On the Security of the EMV Secure Messaging API

Ben Adida Mike Bond Jolyon Clulow Amerson Lin Ross Anderson
Ronald L. Rivest

November 4, 2005

Abstract

We present new attacks against the EMV financial transaction security system (known in Europe as "Chip and PIN"), specifically on the back-end API support for sending *secure messages* to EMV smartcards. We examine how secure messaging is implemented in two major Hardware Security Modules (HSMs). We show how to inject chosen plaintext into encrypted traffic between HSM and smartcard. In the case of IBM's implementation, we further show how to retrieve confidential data from within messages by combining the injection ability with a partial dictionary attack. Such attacks could compromise secret key update of a banking smartcard, permitting construction of a perfect counterfeit, or could change the card's PIN to a value chosen by the adversary.

We discuss the issues underlying such security holes: the unwieldy primitive of cipher block chaining (CBC) has much to answer for, as does an ever-present tension between defining API functionality too specifically or too generically. We stress the importance of using secure primitives when designing security APIs, particularly because their functionality so closely resembles the theoretical adversarial model of Oracle-access.

1 Introduction

EMV[1] is the new electronic payment system designed by VISA and Mastercard [EMV04], which has been deployed widely throughout Europe in the last 12 months, and aims to eventually supersede magnetic stripe based systems. Customers carry smartcards which, upon payment, engage in cryptographic protocols to authenticate themselves to their issuing bank. At the issuing bank (the "back end" for short), the Hardware Security Modules (HSMs), which are tasked with PIN storage and verification for ATM networks, have been extended to provide new EMV security functionality. The HSMs now authenticate and manage the massive card base, ensuring security in an environment with the ever-present risk of insider attack.

Specifically, the back-end HSMs expose a *security Application Programming Interface (security API)*, which the vast untrusted banking application layer can use to perform cryptographic operations, and which enforces a *security policy* on the usage of the secret data it handles. In the last five years, the security of HSM APIs has come under close analysis from the academic community (see [ABCS04] for a treatment of the topic), and, recently, a number of HSM manufacturers have made their EMV functionality available for study.

The new EMV functionality includes three basic classes of commands: firstly, those to verify authorisation requests and produce responses or denials; secondly, those to manage the personalisation of smartcards during the issue process; and thirdly those to produce *secure command*

[1] The EMV standards are named after the names of their original contributing corporations: Europay/Mastercard/Visa.

messages which are decrypted, verified and executed by smartcards for the purpose of updating security parameters in the field.

This paper concentrates on this last class: *secure messaging* commands. Such commands are used for many purposes: to change the PIN on a card, adjust the offline spending limits, replace cryptographic keys, or to enable and disable international roaming.

1.1 Our Results

We present two attacks, which, together, completely undermine the security of EMV secure messaging, assuming a corrupt insider with access to the HSM API for a brief period.

Our first attack allows the injection of chosen plaintext into the encrypted field of a secure message destined for an EMV smartcard (this could be used to update the card's PIN or session key). The second attack discloses any secret data that is potentially exportable to a card, for instance a unique card authentication key. Only one device was found vulnerable to this second class of attack, but it is particularly significant, because it is *passive* with respect to the card. Therefore, if such an attack were used to defraud a bank, it would be much harder to trace how it was committed.

Both attacks exploit the malleability of the CBC mode of operation, combined with an overly generic and extensible security API specification. CBC specifically allows for ciphertext truncation, which, combined with one HSM's template-encryption functionality, yields an encryption oracle. This encryption oracle immediately enables adversarial data injections. In addition, using this encryption oracle and the ability in this same HSM to specify the secure-data injection offset, the attacker can split the secret data across a CBC block boundary and perform an offset-shifting attack to discover the contained plaintext one byte at a time.

The other HSM we examined did not permit injection of any chosen plaintext, so the previous encryption oracle attack does not succeed. However we expose a different attack based on the lack of key separation between CBC encryption keys and MAC keys, permitting the MAC oracle functionality to be warped into an encryption oracle.

Similar attacks against CBC within security protocols have been described in recent literature [**?**], but such protocols tend to mediate long interactions between human parties, each of which can detect protocol-deviating attacks and abort before any damage is caused. The HSM operating environment has only one human party – the attacker – and the HSM generally presents an interface of short, atomic API calls. HSMs are not designed to thwart or even detect malicious sequences of legal API calls: instead, the designer of the API is responsible for ensuring there are no possible malicious sequences of API calls.

1.2 Previous & Related Work

CBC Mode and Attacks. CBC mode was defined in 1983 [**?**] to address obvious issues of extreme malleability in straight ECB mode. The analysis of CBC (among other DES modes of operation) wasn't formalized until 1997 [**?**], though it was always known that CBC mode is vulnerable to chosen-ciphertext attack and that CBC ciphertexts are malleable. CBC MAC was analyzed earlier [MORE HERE].

