APDU-level attacks in PKCS#11 devices

Claudio Bozzato[1], Riccardo Focardi[1,2], Francesco Palmarini[1], and Graham Steel[2]

[1]Ca' Foscari University, Venice
cbozzato@dsi.unive.it, focardi@unive.it, palmarini@unive.it
[2] Cryptosense, Paris
graham@cryptosense.com

Abstract

In this paper we describe attacks on PKCS#11 devices that we successfully mounted by interacting with the low-level APDU protocol, used to communicate with the device. They exploit proprietary implementation weaknesses which allow attackers to bypass the security enforced at the PKCS#11 level. Some of the attacks leak, as cleartext, sensitive cryptographic keys in devices that were previously considered secure. We present a new threat model for the PKCS#11 middleware and we discuss the new attacks with respect to various attackers and application configurations. All the attacks presented in this paper have been timely reported to manufacturers following a *responsible disclosure* process.

1 Introduction

Cryptographic hardware such as USB tokens, smartcards and Hardware Security Modules has become a standard component of any system that uses cryptography for critical activities. It allows cryptographic operations to be performed inside a protected, tamper-resistant environment, without the need for the application to access the (sensitive) cryptographic keys. In this way, if an application is compromised the cryptographic keys are not leaked, since their value is stored securely in the device.

Cryptographic hardware is accessed via a dedicated API. PKCS#11 defines the RSA standard interface for cryptographic tokens and is now administered by the Oasis PKCS11 Technical Committee [14, 15]. In PKCS#11, fresh keys are directly generated inside devices and can be shared with other devices through special key *wrapping* and *unwrapping* operations, that allow for exporting and importing keys encrypted under other keys. For example, a fresh symmetric key k can be encrypted (wrapped) by device d_1 under the public key of device d_2 and then exported out of d_1 and imported (unwrapped) inside d_2 that will perform, internally, the decryption. In this way, key k will never appear as cleartext out of the devices. One of the fundamental properties of PKCS#11 is, in fact, that keys marked as *sensitive* should never appear out of a device unencrypted.

In the last 15 years, several API-level attacks on cryptographic keys have appeared in literature [1, 3, 4, 5, 6, 9, 12]. As pioneered by Clulow [6], the attributes of a PKCS#11 key might be set so to give the key conflicting roles, contradicting the standard *key separation* principle in cryptography. For example, to determine the value of a sensitive key k_1 given a second key k_2, an attacker simply wraps k_1 using k_2 and decrypts the resulting ciphertext using k_2 once again. The fact that a key should never be used to perform both the wrapping of other keys and the decryption of arbitrary data (including wrapped keys) is not explicitly stated in the specification of PKCS#11 and many commercial devices have been recently found vulnerable to Clulow's attack [5].

In this paper, we describe new, unpublished attacks that work at a different API-level. The PKCS#11 API is typically implemented in the form of a middleware which translates the high-level PKCS#11 commands into low-level ISO 7816 Application Protocol Data Units (APDUs) and exposes results of commands in the

expected PKCS#11 format. In our experiments, we noticed that this translation is far from being a 1-to-1 mapping. Devices usually implement simple building blocks for key storage and cryptographic operations, but most of the logic and, in some cases, some of the sensitive operations are delegated to the middleware.

We have investigated how five commercially available devices implement various security-critical PKCS#11 operations, by analyzing in detail the APDU traces. Our findings show that APDU-level attacks are possible and that four out of the five analyzed devices leak symmetric sensitive keys in the clear, out of the device. We also show that, under a reasonable attacker model, the authentication phase can be broken, allowing for full access to cryptographic operations. Interestingly, we found that most of the logic of PKCS#11 is implemented at the library level. Key attributes that regulate the usage of keys do not have any importance at the APDU-level and can be easily bypassed. For example, we succeeded performing signatures under keys that do not have this functionality enabled at the PKCS#11 level. For one device, we also found that RSA session keys are managed directly by the library in the clear violating, once more, the PKCS#11 basic requirement that sensitive keys should never leave the token unencrypted.

The focus of this paper is primarily on USB tokens and smartcards, so our threat model refers to a typical single-user desktop/laptop configuration. In particular, we consider various application configurations in which the PKCS#11 layer and the authentication phase are run at different privileges with respect to the user application. Protecting the PKCS#11 middleware turns out to be the only effective way to prevent the APDU-level attacks that we discovered, assuming that the attacker does not have physical access to the token. In fact, physical access would enable USB sniffing, revealing any key communicated in the clear from/to the token. Separating authentication (for example using a dialog running at a different privilege) offers additional protection and makes it hard to use the device arbitrarily through the PKCS#11 API. However, an attacker might still attach to the process and mount a Man-In-The-Middle attack at the PKCS#11 layer, injecting or altering PKCS#11 calls.

Contributions. In this paper we present: (*i*) a new threat model for PKCS#11 middleware; (*ii*) new, unpublished APDU-level attacks on commercially available tokens and smartcards, some of which were considered secure; (*iii*) a security analysis of the vulnerabilities with respect to the threat model.

Related work. Many API-level attacks have been published in the last 15 years. The first one is due to Longley and Rigby [12] on a device that was later revealed to be a Hardware Security Module manufactured by Eracom and used in the cash machine network. In 2000, Anderson published an attack on key loading procedures on another similar module manufactured by Visa [1] and presented more attacks in two subsequent papers [3, 4]. Clulow published the first attacks on PKCS#11 in [6]. All of these attacks had been found manually or through ad-hoc techniques. A first effort to apply general analysis tools appeared in [20], but the researchers were unable to discover any new attacks and could not conclude anything about the security of the device. The first automated analysis of PKCS#11 with a formal statement of the underlying assumptions was presented in [9]. When no attacks were found, the authors were able to derive precise security properties of the device. In [5], the model was generalized and provided with a reverse-engineering tool that automatically refined the model depending on the actual behaviour of the device. When new attacks were found, they were tested directly on the device to get rid of possible spurious attacks determined by the model abstraction. The automated tool of [5] successfully found attacks that leak the value of sensitive keys on real devices.

Low-level smartcard attacks have been studied before but no previous APDU-level attacks and threat models for PKCS#11 devices have been published in literature. In [2], the authors showed how to compromise the APDU buffer in Java Cards through a combined attack that exploits both hardware and software vulnerabilities. In [8], the authors presented a tool that gives control over the smart card communication channel for eavesdropping and man-in-the-middle attacks. In [13], the authors illustrated how a man-in-the-middle attack can enable payments without knowing the card PIN.

