currently being clones), which is sent in the final message. Finally the target then decrypts the nonce from the source and re-creates the symmetric transport key, and decrypts the incoming application key.

It seemed intuitively clear at this stage that in order to use the cloning protocol to extract keys in the clear or to migrate to another device, the private key corresponding to the public key from the existing certificate had to be extracted from the disassembled token. However, the location of this private key remained unknown.

5 Breaking the Cloning Protocol

5.1 The Luna Mysteries

The cloning routines were by now well-annotated, as well as the low-level crypto primitives, but the encrypted and clear key material they used emerged from a tract of unknown subroutine space. The subroutines were of course named, and two groups of routines in particular (quite literally) held the key – LEELA and JADE.

LEELA was a suite of three or so functions which loaded and saved key material and valuable device data from local buffers and transferred it to memory which lay outside our incomplete understanding of the memory map of the device. LEELA took a *slot identifier* as a parameter, then walked a data structure to find the corresponding data and copied it into the local buffer. Many of the `LUNA_GET` calls corresponded very closely to retrieving and dumping particular 'LEELA slots' verbatim – for instance one of the slots was deduced to contain the public key attestation certificate returned in a `LUNA_GET (slot 0xF)` call.

We reverse-engineered the part of the cloning routine that used the module private key to decrypt the nonce sent from the other token, and found that it was stored in LEELA slot 0xD (encrypted with a 3DES key returned by subroutine JADE). Our first suspicions were that the slots were mapped into memory differently because they corresponded to a sensitive data store separate from the system flash, for instance within the FPGA, whose contents would have been much more difficult to extract than conventional memory chips. However, a colleague examining the LEELA save routine identified it as characteristic of a incremental flash write – it would progressively store new versions of ‘slots’, and garbage collect once the flash space was full. This implied that the slot data really was present within the binary file we had extracted. This renewed our confidence, and we decided to search the entire flash for data matching the LEELA characteristic – a linked list, with an additional field containing values in the set of common slot IDs. We were in luck: two areas matched the characteristic (the primary was at offset `0x88000`). We now had the ciphertext corresponding to the encrypted device private key.

However, JADE was even more mysterious than LEELA: it took no arguments at all, and appeared to navigate a complex data structure held in RAM to retrieve the key. We theorised these routines accessed the main store of objects using an object handle corresponding to some special device key. But without a sample of the data structure in RAM, it would be very difficult to locate the code that created this object, as many many subroutines created objects for one reason or another. The solution lay in a logical guess and some more high-level characterisation of the target code. We guessed that the login routines might read this clear key straight from a data key and load it into RAM, maybe after performing some PIN verification operation, and began to examine them closely. The login routines turned out to be quite difficult to understand, because there appeared to be several legacy modes of operation for login, for instance by sending the PIN directly from the host; these made it difficult to see exactly which code was really being executed in our live devices. The crucial breakthrough happened when we noted that the values of the error codes returned by the JADE functions were unique (and did not map directly to the usual `GENERAL_ERROR` PKCS#11 error code), and we visually searched for similar value error codes in the login code, and found our target.

5.2 First Strategy : Extracting the Device Private Key

We had most of the data we needed, but a perfect re-implementation of the login cryptography was required to extract a clear copy of the device private key. We started writing our own code in ‘C’ to operate upon the encrypted data successfully extracted from the LEELA slots. A sub-component of LEELA slot 0xF contained an encrypted block that was decrypted with a 3DES key to reveal the redundant string ‘`GESC_FIX`’, and the “JADE key”. The key used to decrypt slot 0xF was derived from raw data returned by the PED, but here we spotted something rather strange: the raw data was hashed sequentially five times using MD5. This additional hashing appeared to be quite unnecessary, and it was not clear whether it was performed to add obscurity, out of ignorance, or for some other more mysterious purpose. We later found more evidence of unnecessary complexity added to quite valid cryptographic techniques – for instance the exclusive-ORing of the 32-bit constant `0xDEADBEEF` into completely random key material. We soon began to informally refer to any apparently unnecessary processing as ‘dead beef’.