Two significant practical attacks against CBC stand out in the literature: the Vaudenay attack [**?**], which targets validity checking on the CBC padding mechanism as an information-leaking Oracle in common protocols like SSL/TLS, and the Rogaway attack [**?**], which exposes a chosen-plaintext attack against CBC when the initialization vector is known prior to plaintext selection (or

adversarially chosen). One of our attacks is related to this Rogaway attack. There are numerous other documented attacks against CBC encryption [**?**] and CBC MAC [].

HSM Attacks. The first academic analysis of HSM security is by Anderson [And94] and described the known failure modes of ATM banking systems. A subsequent paper [AK97] describes the deliberate addition of a dangerous transaction to a security API. In 2000, Anderson remarked [And00] that while such failures pertained to a *single* bad transaction, and raised the harder question: "So how can you be sure that there isn't some chain of 17 transactions which will leak a clear key?" The idea of an API attack was born as *an unexpected sequence of transactions which would trick a security module into revealing a secret in a manner contrary to the device's security policy*. Since then, a large number of significant API attacks against HSMs have been published [Bon01, BA01, Bon04, Clu03, BC04, BC05]. A recent survey paper [ABCS04] provides a comprehensive overview of publications in this area.

1.3 Organisation of this Paper

In section 2, we take a look at the EMV Secure Messaging commands and their implementations. Then in sections 3 and 4 we describe our attacks in detail. We provide approaches to fixing the problem in section 5 before discussion and some concluding remarks in section 6.

2 EMV Secure Messaging Support

A completed EMV secure message looks as follows:

$$CommandCode||Parameters||EncryptedField||MoreParameters||MAC \tag{1}$$

On receiving such a command a smartcard would verify the MAC which is calculated over all previous data, decrypt the encrypted field, then execute the command. Each card has encryption and MAC keys that are unique to it. Freshness is achieved by a separate mechanism that derives a specific session key for the message from the main keys, but is outside the scope of this discussion. Before we demonstrate how such a message would be constructing using HSM API calls it is helpful to briefly recap the HSM paradigm for working with keys.

2.1 Security API Fundamentals

Early HSMs had only limited storage within their tamper-resistant boundaries, but large numbers of keys had to be managed, thus the enduring paradigm is to store them externally in encrypted form. The HSM need then only retain a *master key*, KM, and use this to encrypt other keys or data. To ensure that keys and data cannot be abused in algorithms for which they were not intended, the HSM *type system* ensures key separation. We abstract away the specific mechanism and represent the external storage of a key K_1 of type T (i.e. to be used for purpose T) by $\{K_1\}_{KM/T}$. For example, a key K_1 used for general purpose encryption of messages would be stored externally as $\{K_1\}_{KM/DATA}$, and an HSM operator could request a message m to be encrypted using the following API call:

$$\{K_1\}_{KM/DATA}, m \longrightarrow \{m\}_{K_1} \tag{2}$$

The important concept in the above transaction is that the operator of the HSM can have access to perform cryptographic operations using K_1, but never sees K_1 in the clear. When such a key

must be transferred between devices, some kind of re-encryption operation is required – known as a *key export*. From the perspective of the confidential data, the EMV secure messaging commands are just a specialised form of key export. We represent a typical export operation as shown below:

$$\{K_1\}_{KM/DATA}, \{K_S\}_{KM/EXPORT} \longrightarrow \{K_1\}_{K_S} \quad (3)$$

On the left of the arrow, we see the inputs K_1 and K_S, supplied in encrypted form. The re-encryption is performed by unwrapping all the inputs and verifying their type information permits them to be used in this way, then rewrapping K_1 using K_S itself.

Now let's examine specific implementations of commands to create a secure message containing a newly exported key, but preserving the secrecy of that key throughout the transaction.

2.2 CCA Case Study

IBM's Common Cryptographic Architecture (CCA)[Int03], as implemented on various IBM HSM models, provides the secure messaging command called `Secure_Messaging_For_Keys`. This API call exists for the purpose of setting up secure transmission of keys (or other sensitive information) between the HSM and EMV-compliant smart card. In other words, the `Secure_Messaging_For_Keys` capability is a special kind of key export, where the HSM unwraps an exportable key and re-wraps it under a new transmission key. Because this export functionality is specific to EMV smart cards, the API call only accepts export keys having a specific type *SMSG*, which certifies that they are appropriate export keys for this purpose. In the CCA, keys of this type are produced up-front using the `Derive_Key` command. For example, a smartcard session key K_2 would be stored externally as $\{K_2\}_{KM/SMSG}$. This command aims just to achieve the construction of an encrypted field for the secure message, and the MAC over the message is dealt with using a separate command invocation.

In order to accomodate the range of possible secure command messages within EMV, the API call accepts an encrypted field template and an offset pointer into this template to indicate where the clear key data should be placed. This template is limited in size to 4096 bytes.