In [10] a subset of the authors investigated an automated method to systematically reverse-engineer the mapping between the PKCS#11 and the APDU layers. The idea is to provide abstract models in first-order logic of low level communication, on-card operations and possible implementations of PKCS#11 functions. The abstract models are then refined based on the actual APDU trace, in order to suggest the actual mapping

between PKCS#11 commands and APDU traces. The two papers complement each other: the present one illustrates real attacks with a threat model and a security analysis, while [10] focuses on automating the manual, non-trivial reverse engineering task. All of the attacks presented here have been found manually and some of them have been used as test cases for the automated tool of [10].

Finally, for what concerns the threat model, in the literature we find a number of general methodologies (e.g., [17, 18, 19]) that do not directly apply to our setting. In [7] the authors discussed threat modelling for security tokens in the setting of web application while [16] described in details all the actors and threats for smart cards, but none of these papers considered threats at the PKCS#11 middleware layer. To the best of our knowledge, the threat model we propose in this work is the first one in the setting of PKCS#11 tokens and smartcards which takes into account the APDU layer as an attack entry point.

Structure of the paper. The paper is organized as follows: in section 2 we give background about the PKCS#11 and APDU layers; in section 3 we present the threat model; in section 4 we illustrate in detail our findings on five commercially available devices; in section 5 we analyze the attacks with respect to the threat model and in section 6 we draw some concluding remarks.

2 Background

PKCS#11 defines the RSA standard interface for cryptographic tokens and is now developed by the Oasis PKCS11 Technical Committee [14, 15].

The PKCS#11 API is typically implemented in the form of a middleware which translates the high-level PKCS#11 commands into low-level ISO 7816 Application Protocol Data Units (APDUs) and exposes results of commands in the expected PKCS#11 format. Thus, from an architectural perspective, the PKCS#11 middleware can be seen as the combination of two layers: the PKCS#11 API and the device API. All of the devices we have analyzed are based on the PC/SC specification for what concerns the low-level device API.[1] This layer is the one that makes devices operable from applications and allows for communication with the device reader, exposing both standard and proprietary commands, formatted as ISO 7816 APDUs. In the following, we will refer to this layer as the APDU layer.

The PKCS#11 and the APDU layer are usually implemented as separate libraries. As an example, in Windows systems PC/SC is implemented in the winscard.dll library. Then, a separate, device-specific PKCS#11 library links to winscard.dll in order to communicate with the device.

It is important to notice that, even if PC/SC provides a standard layer for low-level communication, different devices implement the PKCS#11 API in various, substantially different ways. As a consequence, each device requires its specific PKCS#11 library on top of the PC/SC one. Figure 1 gives an overview of a typical PKCS#11 middleware architecture with two cards requiring two different PKCS#11 libraries which communicates with the cards using the same PC/SC library.

In subsection 2.1 and subsection 2.2 we illustrate the PKCS#11 and the APDU layers more in detail. Readers familiar with these layers can safely skip the following sections.

2.1 The PKCS#11 layer

As well as providing access to cryptographic functions – such as encryption, decryption, sign and authentication – PKCS#11 is designed to to provide a high degree of protection of cryptographic keys. Importing, exporting, creating and deleting keys stored in the token should always be performed in a secure way. In particular, the standard requires that even if the token is connected to an untrusted machine, in which the operating system, device drivers or software might be compromised, keys marked as sensitive should never be exported as cleartext out of the device.

In order to access the token, an application must authenticate by supplying a PIN and initiate a session. Notice, however, that if the token is connected to an untrusted machine the PIN can be easily intercepted,

[1] http://www.pcscworkgroup.com/

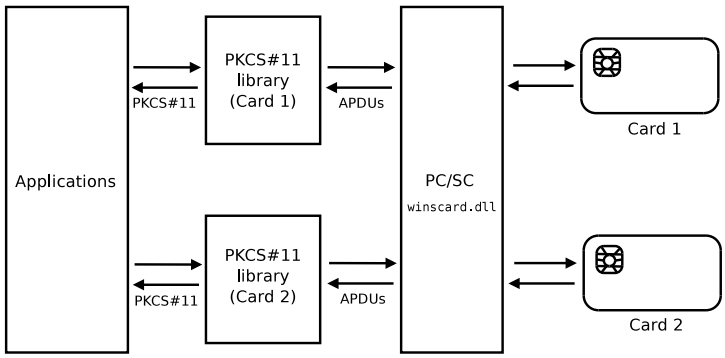

Figure 1: PKCS#11 middleware for two PC/SC (winscard.dll) cards with different PKCS#11 libraries.

e.g., through a keylogger. Thus, the PIN should only be considered as a second layer of protection and it should not be possible to export sensitive keys in the clear even for legitimate users, that know the PIN (cf. [15], section 7).

PKCS#11 defines a number of *attributes* for keys that regulate the way keys should be used. We briefly summarize the most relevant ones from a security perspective (see [14, 15] for more detail):

CKA_SENSITIVE the key cannot be revealed as plaintext out of the token. It should be impossible to unset this attribute once it has been set, to avoid trivial attacks;

CKA_EXTRACTABLE the key can be wrapped, i.e. encrypted, under other keys and extracted from the token as ciphertext; unextractable keys cannot be revelead out of the token even when encrypted. Similarly to CKA_SENSITIVE, it should not be possible to mark a key as extractable once it has been marked unextractable;

CKA_ENCRYPT, CKA_DECRYPT the key can be used to encrypt and decrypt arbitrary data;

CKA_WRAP, CKA_UNWRAP the key can be used to encrypt (wrap) and decrypt (unwrap) other CKA_EXTRACTABLE, possibly CKA_SENSITIVE keys. These two operations are used to export and import keys from and into the device;

CKA_SIGN, CKA_VERIFY the key can be used to sign and verify arbitrary data;

CKA_PRIVATE the key can be accessed even if the user is not authenticated to the token when it is set to false;

CKA_TOKEN the key is not stored permanently on the device (discarded at the end of the session) when it is set to false.

