Extending Secure Execution Environments Beyond the TPM

(An Architecture for TPM & SmartCard Co-operative Model)

Talha Tariq

Technical Report

RHUL-MA-2009-09

16th February 2009

Department of Mathematics

Royal Holloway, University of London

Egham, Surrey TW20 0EX, England

http://www.rhul.ac.uk/mathematics/techreports

TABLE OF CONTENTS

ACKNOWLEDGEMENTS VI
1.0 EXECUTIVE SUMMARY 1
2.0 TRUSTED COMPUTING 3
2.1 Trusted Computing Group 3
2.2 Motivation for Hardware Based Security Device 3
2.3 Minimum Trusted Platform Requirements 4
2.4 The Trusted Platform 4
2.4.1 Root of Trust for Measurement 4
2.4.2 Root of Trust for Storage 5
2.4.3 Root of Trust for Reporting 5
2.5 The Trusted Platform Module 5
2.5.1 Input / Output: 5
2.5.2 Key Generation 5
2.5.3 Cryptographic Engine / Co-processor 5
2.5.4 HMAC Engine 5
2.5.5 SHA-1 Engine 6
2.5.6 Power Detection 6
2.5.7 RNG 6
2.5.8 NV Memory 6
2.5.9 Volatile Memory 6
2.5.10 Opt-In 6
2.5.11 Key Storage: 6
2.5.12 PCRs 6
2.5.13 Execution Engine 6
2.5.14 Authentication and Authorization 7
2.5.14.1 Physical Presence 7
2.5.14.2 Authorization Data 8
2.5.14.3 Authorization Protocols 8
2.5.15 TPM Services 9
2.5.15.1 Integrity Measurement, Recording and Reporting 9
2.5.15.2 Protected Storage 9
2.5.15.3 Monotonic Counters 10
2.6 Summary 10
3.0 OVERVIEW OF SMART CARDS 12
3.1.1 Introduction 12
3.2 Smart Card Industries 12
3.3 Smart Card Charecteristics 13
3.3.1 Portable Platform 13
3.3.2 Processing Power: 13
3.3.3 Computation and Performance limitations 13
3.3.4 Storage and memory 14

3.3.5 Special Purpose Crypto-Processors: 14
3.3.6 Tamper Resistance 14
3.3.7 Protected Storage 15
3.3.8 Secure Execution 15
3.4 Multi-Application programmable SmartCards 15
3.4.1 Smart Card Software 15
3.4.1.1 Operating System 15
3.4.1.2 Applications 16
3.4.1.3 Runtime Environment 16
3.4.1.4 Loader 17
3.4.2 The .NET Smart Card 17
3.4.2.1 The Application Lifecycle 18
3.4.2.2 The Common Language Runtime and its benefits 20
3.5 Summary 21
4.0 TPM & SMARTCARD COUPLING FOR ENHANCED SECURITY SERVICES 22
4.1 Introduction: 22
4.2 A TPM Architecture for Secure Execution 24
4.2.1 A hypothetical architecture of the TPM-Internal Execution Environment 24
4.3 Smart Card Capability Requirements 26
4.3.1 Platform Differences between TPM and the Smart Card 26
4.3.2 Coupling Challenges 26
4.4 Experimental Platform 27
4.5 Secure TPM and Smartcard Binding 27
4.5.1 Extended Crypto Remoting for End-to-End Encryption 28
4.5.1.1 .NET Remoting Overview 28
4.5.2 Securing Remoting communication between the TPM and the SmartCard 32
4.5.2.1 Custom Sinks (Extending .NET Remoting to support encryption) 34
4.5.3 Data Marshalling (Data Transformation between the TPM and the SC) 41
4.5.4 Securing Remoting Components in a Distributed Environment 41
4.5.4.1 Authentication and Encryption with IIS 42
4.6 Summary 43
5.0 EXTENDED TPM SERVICES PROVIDED BY THE SMART CARD 45
5.1 Introduction 45
5.2 Flexible Authorization 45
5.2.1 Introduction: 45
5.2.2 Smart Card participation in Authorization Protocols 48
5.2.2.1 Simple Storage of Authorization Data 48
5.2.2.2 Computation of Shared Secret 49
5.2.2.3 Secure Generation and Insertion of Authorization Data. 51
5.2.3 Benefits of Enhanced Authorization 52
5.3 Validating TPM Generated QUote (Attestation) 53
5.3.1 Introduction: 53
5.3.2 Function Details for SC_ValidateTPMQuote 54
5.3.3 The Process: 54
5.4 Enhanced Sealing and Binding 55
5.4.1 SmartCard Un-Bind 56