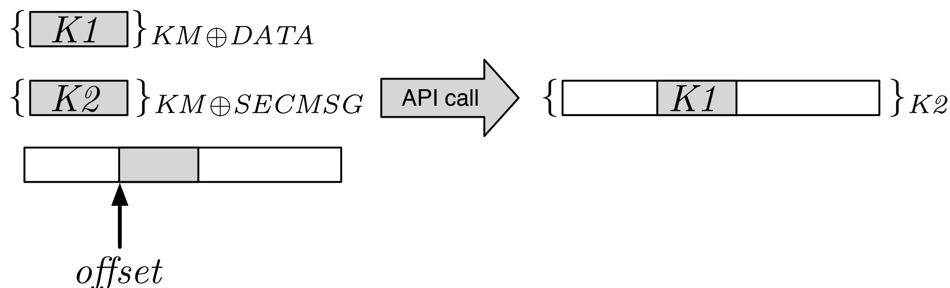

Figure 1: The API Call

Here is the `Secure_Messaging_For_Keys` call in detail:

$$template,\ \ offset\ \ , \{K_1\}_{KM/T}, \{K_2\}_{KM/SMSG} \longrightarrow \{template[K_1 :\ \ offset\ \ ]\}_{K_2} \quad (4)$$

- *template*: the message template, a byte-string to be used in preparing the plaintext.
- *offset* : the offset within *template* where the key material should be placed.
- $\{K_1\}_{KM/T}$: the wrapped key to export. The key type T should indicate an exportable key.

- $\{K_2\}_{KM/SMSG}$: the wrapped key under which to wrap the exported data.
- $template[K_1 : \ offset \ ]$: the template plaintext *template* interpolated with key material K_1 at offset $\ offset \ $.
- $\{template[K_1 : \ offset \ ]\}_{K_2}$: the wrapped, interpolated template plaintext.

The encrypted output of this command is inserted into the template for the full secure command message, and then passed to the `MAC_Generate` command.

2.3 RG7000 Case Study

Thales eSecurity provide equivalent secure messaging in their RG7000 device [Zax00] (the most widely used device in the community), but crucially they integrate a lot more of the functionality into a single command, which can operate in various modes. The `Secure_Message_With_Integrity_And_Optional_Confidentiality` command (identified more briefly by command code `KU`), can derive the appropriate session keys, export a secret key and MAC the entire message with a single atomic command.

The command has a mode selector because the inclusion of an encrypted data field in a secure message is optional, dependent upon the purpose of the command and the sensitivity of its parameters. In mode 0 it provides only integrity (MAC) functionality, while in modes 1 and 2 it provides both integrity and confidentiality – MAC and re-encryption. In essence, the mode 0 MAC is just the final block of a 3DES CBC operation over the plaintext message using a key typed as secure messaging with integrity (*SECMSGINT*). This key is derived from a supplied master key using transaction and card specific data. For the purpose of this analysis, we abstract away this unnecssary detail and use $\{K_1\}_{KM/SECMSGINT}$ to represent the derived integrity key used in the MAC calculation, and show it as an input to the API call. We can simplify the full command, representing mode 0 operation by the `Secure_Message_With_Integrity` API call:

$$template, \{K_1\}_{KM/SECMSGINT} \longrightarrow mac(template, K_1) \qquad (5)$$

- *template*: the message template, a byte-string over which the MAC is calculated.
- $\{K_1\}_{KM/SECMSGINT}$: the wrapped derived key with which to calculate the MAC.
- $mac(template, K_1)$: the MAC.

Modes 1 and 2 are conceptually closer to the IBM function and provide functionality to facilitate the secure transmission of keys between the HSM and an EMV compliant smart card. They enable the user to rewrap a key K_2 of type *transport key* – $\{K_2\}_{KM/TK}$ – under a secure messaging key shared with the smart card and insert it into the template. The MAC is then calculated on the modified template as before. The difference between modes 1 and 2 is the choice of the secure messaging key used to wrap the supplied transport key. In mode 1, the same secure messaging with integrity (*SECMSGINT*) key that is used in the MAC calculation, is also used to wrap the transport key. In mode 2, a different secure messaging with confidentiality key (*SECMSGCONF*) is used to wrap the transport key, after which the MAC is calculated with the *SECMSGINT* key. Mode 1 is relevant for our further discussions. We refer to it as the `Secure_Message_With_Integrity_And_Confidentiality` and is described below.

$$\begin{aligned} &\mathit{template},\ \ \mathit{offset}\ \ , \{K_1\}_{KM/TK}, \{K_2\}_{KM/SECMSGINT} \longrightarrow \\ &\mathit{template}[\{K_1\}_{K_2} :\ \ \mathit{offset}\ \ ], \mathit{mac}(\mathit{template}[\{K_1\}_{K_2} :\ \ \mathit{offset}\ \ ], K_2) \end{aligned} \tag{6}$$