We still needed to determine the content of the raw data from the PIN entry device, so we intercepted and decoded the PED communications link using the logic analyser. Understanding this protocol delayed us significantly: the link used a bizarre proprietary protocol consisting of nibbles transmitted along a data line, with a sporadically asserted 'data valid' signal on a second line (we later discovered that this irregularity was because the serial protocol was implemented entirely in software, and the loop time was data-dependent). Comparison of this data with that directly extracted from the data keys (using a commercial datakey reader) revealed that the PIN entered at the PIN was repeatedly XORed with the data on the datakey.

We began to implement decryption of the encrypted slot using the data from the blue key, but soon realised there was a conundrum – if the private key belonging to the device was held encrypted under the data on the blue key, how could it exist when the device was uninitialised (and had no blue key)? We began to think that we may have entirely misinterpreted the actions of JADE. However, the entirely undocumented grey datakey which was only used during initialisation gave us the missing password – it contained a default password for uninitialised tokens, which decrypted a backup copy of the private key, sitting adjacent to the one encrypted under the blue key in the LEELA slot.

Unfortunately, it proved very difficult to get our re-implementation to correctly decrypt the JADE key (we could observe correct decryption by retrieving the string `GESC_FIX`). There were simply too many unknowns that had to be guessed to completely define the algorithm. Was the IV definitely all zeroes? Had we mis-interpreted the repeated MD5 hashing? Was the MD5 implementation bitwise perfect, or was it some modified internal only version?

5.3 Second Strategy : Customer Verification Keys

We considered emulating the decryption code using ARM ltd.'s "ARMulator", but taking on board yet another new tool appeared a substantial undertaking with no guarantee of success. Instead, we fell back to our first rule of reverse-engineering – "do what you can" – and decided to take a look inside the host DLLs to try and find the routines calling the cloning commands – maybe these could shed some light upon the process, and after all, in order to complete the emulation of a token, we needed a way to intercept and modify traffic going to the source token. We discovered that there were a number of undocumented Chrysalis extensions to PKCS#11 that did other interesting things as well as permit cloning. In particular, we saw

- `CA_SetCloningDomain`
- `CA_SetTokenCertificateSignature`
- `CA_ClonePrivateKey`
- `CA_GenerateTokenKeys`

`CA_SetTokenCertificateSignature` sounded most interesting – it had previously not made any sense when we did not know the important role played by

Chrysalis device certificates in the cloning protocol. We reverse engineered the DLL and found that a Luna call `LUNA_LOAD_CUST_VERIFICATION_KEY` was the essential result of calling `CA_SetTokenCertificateSignature`. Our pragmatic approach of staying focussed on the protocol had meant that loading of customer verification keys sounded irrelevant, and quick inspection of the functionality of the command showed that all it did was to store the data sent to it in a LEELA slot of unknown function. But this time we dug deeper, and were quite amazed to find that the "Customer Verification Key" (CVK) was an alternative public key which could be used for verifying the authenticity of a certificate attesting to the destination token public key. And what was more, we could load a CVK of our choice without it requiring any special authorisation, other than that of the Security Officer. A new plan for key extraction developed, which ran as shown in figure 6. The plan went as follows:

