A Note on EMV Secure Messaging in the IBM 4758 CCA

Ben Adida Mike Bond Jolyon Clulow Amerson Lin Ross Anderson
Ron Rivest

March 22, 2005

Abstract

We present a set of attacks against two recently-added API calls in the IBM 4758 CCA v2.5. We show how the addition of these API calls effectively breaks the security policy of the 4758: the plaintexts of secret keys and PINS are exposed, and the integrity of secure messages is compromised.

1 Introduction: 4758 Basics

The IBM 4758 CCA is used in secure banking communcations to provide methods for safely using secret data to perform well-constrained operations. For example, a data encryption key can be stored in unsafe memory if it is encrypted (by the 4758) using a Master Key (KM). The operator can then perform a data encryption using this wrapped key and the data to encrypt as inputs to the 4758:

$$\{K_1\}_{KM \oplus DATA}, m \longrightarrow \{m\}_{K_1} \tag{1}$$

The important concept to note is that the operator of the 4758 never sees K_1 in the clear. We skip many details about the 4758, but we note the following important and pertinent general concepts:

- Wrapped keys, like $\{K_1\}_{KM \oplus DATA}$, are meant to be kept secret from the operator.
- Messages exported by the 4758, like $\{m\}_{K_1}$, should be difficult to forge by someone who doesn't possess the relevant communications key K_1 in the clear.

2 The `Secure_Messaging_For_Keys` Command & Security Policy

The `Secure_Messaging_For_Keys` API call is defined in v2.5 of the 4758 CCA. This new capability exists for the purpose of setting up secure transmission of keys between the 4758 and an EMV[1]-compliant smart card. In other words, the `Secure_Messaging_For_Keys` capability is a special kind of key export.

Because this export functionality is specific to EMV smart cards, the API allows exporting only towards specially-typed keys: in this case, a key typed as $SECMSG$. Such a key would be stored as $\{K_2\}_{KM \oplus SECMSG}$

In order to accomodate various manufacturer-specific message formats, the `Secure_Messaging_For_Keys` expects a message format template as well as an offset pointer into this template to indicate where the exported key data should be placed. This template is limited in size to 4096 bytes.

[1]Eurocard/Mastercard/Visa

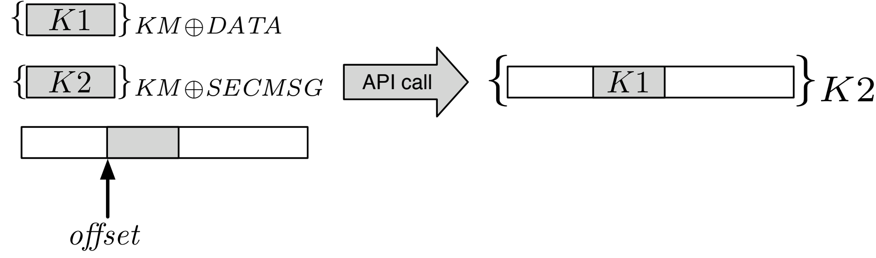

Figure 1: The API Call

We can now provide all details of this API call:

$$m, \mathit{offset}, \{K_1\}_{KM \oplus T}, \{K_2\}_{KM \oplus SECMSG} \longrightarrow \{m[K_1 : \mathit{offset}]\}_{K_2} \tag{2}$$

- m: the message template, a series of bytes to be used in preparing the plaintext.
- offset: the offset within m where the key material should be placed.
- $\{K_1\}_{KM \oplus T}$: the wrapped key to export. Control vector T should indicate an exportable key.
- $\{K_2\}_{KM \oplus SECMSG}$: the wrapped key under which to wrap the exported data.
- $m[K_1 : \mathit{offset}]$: the template plaintext m interpolated with key material K_1 at offset offset.
- $\{m[K_1 : \mathit{offset}]\}_{K_2}$: the wrapped interpolated template plaintext.

3 The Attacks

First we show how this API call effectively gives the operator access to an Encryption Oracle for any key typed as $SECMSG$. Then, we show how to use this encryption oracle to spoof a transmission to a smart card. In a coup de grace, we use the templating capability of the API call to expand this oracle into a partial-key dictionary attack mechanism. We use this approach to rapidly crack any key marked as exportable.

3.1 Construction of an Encryption Oracle

The `Secure_Messaging_For_Keys` performs encryptions in either CBC or ECB mode. These modes both have the property that a ciphertext can be truncated to create a ciphertext of an identically truncated plaintext – as long as the truncation is along a block limit.

Thus, one can use `Secure_Messaging_For_Keys` as an *encryption oracle* using any key of type $SECMSG$, as long as the plaintext is one block less than the maximal template length (4096 bytes). We construct this encryption oracle on input m:

1. create a template t by extending m by a single block, e.g. the 0-block.
2. set the offset to $|m|$, where $|m|$ does not include the newly added single block.
3. perform the call to `Secure_Messaging_For_Keys` with any exportable key K_1 under key K_2 of type $SECMSG$, obtaining a ciphertext c

4. note that the 4758 will fill in the last block (indicated by *offset*) with K_1, leaving the entire m component of t untouched.
5. chop off the last block of c.
6. the result is $\{m\}_{K_2}$.