- *template*: the message template, a byte-string to be used in preparing the plaintext.
- *offset* : the offset within *template* where the key material should be placed.
- $\{K_1\}_{KM/TK}$: the wrapped key to export. The key type TK should indicate an exportable key (i.e., a transport key).
- $\{K_2\}_{KM/SECMSGINT}$: the wrapped derived key under which to wrap the exported transport key and with which to calculate the MAC.
- $\mathit{template}[\{K_1\}_{K_2} :\ \mathit{offset}\ ]$: the template plaintext *template* interpolated with the encrypted key material $\{K_2\}_{KM/SECMSGINT}$ at offset *offset* .
- $\mathit{mac}(\mathit{template}[\{K_1\}_{K_2} :\ \mathit{offset}\ ], K_2)$: the MAC over the interpolated template plaintext.

3 CCA Injection and Extraction Attacks

3.1 Construction of an Encryption Oracle

Gaining access to an encryption oracle is the crucial step towards creating a vulnerability. The CBC mode used in `Secure_Messaging_For_Keys` [2] has an unfortunate malleability property: a ciphertext can be truncated to create a ciphertext of an identically truncated plaintext – as long as the truncation is block-aligned. Thus, one can use `Secure_Messaging_For_Keys` as an *encryption oracle* using any key of type *SMSG*, as long as the plaintext is one block less than the maximal template length (4096 bytes). We can thus construct this encryption oracle on input m:

$\mathsf{EncryptionOracle}\Big(\mathit{plaintext}, \{K_2\}_{KM/SMSG}\Big)$:

1. create a template *template* by extending *plaintext* by a single block, e.g. the 0-block.
2. set the *offset* to $|\mathit{plaintext}|$, which is effectively the beginning of the 0-block just added.
3. perform the call to `Secure_Messaging_For_Keys` using any available exportable key $\{K_1\}_{KM/T}$:

   $$\mathit{plaintext}||\text{“00000000”}, |\mathit{plaintext}|, \{K_1\}_{KM/T}, \{K_2\}_{KM/SMSG} \longrightarrow c$$

   the HSM will fill in the last block *template* (as indicated by *offset*) with K_1, leaving the entire *plaintext* component of *template* untouched.
4. consider the first $|\mathit{plaintext}|$ blocks of c, effectively discarding the last block. This truncated value is simply $\{\mathit{plaintext}\}_{K_2}$, our desired result.

This technique effectively undermines any security merits of the template-fill-in operation of the HSM and lets an operator use any *SMSG* key as a normal key – as if it were being used in the conventional `Data_Encrypt` command.

[2]Note that the call can perform encryptions in either CBC or ECB mode. Since ECB provides even less security than CBC, we limit our exposition to the harder case of CBC mode.

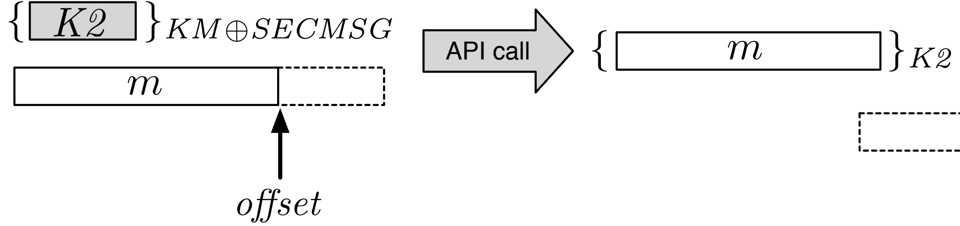

Figure 2: Construction of an Encryption Oracle. The last block is chopped off.

3.2 Injecting Spoofed Secure Messages

Given the encryption oracle above, it's easy to create any chosen export message of length no greater than 4088 bytes, including key material chosen by the adversary. Thus, taking any standard manufacturer *template*, an attacker can manually fill in chosen key or PIN data at the appropriate location, then use the oracle to encrypt this fully-known message under a key of type *SMSG*. If the receiving smart card uses this key as a secure transmission key, the security of these later transmissions is compromised.

3.3 Extracting Secure Messaging Data

While message injection can compromise the operation of particular cards, retrieval of communications keys or PINs without affecting card state is far more dangerous. In this attack we use the templating capability of the API call to expand this oracle into a partial-key dictionary attack mechanism. Using this approach, one can rapidly retrieve the key from any encrypted data field, one byte at a time.