Example 1 (PKCS#11 symmetric key encryption) *Listing 1 reports a fragment of C code performing symmetric* `DES/CBC` *encryption of plaintext* `"AAAAAAAA"` *with initialization vector* `0x0102030405060708`*.* `PKCS#11` *session has already been initiated and* `session` *contains a handle to the active session. We also assume that* `DESkey` *is a valid handle to a DES encryption key.*

We can see that `C_EncryptInit` *initializes the encryption operation by instantiating the* `DES/CBC` *cryptographic* `mechanism` *and the cryptographic key* `DESkey`*. Then,* `C_Encrypt` *performs the encryption of the string* `plaintext` *and stores the result and its length respectively in* `ciphertext` *and* `ciphertext_len`*. In order to keep the example simple, we skipped checks for errors that should be performed after every* `PKCS#11` *API call (cf. [15], section 11). In subsection 2.2 we will show how this example is mapped in* `APDUs` *on one token.*

```
0  /* Session initialization and loading of DESkey has been omitted ... */
1
2  CK_BYTE_PTR plaintext = "AAAAAAAA";                    /* plaintext */
3  CK_BYTE iv[8] = {1, 2, 3, 4, 5, 6, 7, 8};              /* initialization vector */
4  CK_BYTE ciphertext[8];                                 /* ciphertext output buffer */
5  CK_ULONG ciphertext_len;                               /* ciphertext length */
6  CK_MECHANISM mechanism = {CKM_DES_CBC, iv, sizeof(iv)}; /* DES CBC mode with given iv */
7
8  /* Initialize the encryption operation with mechanism and DESkey */
9  C_EncryptInit(session, &mechanism, DESkey);
10
11 /* Encryption of the plaintext string into ciphertext buffer */
12 C_Encrypt(session, plaintext, strlen(plaintext), ciphertext, &ciphertext_len);
```

Listing 1: PKCS#11 DES/CBC encryption under key DESkey.

2.2 The APDU layer

The ISO/IEC 7816 is a standard for identification, integrated circuit cards. Organization, security and commands for interchange are defined in part 4 of the standard [11]. The communication format between a smartcard and an off-card application is defined in terms of Application Protocol Data Units (APDUs). In particular, the half-duplex communication model is composed of APDU pairs: the reader sends a Command APDU (C-APDU) to the card which replies with a Response APDU (R-APDU). The standard contains a list of *inter-industry commands* whose behaviour is specified and standardized. Manufacturers can integrate these standard commands with their own *proprietary commands*.

A C-APDU is composed of a mandatory 4-byte header (CLA,INS,P1,P2), and an optional payload (Lc,data,Le), described below:

CLA one byte referring to the instruction class which specifies the degree of compliance to ISO/IEC 7816 and whether the command and the response are inter-industry or proprietary. Typical values are 0x00 and 0x80, respectively for inter-industry and proprietary commands;

INS one byte representing the actual command to be executed, *e.g.* READ RECORD;

P1,P2 two bytes containing the instruction parameters for the command, *e.g.* the record number/identifier;

Lc one or three bytes, depending on card capabilities, containing the length of the optional subsequent data field;

data the actual Lc bytes of data sent to the card;

Le one or three bytes, depending on card capabilities, containing the length (possibly zero) of the expected response.

The R-APDU is composed of an optional Le bytes data payload (absent when Le is 0), and a mandatory 2-bytes status word (SW1,SW2). The latter is the return status code after command execution (*e.g.* FILE NOT FOUND).

Example 2 (Symmetric key encryption in APDUs) *We show how the PKCS#11 code of 1 is mapped into APDUs on the Athena ASEKey USB token. Notice that this token performs a challenge-response authentication before any privileged command is executed. For simplicity, we omit the authentication part in this example but will discuss it in detail in section 4.1.*

The encryption operation begins by selecting the encryption key from the right location in the token memory: at line 3, the token selects the directory (called Dedicated File in ISO-7816) and, at line 6, the file (Elementary File) containing the key. At line 9, the encryption is performed: the Initialization Vector and the plaintext are sent to the token which replies with the corresponding ciphertext.

We describe in detail the the APDU format specification of SELECT FILE command at line 3:

CLA *value 0x00 indicates that the command is ISO-7816 inter-industry;*

```
0  # The challenge-response authentication is omitted. For details see section 4.1
1
2  # ISO-7816 SELECT FILE command to select the folder (DF) where the key is stored
3  APDU: 00 a4 04 0c 00 00 06 50 55 42 4c 49 43
4  SW: 90 00
5  # ISO-7816 SELECT FILE command to select the file (EF) containing the encryption key
6  APDU: 00 a4 02 0c 00 00 02 83 01
7  SW: 90 00
8  # Encryption of the plaintext (red/italic) using the selected key and the given IV (green/overlined). The ciphertext is
       returned by the token (blue/underlined).
9  APDU: 80 16 00 01 00 00 10 01 02 03 04 05 06 07 08 41 41 41 41 41 41 41 41 00 00
10 SW: d2 ef a5 06 92 64 44 13 90 00
```

Listing 2: APDU session trace of the PKCS#11 symmetric key encryption.

INS *value 0xA4 corresponds to inter-industry SELECT FILE (cf. [11], section 6);*

P1 *value 0x04 codes a direct selection of a Dedicated File by name;*

P2 *value 0x0C selects the first record, returning no additional information about the file;*

Lc *the tokens is operating in extended APDU mode, thus this field is 3 bytes long. Value 0x000006 indicates the length 6 of the subsequent field;*

data *contains the actual ASCII-encoded name ("PUBLIC") of the DF to be selected;*

SW1,SW2 *the status word 0x90 0x00 returned by the token indicates that the command was successfully executed.*

It is important to notice that the C_EncryptInit function call sends no APDU to the token: we can infer that the low level protocol of the encryption operation is stateless and the state is managed inside the PKCS#11 library. This example shows that the mapping between the PKCS#11 layer and the APDU layer is not 1-to-1 and the PKCS#11 library is in some cases delegated to implement critical operations, such as maintaining the state of encryption. We will see how this leads to attacks in section 4.

3 Threat model

In this section we analyze various threat scenarios and classify them based on the attacker capabilities.

We consider a typical scenario in which the target token is connected to a desktop or laptop computer running in a single-user configuration. We describe the threat model by focusing on the following sensitive targets:

PIN If the attacker discovers the PIN he might be able to perform cryptographic operations with the device when it is connected to the user's host or in case he has physical access to it;

Cryptographic operations The attacker might try to perform cryptographic operations with the token, independently of his knowledge of the PIN;

Cryptographic keys The attacker might try to learn sensitive keys either by exploiting PKCS#11 API-level attacks such as Clulow's wrap-and-decrypt [6] (cf. subsection 2.1) or by exploiting the new APDU-level vulnerabilities we will discuss in section 4.