5.4.1.1 Function Details for SC_Unbind 56
5.4.2 SmartCard Bind / Seal 58
5.4.2.1 Function Details for SC_Bind 59
5.5 Enhanced Virtual Monotonic Counters 60
5.5.1 Issues with the TPM Monotonic Counters: 60
5.5.2 Smart Card Enhanced Counters: 61
5.5.2.2 Implementation Details and Function Calls: 62
5.5.2.3 Helper Functions: 63
5.5.2.4 Implementing .NET Transactions in the Smart Card 64
5.6 Summary 66
6.0 SAMPLE APPLICATIONS FOR THE TPM & SC CO-OPERATIVE MODEL 67
6.1 Enhanced Digital Signature 67
6.2 Roaming DRM 70
6.3 Some Further Applications 73
6.3.1 E-cash Tokens that only work with authorized platforms 73
6.3.2 Ease of data migration between trusted platforms 73
6.3.3 Crypto schemes not supported by the TPM 73
6.3.4 Flexible Authorization Applications 74
6.3.5 Strengthen Remote Attestation 74
6.3.6 Smart Card SIM Bound with Mobile TPM 74
7.0 SUMMARY AND CONCLUSION 75
7.1 The Trusted Platform Module 75
7.2 The SmartCards 75
7.3 The SmartCard and TPM Coupling 76
7.4 Enhanced TPM and SMART CARD Services 77
7.5 Some Applications Developed with the coupled services 78
7.6 Similar Work 79
APPENDIX A – USING TPM SERVICES IN WINDOWS 81
APPENDIX B - .NET CARD SPECIFICATIONS [8] 86
APPENDIX C – DIFFERENCES BETWEEN MICROSOFT .NET AND GEMALTO .NET FRAMEWORK [33] 91
REFERENCES 93

LIST OF FIGURES

Figure 2-1 The Trusted Platform Module Components ... 7
Figure 3-1 Gemalto. NET Card V2 ... 17
Figure 3-2. The Gemalto Smart Enterprise Guardian ... 18
Figure 3-3 Smart Card .NET Architecture ... 18
Figure 4-1 A Smart Card TPM Cooperative Model ... 23
Figure 4-2 A TPM Architecture for Trusted Execution ... 25
Figure 4-3 Simplified .NET Remoting Architecture. ... 30
Figure 4-4 Session based symmetric key exchange ... 34
Figure 4-5 .NET Remoting with pluggable custom sinks ... 38
Figure 4-6 A Distributed TPM / Smart Card co-operative model ... 42
Figure 5-1. Password based TPM Administration in Windows Vista ... 47
Figure 5-2 Authorization Data Stored & Presented by SmartCard in Authorization Protocols ... 49
Figure 5-3 SmartCard Calculating the shared secret in Authorization protocols ... 51
Figure 5-4 Smart Card participating in ADIP ... 52
Figure 5-5 TPM Attestation Validation by the Smart Card ... 55
Figure 5-6 TPM Bound Data decrypted by a SmartCard by SC_UnBind. ... 58
Figure 5-7 Sample scenario for SmartCard Seal / Bind data for a particular TPM ... 59
Figure 5-8 Smart Card Enhanced Seal / Bind ... 60
Figure 5-9 Smart Card Enhanced Monotonic Counter Creation and Usage ... 62
Figure 6-1 Enhanced Digital Signature by TPM and SC ... 68
Figure 6-2 Smart Card Enhanced Flexible Sealing ... 72
Figure 7-1 The Trusted Base Services Library ... 85
Figure 7-2 Block Diagram of the SLE88CFX2000P hardware [32] ... 88
Figure 7-3 Microsoft Crypto Architecture for Smart Cards [8] ... 89
Figure 7-4 Crypto Architecture in .NET Card [8] ... 90

ACKNOWLEDGEMENTS

First of all I would like to thank Paul England from Microsoft for his constant inspiration, guidance and advice over the whole duration of the project.

I would also like to thank my supervisor Chris Mitchell for his invaluable guidance and support throughout my studies at Royal Holloway. Without his feedback, comments and guidance this thesis would have never been completed.

Special thanks to the David Robinson and the System Incubation Team at Microsoft Research for all their valuable comments and suggestions and for making the whole summer at Microsoft really enjoyable.

I would like to thank my parents for their continuous love, support and motivation over the years that I have been away from home.