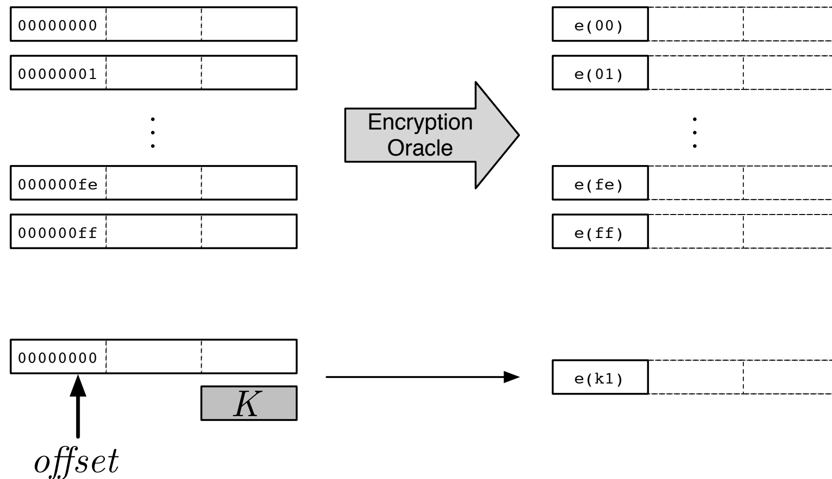

Figure 3: In the key-shifting attack, A 256-element dictionary is built up for each byte of the key that we want to check.

Note first that we use `[ ... ]` to denote hex notation of a single block, in this example 8 bytes (though 16 byte-blocks are just as vulnerable to this attack). The attack uses the EncryptionOracle from section 3.1. It works on any wrapped key K_1 whose type T permits export through this command:

$\mathsf{ExtractKey}\Big(\{\mathsf{K}_1\}_{KM/\mathsf{T}}\Big)$:

1. prepare 256 plaintext blocks of the form [0000 0000 0000 00yy] where $\texttt{00} \leq \texttt{yy} \leq \texttt{ff}$.
2. use EncryptionOracle on all of 256 plaintext blocks to generate a dictionary of 256 ciphertexts indexed by the ciphertext: $\{\forall yy, \texttt{00} \leq yy \leq \texttt{ff} : (c, yy)\}$.
3. given any secure messaging key $\{K_2\}_{KM/SMSG}$, make an API call as follows:

$$\texttt{[0000 0000 0000 0000]},\ \ offset\ = 7, \{K_1\}_{KM/T}, \{K_2\}_{KM/SMSG} \longrightarrow c$$

4. compare c against the dictionary of ciphertext-indexed bytes. The match yields the first byte of the key, call it aa.
5. in order to discover the next byte of the key, repeat the process with a dictionary built from 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, one byte at a time.

For a k-byte key, it takes $257k$ queries to extract the whole key: 256 to build up each dictionary, and one more query to identify the specific key byte. A DES key can be extracted in 2056 queries, while a double-size 3DES key can be extracted in 5112 queries.

4 RG7000 Key Separation Attacks

As described in section 2.3, the RG7000 EMV support is more restrictive, because it does not permit any chosen or known plaintext to be re-encrypted under a secure messaging key. A CBC truncation attack would thus be ineffective, but there is another way to obtain the oracle.

4.1 Construction of an Encryption Oracle

The Secure_Message_With_Integrity function may not offer encryption of arbitrary plaintext, but it is clearly a message authentication oracle – used in mode 0, a MAC can be calculated over any plaintext provided, and the MAC is in essence the final block of a CBC encryption. So our first observation is that the use of 3DES for MAC construction gives an equivalence between a MAC calculated over a single block of input ($template = m_0$), and the CBC mode encryption of the same input block, assuming the same key and initial vector.

$$mac(m_0, K) = \{m_0\}_K \tag{7}$$

Thus, an attacker can exploit the functional similarity between MACs and CBC mode encryption through the use of the Secure_Message_With_Integrity. This provides a single block encryption oracle, and it remains only to show how we can develop this to a multi-block encryption oracle. A logical extension would be to iterate the use of Secure_Message_With_Integrity call, successively calculating the last ciphertext block of longer messages.

$$\begin{aligned} m_0, \{K_1\}_{KM/SECMSGINT} &\longrightarrow mac(m_0, K_1) \\ m_0||m_1, \{K_1\}_{KM/SECMSGINT} &\longrightarrow mac(m_0||m_1, K_1) \end{aligned}$$