3.1 Administrator privileges

If the attacker has administration privileges, he basically has complete control of the host. He can modify the driver, replace the libraries, intercept any input for the users and attach to any running process[2]. As

[2]This is typically done by using the operating system debug API to instrument or inspect the target process memory. Examples are the Event Tracing API for Windows and the Linux ptrace() syscall.

such, he can easily learn the PIN when it is typed or when it is sent to the library, use the PIN to perform any cryptographic operations and exploit any PKCS#11 or APDU level attacks to extract cryptographic keys in the clear.

3.2 User privileges

The most common situation is when the attacker has user privileges. In this case we have different scenarios:

Monolithic. The application is run by the same user as the attacker and directly links both the PKCS#11 and the APDU library. The attacker can easily sniff and alter data by attaching to the application process and by intercepting library calls. The attacker can easily learn the PIN when it is sent to the library, use the PIN to perform any cryptographic operations and exploit any PKCS#11 or APDU level attacks to extract cryptographic keys in the clear.

Separate authentication mechanism. The application is run by the same user as the attacker and directly links the PKCS#11 library but authentication is managed by a separate software or hardware which is not directly accessible with user privileges. Examples could be a separate dialog for entering the PIN running at different privileges or some biometric sensor integrated in a USB token. The attacker cannot directly log into the token but can still sniff and alter data by attaching to the application process and by intercepting library calls. If the attacker is able to place in the middle and alter data, he could additionally exploit PKCS#11 or APDU-level attacks to extract cryptographic keys in the clear. Notice that, knowing the PIN, this can be done by simply opening a new independent session. Without knowledge of the PIN, instead, the attacker needs a reliable Man-In-The-Middle (MITM) attack.

Separate privileges. If the middleware layer is run as separate process at a different privilege level, the attacker cannot attach to it and observe or alter APDUs. The attacker can still try to access the token directly, so if there are ways to bypass authentication he might be able to perform cryptographic operations and exploit PKCS#11 or APDU-level attacks.

3.3 Physical access

If the attacker has physical access to the user host he might install physical key-loggers and USB sniffers. This is not always feasible for example if the devices are integrated, as in laptops. In the case of a key-logger, the attacker can easily discover the PIN if it is typed through the keyboard. The case of directly sniffing APDUs passing, e.g., through USB, is interesting and more variegated since different sensitive data could be transmitted through the APDU layer, as we will illustrate in section 4.

3.4 Summary of the threat model

Table 1 summarizes what the various attackers can access and exploit in different settings. We distinguish between passive APDU attacks, where the attacker just sniffs the APDU trace, and active APDU attacks, where APDUs are injected or altered by the attacker. In some cases active APDU attacks require mounting a MITM, e.g., when the PIN is now known or when the attacker does not have access to the middleware, as in physical attacks.

We point out that, if the application is monolithic, an attacker with user privileges is as powerful as one with administrative privileges. The maximum degree of protection is when the application offers separate authentication and the middleware runs with different privileges. We notice that the attacker can still perform PKCS#11-level attacks without knowing the PIN by mounting a MITM and altering or hijacking the API calls. Finally, physical attacker can in principle perform all the attacks, except the ones that are based on inspecting process (or middleware) memory and assuming, in some cases, MITM capabilities.

Attacker	Application	Attacker can access		Attacker can exploit			
		PKCS#11	APDU	PIN	PKCS#11	APDU passive	APDU active
Admin	Any	✓	✓	✓	✓	✓	✓
User	Monolithic	✓	✓	✓	✓	✓	✓
	Sep. Auth.	✓	✓	✗	✓[1]	✓	✓[1]
	Sep. Privileges	✓	✗	✓	✓	✗	✗
	Sep. Auth.&Priv.	✓	✗	✗	✓[1]	✗	✗
Physical	Any	✗	✓	✓[2]	✓[1]	✓[3]	✓[1,3]

[1] Requires MITM.
[2] Through a keylogger or a USB sniffer.
[3] Only APDU payloads, cannot access middleware memory.

Table 1: Threats versus attackers and applications.

4 APDU-level attacks on real devices

We have tested the following five devices from different manufacturers for possible APDU-level vulnerabilities.

- Aladdin eToken PRO (USB)
- Athena ASEKey (USB)
- RSA SecurID 800 (USB)
- Safesite Classic TPC IS V1 (smartcard)
- Siemens CardOS V4.3b (smartcard)

For readability, in the following we will refer to the above tokens and smartcards as eToken PRO, ASEKey, SecurID, Safesite Classic and Siemens CardOS, respectively. These five devices are the ones tested in [5] for which we could find APDU-level attacks. It is worth noticing that we could not inspect the APDU traces of some other devices analyzed in [5] because they encrypt the APDU-level communication. We leave the study of the security of encrypted APDUs as a future work.

We have systematically performed various tests on selected sensitive operations and we have observed the corresponding APDU activity. We have found possible vulnerabilities concerning the login phase (subsection 4.1), symmetric sensitive keys (subsection 4.2), key attributes (subsection 4.3), private RSA session keys (subsection 4.4).

Quite surprisingly, we have found that, in some cases, cryptographic keys appear as cleartext in the library which performs cryptographic operations in software. Moreover, we have verified that the logic behind PKCS#11 key attributes is, in most of the cases, implemented in the library. We have also found that all devices are vulnerable to attacks that leak the PIN if the middleware is not property isolated and run with a different privilege (which is usually not the case). Moreover, attackers with physical access could sniff an authentication session through the USB port and brute-force the PIN once the authentication protocol has been reverse-engineered.

Our findings have been timely reported to manufacturers following a *responsible disclosure* process and are described in detail in the following subsections. Official answers from manufacturers, if any, will be made available at https://secgroup.dais.unive.it/projects/apduattacks/.