And finally I would like to thank Microsoft Corporation for sponsoring this project and giving me the opportunity to work at Microsoft Research in Redmond.

1.0 EXECUTIVE SUMMARY

This project discusses some of the shortcomings and limitations of secure execution with the current state of the Trusted Computing Group (TCG) specifications. Though we feel that the various industry initiatives taken by the TCG and CPU manufacturers for hardware based platform security are a step in the right direction, the problem of secure isolated code execution and TCB minimization still remains unsolved. This project proposes and implements an alternative architecture for secure code execution. Rather than proposing recommendations for hardware changes or building isolated execution environments inside a Trusted Platform Module (TPM), we use a platform that provides related, yet different services for secure / trusted code execution; couple its functionality and bind it to a TPM using cryptographic primitives. For the purpose of this study we used multi-application programmable SmartCards but similar work can also be implemented on other platforms as long as they meet some pre-requisites described in Section 4.3.

Though newer hardware platforms such as IntelTXT [**1**] (Trusted Execution Technology; formerly known as LaGrande) or AMD-V add support for native virtualization and secure interfacing with the TPM, the solution implemented in this project assumes a highly un-trusted environment and works on general purpose commodity hardware. Implementing a solution like this allows application developers to focus exclusively on the functionality and security of just their own code. Hence enabling them to execute their applications in isolation from numerous shortcomings and vulnerabilities that exist both in the form of hardware and software attacks. Furthermore we provide an interface to extend the existing functionality of the TPM by implementing special purpose code modules inside a smart card which can be used for all the functionalities missing in the TPM (for example replace-able cryptographic algorithms) yet required by high assurance and security sensitive applications. Furthermore by making small application closures running inside the secure execution environment of smart cards, we can minimize the TCB that a user needs to trust.

We first discuss the challenges we face in the coupling process and the platform differences between the TPM and a Smart Card. We also discuss what solutions are possible and impossible in this scenario. Then we describe our implementation of a secure TPM / Smart Card cryptographic binding that gives us assurances of strong authentication with confidentiality and integrity services for the applications built with the coupled architecture. We move forward to describe our implementations of some of the enhanced TPM / Smart Card coupled services that were not possible with either a TPM or Smart Card alone and we discuss how these enhanced services add value to the current applications. With these enhanced TPM services we implement some applications that change the way conventional TPM or Smart Card applications are perceived. Finally we shed some light on potential future applications and future work.

This report is divided into 7 chapters and 3 appendices:

- *Chapter 1* gives an overview of the project and contents of this thesis.

- *Chapter 2* gives a brief overview of Trusted Computing, TPM, its components, its services and some of the limitations.
- *Chapter 3* gives an overview of the Smart Cards, their characteristics. We discuss the platform differences with the TPM, application development environments and the lifecycle. We also describe the .NET Card and what were the special features that motivated us to choose a smart card for the coupling platform.
- *Chapter 4* describes the limitations and differences of both SmartCards and the TPMs. It discusses a hypothetical ideal secure platform processor with a trusted execution environment built into it. But since such a platform is not available we discuss how we can extend and couple TPMs functionality with some other platform i.e. the SmartCards. We discuss the capability requirements of SmartCards and describe a secure TPM/ SmartCard binding architecture.
- *Chapter 5* describes the actual coupling implementation. Detailed design, analysis and implementation details are given with code snippets for the reader's interest of the implementation of extending the secure execution environments of the TPM. Enhanced Services exposed by the smart card are also described in detail. Background motivation, challenges and special requirements are also discussed in detail.
- *Chapter 6* continues the theory and practice from Chapter 4 and 5 and we describe some sample applications that were developed using the building blocks from Chapter 4. 2 Detailed applications are discussed from end to end and the implementations of some other few are described briefly.
- *Chapter 7* summarizes the whole project with directions of further work and similar research areas.

- *Appendix A* gives a brief overview with screenshots of how to use the TPM services in Windows Vista. It explains the TPM ownership process and how to enable disable TPM calls from the Microsoft Management Console.
- *Appendix B* lists the technical details and capabilities of the .NET smart card. This would give the reader a general idea of the platform used, its capabilities and limitations.
- *Appendix C* describes some of the major differences between the SmartCard .NET Compact framework and the standard .NET framework. We recommend any reader who wishes to start development on a .NET Smart Card to analyze these differences before making any efforts on application development as these differences will help a developer to have an idea of the limitations and differences beforehand.