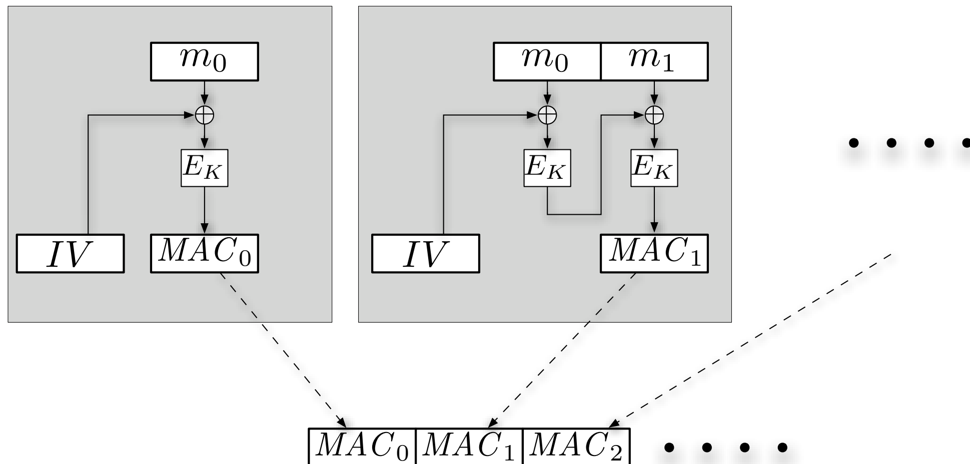

Figure 4: Simplified attack on the RG7000 – An Encryption Oracle from MAC'ing

We would then construct the two-block ciphertext that corresponds to the plaintext message $m_0||m_1$, and repeat until we obtain a n chosen blocks as shown in Figure 4.

$$mac(m_0, K) \; || \; mac(m_0||m_1, K) = \{m_0||m_1\}_K \tag{8}$$

However such an approach does not succeed because the triple DES MAC calculation method specified by VISA for use in EMV is not entirely conventional, and is described below:

Let $\langle K_1, K_2 \rangle$ be the double length 3DES key used in the calculation. The device first calculates a CBC MAC over the plaintext using DES with key K_1 only. The result is then decrypted using K_2 and then re-encrypted using K_1 to achieve 3DES strength. This approach was most likely chosen to reduce the amount of calculation required to verify MACs in smartcards with limited processing power. For a single block plaintext, this MAC calculation remains functionally equivalent to 3DES CBC mode encryption. However, our initial straightforward construction of a multiblock oracle is not consistent with this optimisation of the MAC calculation, so we must use a second approach.

So to create the oracle, we find an alternative solution which exploits a different malleability property of CBC mode to construct longer known ciphertext-plaintext messages: a ciphertext can be composed from two other ciphertext messages with a resultant plaintext value a known function of the two component plaintext messages. Suppose we wish to obtain the encrypted ciphertext corresponding to the chosen plaintext message $m_0||m_1$. We achieve this through two invocations of the `Secure_Message_With_Integrity` call. The first call uses m_0 as plaintext while the second uses the construction $m_0 \oplus mac(m_0, K)$.

$$\begin{aligned} m_0, \{K\}_{KM/SECMSGINT} &\longrightarrow mac(m_0, K) \\ m_1 \oplus mac(m_0, K), \{K\}_{KM/SECMSGINT} &\longrightarrow mac(m_1 \oplus mac(m_0, K), K) \end{aligned}$$

We can now demonstrate the encryption oracle capability by constructing the composed ciphertext as:

$$mac(m_0, K)||mac(m_1 \oplus mac(m_0, K), K) = \{m_0||m_1\}_K \tag{9}$$

The process is shown in Figure 5.

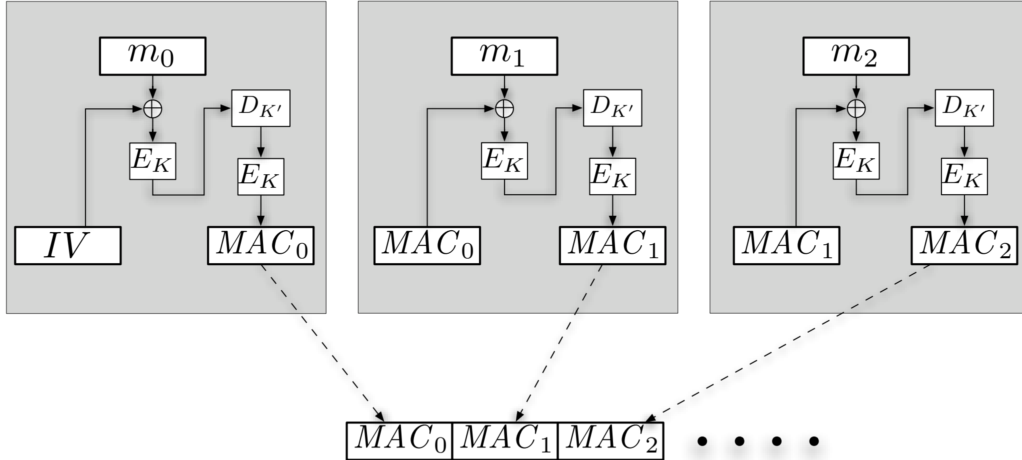

Figure 5: Full attack on the RG7000

4.2 Injecting Spoofed Secure Messages

Given the encryption oracle above, it is straightforward to create a chosen encrypted data field for a secure message, including key material, or a PIN chosen by the adversary. Effectively the attacker takes a standard secure message *template*, but manually fills in the encrypted chosen key at the appropriate location, then uses the API call in MAC-only mode to produce the authentication code over the entire message. The resulting bit stream is identical to the returned template from the `Secure_Message_With_Integrity_And_Confidentiality` call when using the same $SECMSGINT$ key for integrity and confidentiality. If the receiving smart card uses this key as a secure transmission key the security of these later transmissions is compromised.