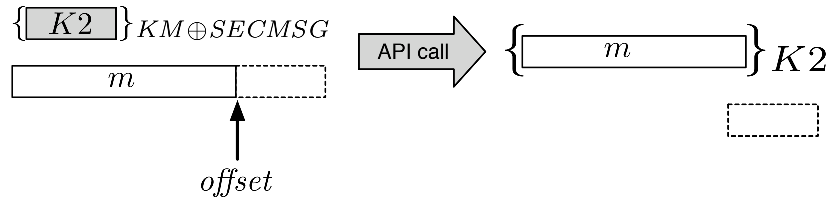

Figure 2: Construction of an Encryption Oracle. The last block is chopped off.

Note that this attack uses `Secure_Messaging_For_Keys` in the strictest possible way, with *offset* a block multiple, even though this isn't required. Unless the documentation is severely flawed, this encryption oracle attack should function.

3.2 Injecting Spoofed Export Messages

Given the encryption oracle above, it's easy to create any export message of length no greater than 4088 bytes. Thus, taking any template t, one can "fill in" any known key material at the appropriate location, then use the oracle to encrypt this fully-known message under a key of type $SECMSG$.

If the receiving smart card uses this key as a secure transmission key (the most likely use case for this API call), the security of these later transmissions is compromised.

3.3 Cracking Any Exportable Key

This next attack assumes that a key can be interpolated into a template at an *offset* that is not a multiple of block length. Certainly, the documentation makes no mention of such an alignment requirement, and one could assume that the EMV message format does not confirm to block-alignment of keys.

The attack proceeds as follows:

1. set up 256 plaintext blocks of the form `0x00000000000000yy` where `yy` varies from `00` to `ff`.
2. use the encryption oracle on all of 256 plaintext blocks to generate a dictionary of 256 ciphertexts.
3. perform an API call with a template of all 0's and an *offset* of 7, exporting a key K_1 that we wish to attack.
4. compare the first output block to our dictionary of 256 attacks. This comparison yields aa, the first byte of K_1.
5. repeat the process with 256 plaintext blocks of the form `0x000000000000aayy`, with an *offset* of 6. This will yield the 2nd byte `bb` of K_1. By continually shifting the key over by one block, we can extract the entire key. For a k-byte key, it takes $256k$ queries to extract the whole key.

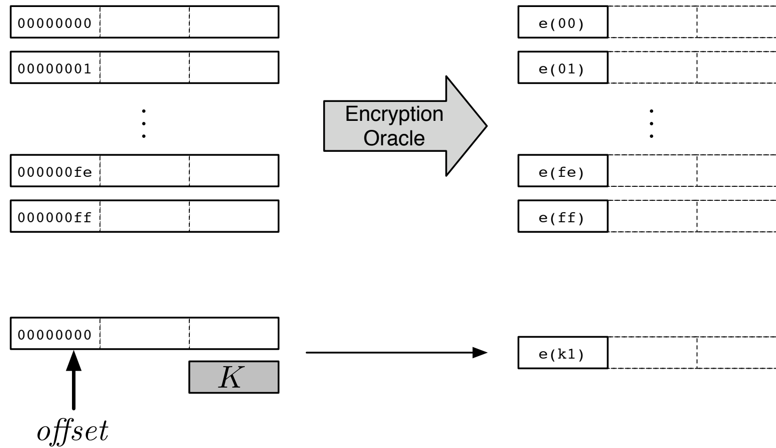

Figure 3: In the key-shifting attack, A 256-element dictionary is built up for each byte of the key that we want to check.

4 The `Secure_Messaging_For_PINs` Command

The same 4758 API specification includes the command `Secure_Messaging_For_PINs`, which is similar to `Secure_Messaging_For_Keys` except that it is used for transmitting PINs instead of keys. This API call exhibits the exact same *template and offset* construction.

Thus, the techniques described above for creating an Encryption Oracle can be used with `Secure_Messaging_For_PI` too. This allows a malicious operator to:

1. recover the plaintext of an encrypted PIN block, or
2. inject a known PIN block into a secure PIN message.

5 Conclusion

The 4758 documentation clearly implies that `Secure_Messaging_For_Keys` and `Secure_Messaging_For_PINs` are meant to provide *secure transmission* of keys and PINs. We have shown how to:

1. Inject a known key or PIN into the smart card.
2. Extract a key or PIN exported from the 4758.

These attacks work in both CBC and ECB modes, with single- or double-size wrapper keys. The only assumption for the second class of attack is that the *offset* can be non-block-aligned – an assumption that seems credible, given that the documentation goes to great length to specify that the message block must be a multiple of 8 bytes, but says nothing about the offset.

It seems the security policies of `Secure_Messaging_For_Keys` and `Secure_Messaging_For_PINs` have been fully compromised.