C_Login session trace	Device
0 # Custom Get challenge: 1 APDU: 80 17 00 00 08 2 SW: *df 89 61 34 62 05 13 36* 90 00 3 # Custom External authenticate: 4 APDU: 80 11 00 11 0a 10 08 *64 d5 97 15 4a 44 eb 23* 5 SW: 90 00	Aladdin eToken PRO
0 # Standard ISO-7816 Get challenge: 1 APDU: 00 84 00 00 00 00 08 2 SW: *bb 8b ec f8 a3 a8 62 63* 90 00 3 # Standard ISO-7816 External authenticate: 4 APDU: 00 82 02 00 00 00 18 00 00 11 12 *8f e3 fa a6 a8 a8 07 10 47 e0 af 90 65 20 42 43 2d f0 47 16* 5 SW: 90 00	Athena ASEKey USB
0 # Send 8 random bytes: 1 APDU: 80 50 81 01 08 *c9 ff 3c d6 63 a2 13 b0* 2 SW: 61 1c 3 # Standard ISO-7816 Get response: 4 APDU: 00 c0 00 00 1c 5 SW: 35 34 95 09 14 02 1d 3a 03 2a 81 01 03 *2a ec a5 97 cc d0 ea 8a cb 05 59 94 78 e1 04* 90 00 6 # Custom External authenticate: 7 APDU: 84 82 03 00 10 *fb bb dd 65 5f 0d 70 cc 41 a7 23 47 1d af b0 72* 8 SW: 90 00	RSA SecurID 800
0 # Standard ISO-7816 Select file: 1 APDU: 00 a4 04 00 0c a0 00 00 00 18 0a 00 00 01 63 42 00 2 SW: 90 00 3 # Standard ISO-7816 Verify: 4 APDU: 00 20 00 01 08 *31 32 33 34* 00 00 00 00 5 SW: 90 00	Safesite Classic TPC IS V1
0 # Standard ISO-7816 Select file: 1 APDU: 00 a4 04 0c 0c a0 00 00 00 63 50 4b 43 53 2d 31 35 2 SW: 90 00 3 # Standard ISO-7816 Verify: 4 APDU: 00 20 00 81 05 *31 32 33 34 35* 5 SW: 90 00	Siemens CardOS V4.3b

Table 2: APDU session trace of the PKCS#11 C_Login function for the five devices.

4.1 Authentication

In PKCS#11 the function C_Login allows a user to authenticate, in order to activate a session and perform cryptographic operations. For the five devices examined, we found that authentication is implemented in two different forms: plaintext and challenge-response.

Plain authentication. This authentication method is used by Safesite Classic and Siemens CardOS. When the function C_Login is called, the PIN is sent as plaintext to the token to authenticate the session. This operation does not return any session handle at the APDU level, meaning that the low level protocol is stateless: a new login is transparently performed by the library before any privileged command is executed. The fact the PIN is sent as plaintext allows to easily sniff the PIN even without having control of the computer, for example using a hardware USB sniffer.

In Table 2 we report an excerpt of a real APDU session trace of the C_Login function. We can see that Safesite Classic and Siemens CardOS tokens use (line 4) the standard ISO-7816 VERIFY command to authenticate: the PIN, in red color/italic, is sent as a ASCII encoded string ("1234" and "12345", respectively).

Challenge-Response authentication. In the eToken PRO, ASEKey and SecurID tokens the function C_Login executes a challenge-response protocol to authenticate the session: the middleware generates a response based on the challenge provided by the token and the PIN given by the user. At the APDU level, eToken PRO and ASEKey do not return any session handle thus, as for the previous devices, the low level protocol is stateless and a new login is transparently performed by the library before executing any privileged

APDU session trace	Token
0 # DES Key generation: red/italic = plain key value sent to the token 1 APDU: 80 16 01 00 2b 01 01 02 02 02 40 01 03 02 00 18 04 04 11 11 11 11 10 18 17 3f ff ff ff ff 01 08 *3f 44 5f c4 eb 76 f1 86* 06 64 65 73 6b 65 79 00 2 SW: 90 00	`C_GenerateKey` sample on Aladdin eToken PRO
0 # Fetch the key: green/overlined = attributes, red/italic = plain key value, blue/underlined = label 1 APDU: 80 18 00 00 04 0e 02 00 00 18 2 SW: *17 3f ff ff ff ff 01* 08 *3f 44 5f c4 eb 76 f1 86* 06 *64 65 73 6b 65 79 00* 90 00	`C_WrapKey` sample on Aladdin eToken PRO
0 # 3DES Secret key generation 1 APDU: 80 16 00 00 1a 72 35 *be 4e aa de 2d 47 72 b2 8b 47 5f de 63 4d 7e 30 a5 f0 ac 5f c0 56 c6 90* 2 SW: 90 00	`C_GenerateKey` sample on RSA SecurID 800
0 # 3DES key is read in the clear even if CKA_SENSITIVE is set to true 1 APDU: 00 c0 00 00 18 2 SW: *36 90 fa c9 4e 82 55 b1 71 1d 81 e4 3c d1 bd fa 44 9c bb c3 b1 8b 1e 8d* 90 00	`C_GetAttributeValue` sample on RSA SecurID 800
0 # Get challenge (Standard ISO-7816): 1 APDU: 00 84 00 00 00 00 08 2 SW: *b7 c8 14 4b 4e 5f e6 3e* 90 00 3 # External authenticate (Standard ISO-7816): 4 APDU: 00 82 02 00 00 00 18 00 00 11 12 95 fa da de 0d 70 42 d9 21 c2 27 a4 8b af 7a 8b 90 47 ae 54 5 SW: 90 00 6 # Get an RSA modulus (in red/italic) 7 SW: *79 23 57 33 9a be 2a dd ba ae 2e 09 4c d0 3d 57 8b d0 07 e4 cb ... (omitted) ... 19 6d 15 ea b6 aa cc 2b e8 30 c3 e8 cf* 90 00 8 # Send the encrypted key to the token 9 APDU: 80 24 00 80 00 00 a0 *20 5b f1 f9 cd 67 c8 3d e0 cf 9b 1b c7 ad ... (omitted) ... 33 0b 85 1a 27 7e cd 69 95 71 ca 2e 88 33 a7 f6 4a 97 22 a0* 10 SW: 90 00	`C_GenerateKey` sample on Athena ASEKey

Table 3: Leakage of sensitive symmetric keys during `PKCS#11` operations.

command. Instead, on the SecurID the challenge-response routine is executed only once for each session as it returns a session handle.

`PKCS#11` standard allows PIN values to contain any valid UTF8 character, but the token may impose restrictions. Assuming that the PIN is numeric and short (4-6 digits), which is the most common scenario, an attacker is able to bruteforce the PIN offline, *i.e.* without having access to the device, as it is enough to have one `APDU` session trace containing one challenge and one response. As a proof of concept, we have reverse engineered the authentication protocol of eToken PRO and ASEKey implemented in the `PKCS#11` library. This allowed us to try all possible PINs and check whether or not the response computed from the challenge and the PIN matches the one in the trace.