5 Re-armouring Secure Messaging APIs

In this section, we discuss different approaches to reducing the exposure risks in APIs implementing secure messaging, and consider the design decisions made which may have led to the vulnerabilities.

5.1 Certified Message Templates for the CCA

Updating of HSM firmware is an expensive and hazardous operation to perform at a bank data center, thus API designers try to find the most generic possible commands which can provide a whole range of functionality while maintaining the security policy. The entirely flexible encrypted field template *template* provided by the CCA is unfortunately rather too flexible, and one option is to limit its capability by authenticating particular parameter sets. In essence, *template*, *offset* pairs could be authenticated by some authority. An EMV messaging API would then proceed with the API execution only if the provided message *template*, *offset* , *cert* are correct. Such an approach for authenticating paramters in fact already exists within the CCA through the `Encipher_Cryptographic_Variable` command suite, and it could maybe be extended to authenticate secure command message templates. Thus a modified secure messaging command could operate approximately as follows, with *TMPL* representing the key type for certified template data:

$$\{template, \quad offset \quad \}_{K_3}, \{K_3\}_{KM/TMPL}, \{K_1\}_{KM/T}, \{K_2\}_{KM/SMSG} \longrightarrow \{template[K_1 : \quad offset \quad]\}_{K_2}$$

There are shortcomings of certifying parameters however, firstly if it turns out that the certification process is at least as much hassle as firmware update. Secondly, it still does not address the fundamental issue of the malleability of CBC, though this of course a matter which involves the EMV standardisers. Message truncation by an adversary would remain possible, but with a sufficiently carefully designed set of templates it may be possible to ensure that no truncated template is ever valid.

5.2 Interactions between Encryption and MAC

The Thales RG7000 is more conservative than the CCA as it packages the re-encryption and MAC operations into one atomic API call, and simply offers less functionality in encrypting no chosen plaintext at all. From the perspective of EMV adopters, this approach is maybe a better conceptual one. However this comes at the cost of requiring re-writing of the HSM firmware should the range of EMV secure messages actually used in real life broaden.

In fact, as the Thales attack of section 4.1 shows, the benefit of atomicity was partly undone by permitting command modes that permit decomposition of the command back into separate encryption and MAC stages. The actual functionality of sharing the same key for MAC and Encryption is clearly less secure and this was not unexpected, but the surprise is that the provision for a less secure mode of operation compromises the more secure mode of operation too. Further examples of legacy API support compromising up-to-date functionality can be found in [BA01].

5.3 Selection of Cryptographic Primitives

All the solutions presented so far seek to find API-level solutions to problems which are really related to the nature of the systems' cryptographic primitives. If CBC mode weren't malleable, and if the MAC mechanism didn't closely resemble the encryption operation, our attacks would fail. What these attacks reveal is that the cryptographic primitives built into even modern protocols such as EMV are very fragile, and supporting them through an HSM API is a delicate business.

Trusted Randomness. If an HSM performs CBC encryption with an operator-provided initialization vector, it is clearly a poor method of operating a CBC cipher, since the when the operators is replaced by the adversary, the adversary may control this initialization vector, making it anything but random. However, given that the HSM is a powerful trusted operating device, it may originate trusted randomness, which could add substantial robustness to the protocols. Ideally all sources of randomness used security protocols should be trustworthy and, thus, should originate from within HSMs or smartcards (barring more complicated solutions of verifiable randomness, which are difficult to implement in practice).

Thus, an API call that performs encryption using CBC mode should generate its own IV for new ciphertexts, and only release its value after any adversary-controlled plaintext is already submitted, instead of accepting the IV as an argument. Decryption, of course, still requires an operator-provided IV. Thus, a generic re-wrapping API should accept a single IV, the IV required to decrypt the incoming ciphertext, and produce a single IV, the newly-generated IV that will be necessary to decrypt the output ciphertext.

With trusted randomness, it becomes impossible for an attacker to build a known-plaintext dictionary as done in section 3.3.

Non-Malleability. CBC feedback mode, even with trusted randomness, remains malleable. One reasonably efficient mechanism for non-malleable, symmetric encryption is the Package Transform [?] using OAEP padding [?]. With this approach the plaintext is pre-processed before performing CBC encryption. This pre-processing involves randomizing the plaintext m into a pseudo-plaintext m' such that, given the pseudo-plaintext m', it is easy to compute m, yet with anything less than all of m', nothing can be computed about m. In particular, one block of m' does not yield one block of m, even with unlimited computation power.