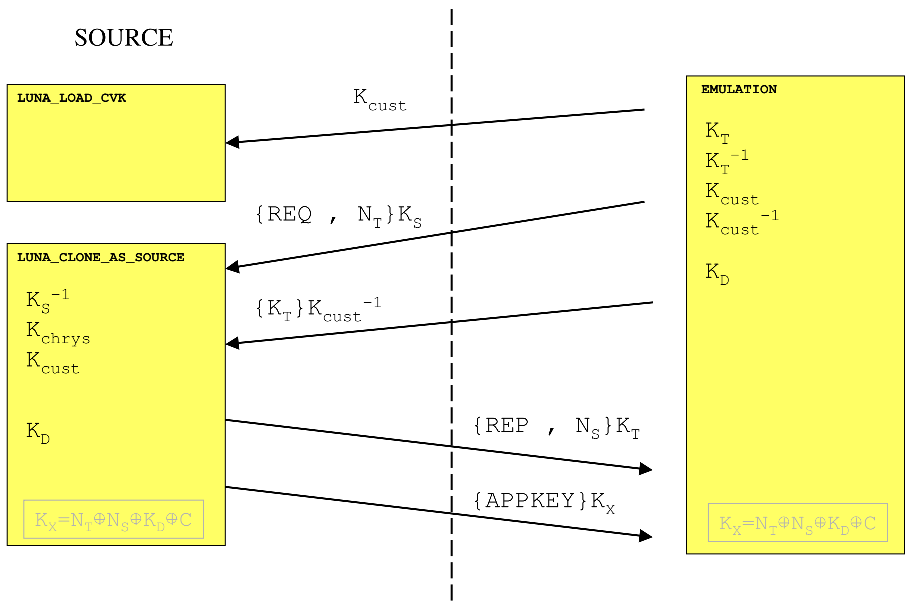

Fig. 6. Extracting to clear using the CVK

1. Generate a known customer verification keypair – K_{cust} and K_{cust}^{-1}.
2. Load the public half into the source token as the CVK.
3. Choose an arbitrary known nonce N_T and send this to the source.
4. Generate a known emulated target device keypair – K_T and K_T^{-1}.
5. Use K_{cust}^{-1} to sign a certificate attesting to the validity of K_T, and send this certificate to the source also.
6. Receive the source nonce encrypted under the chosen K_T.
7. Combine the nonces and red key to make K_X and decrypt $APPKEY$.

It seemed our task annotating the ARM code was almost done, and we now had to participate in the protocol ourselves. We used the `CA_ClonePrivateKey` routine to perform a cloning operation between a real source and target, but cautiously substituted in our own messages to the source token, by pausing the debugger and changing the contents of the memory. We progressively built our messages, exploiting the quite rich set of error codes passed back by the Luna API to help us when we went wrong. The first challenge was to encrypt a nonce of our own and substitute in this message. This was easy to do as it only required knowledge of a public key.

Our next step was to modify the certificate from the target to include our own public key, and observe everything work properly except for the signature failing to verify. Finally, we reverse-engineered the processing done by the `CA_SetTokenCertificateSignature` and discovered that the public key to be loaded as a CVK could be provided in PKCS#11 attribute list form, and it would automatically flatten it down into a CVK entry. However, we needed to execute the `CA_SetTokenCertificateSignature` command under a debugger and with extreme caution, as it went on to call Luna API commands with unknown purposes, firstly apparently loading some unknown certificate over the top of the existing one in the source token (potentially damaging its ability to clone keys using the normal method ever again), and then calling a Luna API command whose name and function was entirely unknown!

One the correct public key had been loaded as the CVK, we were pleasantly surprised to find that the `LUNA_CLONE_AS_SOURCE` call terminated with no errors reported – the application key was now encrypted under a target nonce of our choice, and the source nonce encrypted under a known key. One final hurdle stood between us and correct decryption of the application key – calculation of the "Key Cloning Vector" (KCV) from the raw data on the red data key.

5.4 Making the Key Cloning Vector

During initialisation of a Luna CA3 token, the PED gives the user the option to either generate a new domain key (i.e. KCV), or to use an existing key. For cloning to be possible, one token in a group must have generated a fresh domain key, and the rest must be presented with it during initialisation to be admitted to the domain. We found the code that processed data imported from a red datakey in the `LUNA_INIT_TOKEN` command, but it was the most bizarre code yet. Figure 7 gives a flavour of the excessive processing performed to translate 80 bytes of random data on the datakey to the 24 bytes of a 3-key 3DES key.