In Table 2 we can see that eToken PRO makes use of proprietary commands to request the challenge and provide the response, while ASEKey uses the standard ISO-7816 `GET CHALLENGE` and `EXTERNAL AUTHENTICATE` commands. We have not reverse engineered the challenge-response protocol of the SecurID token but, looking at the `APDU` session trace, we can identify a three-steps authentication protocol. At line 1 eight random bytes are sent to the token; then, a standard ISO-7816 `GET RESPONSE` command is issued to retrieve the challenge (highlighted in red and italic at line 5) and the identifier of the `PKCS#11` session (highlighted in green and overlined). Line 7 contains the response generated by the middleware.

On both plain and challenge-response authentication, we have found that tokens implement no protection against MITM: if an attacker can place himself in the middle of the connection he could exploit an authentication exchange to alter user commands or inject his own ones.

4.2 Sensitive symmetric keys

We discovered that in Siemens CardOS, eToken PRO and SecurID encryption and decryption under a sensitive symmetric key is performed entirely by the middleware. As a consequence, the value of the sensitive key is sent out of the token as plaintext. This violates the basic `PKCS#11` property stating that sensitive keys should never be exported in the clear. We also found that ASEKey surprisingly reuses the authentication challenge (sent in the clear) as the value of freshly generated DES keys.

In the following, we describe the four devices separately.

Siemens CardOS V4.3b. This smartcard does not allow to create symmetric keys with `CKA_TOKEN` set to true, meaning that symmetric keys will always be session keys. According to `PKCS#11` documentation, session keys are keys that are not stored permanently in the device: once the session is closed these keys are destroyed. Notice that this does not mean that sensitive session keys should be exported in the clear out of the token. What distinguishes a session key from a token key is *persistence*: the former will be destroyed when the session is closed while the latter will persist in the token.

We observed that encryption under a sensitive key sends no `APDUs` to the token. This gives evidence that encryption takes place entirely in the middleware. Moreover, we verified that even `C_GenerateKey` function does not send any `APDU`: in fact, it just calls the library function `pkcs11_CryptGenerateRandom` to generate a random key value whose value is stored (and used) only inside the library.

Aladdin eToken PRO. In Table 3 (first row), we show that symmetric key generation in eToken PRO is performed by the middleware. We can see, in red and italic, a DES key value sent to the token in the clear.

The value of symmetric keys stored in the eToken PRO can be read by using the proprietary `APDU` command `0x18`. No matter which attributes are set for the key, its value can be read. We tested it over a DES key with attributes `CKA_TOKEN`, `CKA_PRIVATE`, `CKA_SENSITIVE` set to true. In order to perform this attack a valid login is required. Since symmetric key operations are performed by the library, this `APDU` command is used to retrieve the key from the token before performing operations in software.

As an example, in Table 3 (second row) we see part of a `C_WrapKey` operation that retrieves a the DES cryptographic key from the token. We can see the value of the key in the clear.

RSA SecurID 800. In Table 3 (third row), we show that symmetric key generation in SecurID is also performed by the middleware. We can see, in red and italic, a 3DES key value sent to the token in the clear.

We were also able to retrieve the value of a sensitive key stored inside the SecurID by just issuing the correct `APDU` command. In fact, when trying to use the `C_GetAttributeValue` function, the library correctly returns the `CKR_ATTRIBUTE_SENSITIVE` error. However, what really happens is that the key is read from the token but the library just avoids to return it. In Table 3 (fourth row) we can see (in red and italic) the value of the sensitive key leaked by the token.

Athena ASEKey. The most surprising behaviour is shown by the ASEKey: the value of token sensitive symmetric keys cannot be read arbitrarily via `APDU` commands, as they are stored in a separated Dedicated File (DF) which requires authentication. Nonetheless the key value is unnecessarily leaked when the key is generated.

In Table 3 (fifth row) we report an excerpt of `APDU` session for the `C_GenerateKey` function. We notice that `C_GenerateKey` sends (line 9) the key encrypted under RSA with a modulus (line 7), using the public exponent `0x010001`. In fact, the library encrypts the whole Elementary File (EF) containing the key value, that is going to be written in the token. This means that special care was taken to avoid leaking the value as plaintext when importing it in the token. Unfortunately the key value already appeared in the clear: quite surprisingly, key generation re-uses the 8-bytes random string which is used by the authentication step (line 2) as the sensitive key value.

```
0 # Manage security environment
1 APDU: 00 22 41 b6 06 80 01 12 84 01 07
2 SW:   90 00
3 # Custom perform security operation
4 APDU: 80 2a 9e ac 16 90 14 59 b7 b5 0c 2e 69 4e 3f 7e 2f 06 7f 07 1d 8e dd de ba 8c c0
5 SW:   61 80
6 # Custom getData
7 APDU: 80 c0 00 00 80
8 SW:   9d 70 aa 8d c4 af 7a 88 ba e4 6c ab 47 3e 02 19 81 e5 85 53 8a 6a 1b 83 8c 73 39 29 9e 49 bb 24 a7 27 4f 8e 38 60 b6
        d1 71 c6 92 75 58 fe 33 78 d2 fe 99 5c 96 4e 3e 43 15 9d 67 f9 db 7b 8b 3c 29 d4 97 d5 ec 2e 46 7e 2b c9 c4 92 0f 38
        eb 65 11 2b e1 ba 61 33 7c a1 03 62 f4 2c 2c f2 52 85 2a ee ab 77 ca 6e 37 8e 3b 5a 57 dd c1 64 ea d0 76 71 2a 46
        0b bc d4 2a ef c0 6c 32 77 c3 5e 79 90 00
```

Listing 3: Forced signature sample

```
>>> signed = 0x9d70aa8dc4af7a88bae46cab473e021981e585538a6a1b838c7339299e49bb24a7274f8e3860b6
    d171c6927558fe3378d2fe995c964e3e43159d67f9db7b8b3c29d497d5ec2e467e2bc9c4920f38eb65112be1b
    a61337ca10362f42c2cf252852aeeab77ca6e378e3b5a57ddc164ead076712a460bbcd42aefc06c3277c35e79
>>> modulus =0xc1886b5f26ad5349426b8e8bfc9f73385d14f6cf2b2f1d95b080ae2df7a1db11b91d36db33f3b9
    8f16871774711c03b22d7d97939062031df2d15371173b468f9986701d144f315005ec99a71b226fc71b95660
    8c60747ceb4ac0c3725b7d04484ac286196975f18911361e28ec50b661273362131b4a4183e01667b090c96f9
>>> pubkey = 0x010001
>>> hex(pow(signed, pubkey, modulus))
'0x1fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
'ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
'fffffffffffffffffffffffffffff0059b7b50c2e694e3f7e2f067f071d8edddeba8cc0L
```

Listing 4: Signature verification in Python

As a proof of concept, we encrypted a zero-filled 8-bytes buffer using the `C_Encrypt` function with the generated key and a null initialization vector. We then performed the same encryption using the 8-bytes challenge as the DES key value obtaining the same value.