Function Separation. MAC computation in most HSM based applications is derived from block cipher computation. This leads to the potentially confusing situation where using the same key for MACing and encrypting opens up all sorts of flaws. Best practice would not have the same cryptographic key used for different purposes, but it is clearly sometimes agreeable for devices like smartcards with limited processing power to re-use functionality. So a balance must be struck between convenience and the need for function separation – and if the MAC and encrypt functions were completely algorithmically independent, then designers would not even conceive that the same key could be shared.

One could consider, for example, using HMAC for MAC'ing, with the assurance that the underlying compression function of the HMAC is not derived from the block function of the encryption scheme. With such a construction based on a strong-enough hash function, MAC and Encrypt will always appear functionally distinct, making each operation more "ideal" and thus less likely to fail for unexpected reasons.

5.4 The Oracle Adversary Model

Security modeling of cryptographic primitives often provides the adversary with oracle access to certain functions and asks whether the adversary can use these queries to break a security feature of the primitive. In a number of such models, the queries are adaptively chosen: the result from earlier queries can be used to determined the input to later queries. The burden of such modeling is fairly high, as the adversary is given significant power. In practice, this model has often been ignored, as it seems unrealistic.

In the case of an HSM or, more generally, a security API, the oracle adversarial model is particularly relevant. The HSM operator may well be the adversary, and the behavior of a security API maps fairly closely to that of an oracle: the operator can issue large numbers of queries, adapt them according to prior output, and use the gleaned information to break a security property of the system.

Thus, in the context of defining security APIs, it may be best practice in the future to use cryptographic primitives that are proven secure against such significantly powerful adversaries.

6 Conclusion

We have discovered and demonstrated a suite of attacks on EMV secure messaging support, as implemented by several HSM vendors. The attacks are significant in their impact on the security of the EMV electronic payment system, and submission of this work was delayed by 12 months to allow vendors to release new versions of their APIs, and to fully explore the consequences.

The attacks exploit familiar pitfalls in the usage of the CBC mode of operation, but the environment where such techniques have been applied is unfamiliar – the world of security APIs is

incredibly fragile to attacks on cryptographic primitives, compared with the world of conventional security protocols which mediate between human parties.

While a large number of potential solutions may patch these particular problems, and some of the failures are a direct result of design compromises, the interesting research questions are to consider longer-term approaches to ensuring API security: one should consider how to correctly trade-off between flexiblity and atomicity, and whether more systematic, formal, approaches to building secure cryptographic APIs can help. In particular, since it appears that the interface of an API closely resembles that Oracle adversary model, cryptographic API designers should seriously consider the use of cryptographic primitives proven secure in appropriate Oracle-adversary conditions.

References

[ABCS04] Ross Anderson, Mike Bond, Jolyon Clulow, and Sergei Skorobogatov. Cryptoprocessors – a survey. *Proceedings of the IEEE – Special Issue on Cryptography and Security Issues*, 2004. to appear.

[AK97] Ross Anderson and Markus Kuhn. Low cost attacks on tamper resistant devices. In *IWSP: International Workshop on Security Protocols, LNCS*, volume 1361, pages 125–136, 1997.

[And94] Ross Anderson. Why cryptosystems fail. *Communications of the ACM*, 37(11):32–40, November 1994.

[And00] Ross J. Anderson. The correctness of crypto transaction sets. In Bruce Christianson, Bruno Crispo, and Michael Roe, editors, *Security Protocols Workshop*, volume 2133 of *Lecture Notes in Computer Science*, pages 125–127. Springer, 2000.

[BA01] Mike Bond and Ross Anderson. API-level attacks on embedded systems. *IEEE Computer Magazine*, 34(10):67–75, October 2001.

[BC04] Mike Bond and Jolyon Clulow. Encrypted? randomised? compromised? (when cryptographically secured data is not secure). In Ed Dawson and Wolfgang Klemm, editors, *Cryptographic Algorithms and their Uses*, pages 140–151. Queensland University of Technology, 2004.

[BC05] Mike Bond and Jolyon Clulow. Extending security protocol analysis: New challenges. *Electr. Notes Theor. Comput. Sci.*, 125(1):13–24, 2005.

[Bon01] Mike Bond. Attacks on cryptoprocessor transaction sets. Presented at the CHES 2001 Workshop in Paris, January 2001.

[Bon04] Mike Bond. *Understanding Security APIs.* PhD thesis, University of Cambridge, 2004.

[Clu03] Jolyon S. Clulow. The design and analysis of cryptographic application programming interfaces for devices. Master's thesis, University of Natal, Durban, 2003.

[EMV04] EMVCo. *EMV Integrated Circuit Card Specifications for Payment Systems*, May. 20004.

[Int03] International Business Machines Corporation. *IBM PCI Cryptographic Coprocessor: CCA Basic Services Reference and Guide Release 2.41, Revised September 2003 for IBM 4758 Models 002 and 023*, Sep. 2003.

[Zax00] Zaxus (now owned by Thales). *Host Security Module RG7000 Programmers Manual 1270A514 Issue 3*, May. 2000.