This was the probably the ultimate example of "dead beef", and added approximately a week to the time taken to re-implement the protocol. Two of our team made clean-room implementations of the code according to the assembler, and cross checked them against each other, gradually eliminating bugs and uncertainties until the implementations matched. We then used an ARM emulator to double check our answer. Finally, in November of 2003, we eliminated the last bug (counting a string terminator twice when calculating the offset used to

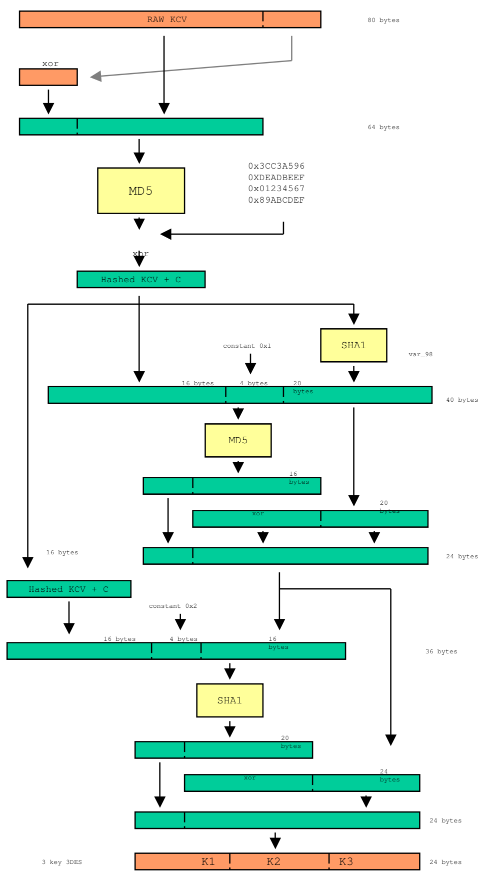

Fig. 7. The KCV calculation algorithm

retrieve the source nonce from its data structure), and low entropy data emerged from the 3DES decryption function. We had an application key in the clear!

6 How Long Does it Take to Reverse-engineer an HSM?

The table below gives a breakdown of how our time reverse-engineering was spent. Surprisingly, it took less than a day to complete the physical component of the attack on the Luna CA3 – the device was not even potted. Lack of obfuscation of the processor operation (for instance by permuting the address lines to obfuscate the RAM contents) also meant this stage was very quick.

Task	*Duration*
Disassembly	Less than a day
Initial Mark-up of Code	One month
Locating Command Dispatcher	One week
Annotating Commands	One month
Annotating Cloning Protocol	One month
Annotating Crypto Routines	One month
Intercepting PCMCIA Bus	One month
Decoding PED Protocol	Several days
Decrypting Private Key	One week
Initial Mark-up of Host DLL	Two weeks
Re-implementation of Protocol	Three weeks
Total	***7 months***

The effort involved in reverse-engineering the ARM code was substantial – it took several months of work to become familiar with the code, find the command dispatcher, and identify general structure in the code. During annotation of specific commands and the cloning protocol, "security through obscurity" came into play. Approximately a third of the effort spent during this period was spent dealing with dead beef or legacy code. Dead beef code was not actually a large hurdle to *understanding* of the code, but it was time-consuming to annotate, and very hard to accurately re-implement. The toughest code to understand was that including legacy modes of operation – as the redundant code looked *extremely plausible* – it had once been the genuine code.

The peripheral activities to the ARM code analysis – intercepting buses, re-implementing protocols and understanding the host driver software were not individually large expenditures, but added up to a fair proportion of our total time. Learning exactly when to invest time building tools to help with analysis is a crucial skill required to maximise efficiency.

7 Summary

This work enables owners of key material stored in Chrysalis Luna CA3 products to extract keys to the clear, or migrate them into other manufacturer's devices, should they chose to be compatible with the protocol. Note however that the authorisation of the token Security Officer is still an integral part of the procedure, and the Luna CA3 has not been proved insecure. We are currently in dialogue with SafeNet (the new owners of Chrysalis) with a view to feeding forward suggestions for improvements to the architecture.

References

1. M. Bond, "Attacks on Cryptoprocessor Transaction Sets", CHES 2001, Springer LNCS 2162, pp. 220-234
2. Chrysalis-ITS Inc. http://www.chrysalis-its.com
3. The Interactive DisAssembler (IDA) http://www.datarescue.com