4.3 Bypassing attribute values

In all five tokens examined, `PKCS#11` attributes are interpreted by the middleware and do not have any import on the internal behaviour of the token. We performed a simple test by signing a text using an RSA key having the attribute `CKA_SIGN` set to false:

1. take a private RSA key with `CKA_SIGN` false;
2. verify that it cannot sign a message via the `PKCS#11` API, as expected;
3. perform the sign operation manually, via `APDU`, using the private key and the message. Some tokens use the standard ISO-7816 command `PERFORM SECURITY OPERATION` and some others use a proprietary command but, in both cases after sniffing, it is easy to replicate any valid `APDU` trace for a signature.

This confirms that the low-level behaviour of the token is not compliant to `PKCS#11` specification as it allows to perform signature under a key that has `CKA_SIGN` attribute set to false. Since the behaviour of all five tokens is similar, in Listing 3 we illustrate the case of Safesite Classic as a representative `APDU` example trace. At line 4 the message is sent to the token and, at line 8, the corresponding signature is returned.

We can verify that signature corresponds using Python shell, as shown in Listing 4. In particular, notice that the obtained message corresponds to the one we signed.

4.4 RSA session keys

When using session RSA keys on the eToken PRO, we discovered that key generation, encryption and decryption operations are performed inside the library. This means that the value of the private key is exposed in the clear out of the token.

Token	Auth.	Sensitive symmetric keys		Bypassing attribute values	RSA session keys	
		PKCS#11[1]	APDU		PKCS#11[1]	APDU
eToken PRO	✓[2]	✓	✓	✓	✗	✓[4]
ASEKey	✓[2]	✗	✓[3]	✓	✗	✗
SecurID	✓[2]	✓[5]	✓	✓	✗	✗
Safesite Classic	✓	✗	✗	✓	✗	✗
Siemens CardOS	✓	✗	✓[4]	✓	✗	✗

[1] PKCS#11-level attacks discovered in [5], for comparison.
[2] Requires reverse engineering of the authentication algorithm and bruteforcing.
[3] Leakage occurs only during generation.
[4] Requires access to middleware memory.
[5] Possible for RSA Authentication Client version $< 3.5.3$.

Table 4: Summary of the vulnerabilities found .

Even if one might regard to session keys as less important than long-term keys, as we already discussed in subsection 4.2 for Siemens CardOS, PKCS#11 still requires that if such keys are sensitive they should not be exported out the token in the clear. For example we can generate a session key which, at some point before the end of the session, is persisted in the token's memory by calling the C_CopyObject function. Clearly this newly created object cannot be considered secure as the value of the private RSA key has already been leaked in the clear out of the token.

5 Security analysis

In Table 4 we summarize the APDU-level attacks we found on the five devices. In the columns labelled PKCS#11 we also report the PKCS#11 attacks from [5], for comparison. In particular, the only token that allows for PKCS#11 Clulow-style attack extracting a sensitive key in the clear is eToken PRO. For SecurID we reported that it was possible to directly read the value of sensitive symmetric keys and RSA released a fix starting from RSA Authentication Client version 3.5.3.[3] In the literature we found no known API-level attacks on sensitive keys for the remaining devices.

All devices are affected by attacks on the PIN, some of which requiring reverse engineering and brute forcing, and by attacks bypassing key attributes. For what concerns sensitive keys, only Safesite Classic is immune to attacks. For the remaining four tokens we have reported new attacks that compromise sensitive keys that are instead secure when accessed from the PKCS#11 API.

In order to clarify under which conditions the attacks are possible we cross-compare Table 1 with Table 4 producing table Table 5. In particular, for each device we take the vulnerabilities reported in Table 4 and we check from Table 1 if the combination attacker/application offers the necessary conditions for the attack. We omit the Admin attacker as it is in fact equivalent to the User attacker when the application is monolithic. In particular, we observe that:

User/Monolithic the attacker can attach to the process and eavesdrop the PIN at the PKCS#11 level. Knowing the PIN the attacker can perform any operation and inspect the process memory. So all attacks of Table 4 are enabled;

User/Separate authentication mechanism the attacker cannot eavesdrop the PIN directly. Interestingly PKCS#11-level attacks and attribute bypass are still possible through a MITM on the middleware. Moreover, APDU-level attacks on keys are still valid as they only require to eavesdrop the APDUs;

[3] See https://secgroup.dais.unive.it/projects/tookan/

Attacker	Application	Auth.	Sensitive symmetric keys		Bypass attribute values	RSA session keys	
			PKCS#11[4]	APDU		PKCS#11[4]	APDU
Aladdin eToken PRO							
User	Monolithic	✓	✓	✓	✓	✗	✓
	Sep. Auth.	✗	✓[1]	✓	✓[1]	✗	✓
	Sep. Privileges	✓	✓	✗	✗	✗	✗
	Sep. Auth.&Priv.	✗	✓[1]	✗	✗	✗	✗
Physical	Any	✓[2,5]	✓[1]	✓	✓[1]	✗	✗
Athena ASEKey							
User	Monolithic	✓	✗	✓	✓	✗	✗
	Sep. Auth.	✗	✗	✓[6]	✓[1]	✗	✗
	Sep. Privileges	✓	✗	✗	✗	✗	✗
	Sep. Auth.&Priv.	✗	✗	✗	✗	✗	✗
Physical	Any	✓[2,5]	✗	✓	✓[1]	✗	✗
RSA SecurID 800							
User	Monolithic	✓	✓[7]	✓	✓	✗	✗
	Sep. Auth.	✗	✓[1,7]	✓	✓[1]	✗	✗
	Sep. Privileges	✓	✓[7]	✗	✗	✗	✗
	Sep. Auth.&Priv.	✗	✓[1,7]	✗	✗	✗	✗
Physical	Any	✓[2,5]	✓[1,7]	✓	✓[1]	✗	✗
Safesite Classic TPC IS V1							
User	Monolithic	✓	✗	✗	✓	✗	✗
	Sep. Auth.	✗	✗	✗	✓[1]	✗	✗
	Sep. Privileges	✓	✗	✗	✗	✗	✗
	Sep. Auth.&Priv.	✗	✗	✗	✗	✗	✗
Physical	Any	✓[2]	✗	✗	✓[1]	✗	✗
Siemens CardOS V4.3b							
User	Monolithic	✓	✗	✓	✓	✗	✗
	Sep. Auth.	✗	✗	✓	✓[1]	✗	✗
	Sep. Privileges	✓	✗	✗	✗	✗	✗
	Sep. Auth.&Priv.	✗	✗	✗	✗	✗	✗
Physical	Any	✓[2]	✗	✗	✓[1]	✗	✗

[1] Requires MITM.
[2] Through a keylogger or a USB sniffer.
[3] Only APDU payloads, cannot access middleware memory.
[4] PKCS#11-level attacks discovered in [5], for comparison.
[5] Requires reverse engineering of the authentication algorithm and bruteforcing.
[6] Leakage occurs only during generation.
[7] Possible for RSA Authentication Client version < 3.5.3.

Table 5: Summary of vulnerabilities with respect to attackers and applications.

User/Separate privileges the attacker can still eavesdrop the PIN and work at the PKCS#11 level but all APDU-level attacks are prevented. In this setting the only insecure token is eToken PRO since it allows for PKCS#11-level attacks on sensitive keys;

User/Separate authentication and privileges this is the more secure setting: the attacker con only perform PKCS#11-level attacks on eToken PRO through a MITM, since he cannot learn the PIN. All the other tokens are secure;

Physical/Any application through a keylogger or a USB sniffer the attacker can learn the PIN. In case of a USB sniffer, for the tokens adopting challange-response it is also necessary to reverse-engineer the protocol in the library and perform brute-forcing on the PIN. APDU-level attacks are possible only when the keys are transmitted from/to the device. So, for eToken PRO RSA session keys and Siemens CardOS symmetric keys the attacks are prevented, as keys are directly handled by the library and are never transmitted to the device. Other attacks can be performed only through a MITM at the USB level.

5.1 Fixes and mitigations

Compliant PKCS#11 devices should implement all the cryptographic operations inside the hardware. This would prevent all of the attacks we have discussed so far, except for the ones on authentication. However, fixing this at the hardware level requires to redesign the device and is probably just not affordable, in general.

We have seen, however, that having separate authentication and privileges is a highly secure setting that fixes the problem of cryptographic operations implemented at the library level and, at the same time, protects PIN authentication. It is worth noticing that running the middleware with separate privileges can be done transparently to the application while having separate authentication requires to modify the application so that the login step is managed by separate software or hardware.

An alternative way to mitigate attacks on PIN, with no changes in applications, could exploit the OTP functionality of the devices with a display, such as SecurID. A one-time PIN might be generated by the token and shown on the display asking the user to combine it with the secret token PIN. In this way, offline brute-forcing would be slowed down by the longer, combined PIN and, even if successful, would require physical access to the token in order to re-authenticate since part of the PIN is freshly generated by the token each time the user authenticates.

6 Conclusion

We have presented a new threat model for the PKCS#11 middleware and we have analysed the APDU-level implementation of the PKCS#11 API for five commercially available devices. Our findings show that all devices present APDU-level attacks that, for four of them, make it possible to leak sensitive keys in the clear. The only smartcard immune to attacks to keys is Safesite Classic. We have also found that all devices are vulnerable to attacks that leak the PIN if the middleware is not property isolated and run with a different privilege (which is usually not the case). Moreover, attackers with physical access could sniff an authentication session through the USB port and brute-force the PIN once the authentication protocol has been reverse-engineered.

We have reported our finding to manufacturers following a responsible disclosure principle and we are interacting with some of them to provide further information and advices.

References

[1] R. Anderson. The correctness of crypto transaction sets (discussion). In *Revised Papers from the 8th International Workshop on Security Protocols*, pages 128–141, London, UK, 2001. Springer-Verlag.

[2] Guillaume Barbu, Christophe Giraud, and Vincent Guerin. Embedded Eavesdropping on Java Card. In *27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012.*, pages 37–48, 2012.

[3] M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 220–234, Paris, France, 2001. Springer.

[4] M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, 34(10):67–75, October 2001.

[5] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 260–269. ACM, 2010.

[6] J. Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, volume 2779 of *LNCS*, pages 411–425. Springer, 2003.

[7] Danny De Cock, Karel Wouters, Dries Schellekens, Dave Singelee, and Bart Preneel. Threat modelling for security tokens in web applications. In *Communications and Multimedia Security*, pages 183–193. Springer, 2005.

[8] Gerhard de Koning Gans and Joeri de Ruiter. The smartlogic tool: Analysing and testing smart card protocols. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 864–871, 2012.

[9] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, November 2010.

[10] Andriana Gkaniatsou, Fiona McNeill, Alan Bundy, Graham Steel, Riccardo Focardi, and Claudio Bozzato. Getting to know your card: Reverse-engineering the smart-card application protocol data unit. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 441–450. ACM, 2015.

[11] ISO/IEC 7816-4. *Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange*, 2013.

[12] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.

[13] Steven J. Murdoch, Saar Drimer, Ross J. Anderson, and Mike Bond. Chip and PIN is broken. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 433–446, 2010.

[14] OASIS Standard. *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40*, April 2015. `http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html`.

[15] RSA Laboratories. *PKCS #11 v2.30: Cryptographic Token Interface Standard*, April 2009. `http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm`.

[16] Bruce Schneier, Adam Shostack, et al. Breaking up is hard to do: modeling security threats for smart cards. In *USENIX Workshop on Smart Card Technology, Chicago, Illinois, USA, http://www. counterpane. com/smart-card-threats. html*, 1999.

[17] Adam Shostack. Experiences threat modeling at microsoft. In *Modeling Security Workshop. Dept. of Computing, Lancaster University, UK*, 2008.

[18] Frank Swiderski and Window Snyder. *Threat Modeling.* Microsoft Press, Redmond, WA, USA, 2004.

[19] Linzhang Wang, Eric Wong, and Dianxiang Xu. A threat model driven approach for security testing. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, SESS '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.

[20] P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, August 2005.