2.0 TRUSTED COMPUTING

This Chapter gives a brief overview of Trusted Computing (TC), The Trusted Computing Group (TCG), the Trusted Platform Module (TPM) and its services. We briefly describe the industry trend that led to the development of the Trusted Computing Group and hence trusted computing. We discuss to the extent Trusted Computing solves some of these problems and discuss a few of its problems and limitations. We also briefly describe the components of a TPM and some of its major services including protected storage and attestation capabilities.

2.1 TRUSTED COMPUTING GROUP

The Trusted Computing group (TCG) is a joint consortium of some of the major hardware and software manufacturers. The TCG is a successor to the Trusted Computing Platform Alliance (TCPA), which was an initiative started by AMD, Hewlett-Packard, IBM, Infineon, Intel, Microsoft, and Sun Microsystems. At the time of this writing it contains 6 fundamental organization components and 13 working groups.[1]

A trusted platform provides a secure mechanism for verifying the integrity of a platform remotely. This is done by examining a chain of trust which goes through the examination of hardware bios or boot block, master boot record, operating system and finally the application state. A challenger (remote server or any authority interested in verifying the integrity of the system) checks the appropriateness of the integrity measurement process and compares the supplied values with the expected values and hence makes a decision on the next action.

2.2 MOTIVATION FOR HARDWARE BASED SECURITY DEVICE

All modern systems of software are becoming increasingly complex. As the computation and storage capacities of general purpose hardware are increasing, so is the complexity and size of the software running on them. A typical operating system is millions of lines of code and as more and more applications are added, the size becomes exponential. Several studies have shown and most of the ever emerging vulnerabilities in software prove that typical software products have at least a few critical security vulnerabilities per thousand lines of source code. Thus, a typical system has potentially thousands of security bugs and it only takes a matter of time for the security industry to find them. Most of the earlier systems (both hardware and software) were never designed with security in mind, hence building security on top of existing insecure software is not a termed a good idea as the amount of effort to replace billions of lines of code is not practical, neither in terms of cost of time and money, and nor does it gives us assurance of a minimum trusted base. Furthermore without hardware based platform security it is almost impossible to detect the presence of any malicious code as the program execution on a conventional computing platform can be easily attacked. Even with a little bit of hardware

[1] https://www.trustedcomputinggroup.org/home

support, it is quite easy to implement a better and more secure solution by adding another layer of security which if trusted can easily detect and report compromise.

2.3 MINIMUM TRUSTED PLATFORM REQUIREMENTS

The minimum requirements as put forward by a number of studies for trusted platforms are broadly divided into 4 categories:

1. Protected capabilities and shielded locations.
2. Integrity metric reporting secure key management. The platform can only be trusted if this fundamental root is trusted.
3. Platform attestation mechanisms. Attestations in generic terms mean the process by which the assurance of some information is guaranteed. That information is usually the integrity metric reporting of the platform software and state.
4. Integrity metric storage and measurement functionality that records, stores and reports the evidence of platforms claimed identity.

Some more similar studies add more connected requirements for a minimal trusted platform. For example Microsoft Next Generation Secure Computing Base [**2**] also adds the following requirements beyond the ones described earlier;

5. Process isolation with assurances of separation process execution with the help of a secure isolation kernel.
6. Trusted path which is a secure input and output to and from the user.

2.4 THE TRUSTED PLATFORM

A trusted platform module is essentially a secure crypto-processor that is bound to a platform to provide trusted security services. These services include providing secure storage of keys, integrity measurement functionality that records, stores and reports the evidence of platform state. It also includes providing attestation mechanisms and protected shielded locations for data storage.

In order for a platform to be trusted, there has to be a minimum embedded root of trust implemented in the platform which can provide these services and can be trusted itself. At the minimum the 3 rudimentary roots of trust [**2**] are defined as follows:

2.4.1 Root of Trust for Measurement

The Root of Trust for Measurement (RTM) is a computing engine capable of making reliable integrity measurements. On a normal PC the Core RTM is implemented in the BIOS.

2.4.2 Root of Trust for Storage

The measurements taken by the RTM have to be securely stored for them to be trusted. Root of Trust for Storage is a computing engine capable of maintaining an accurate summary of integrity measurements that were made by the RTM.

2.4.3 Root of Trust for Reporting

Root of Trust for Reporting (RTR) is the computing engine capable of reporting data stored by the RTS.

The Root of Trust for Storage and the Root of Trust for Reporting together form the minimum functionality that should be offered by the TPM. Ideally the Core Root of Trust for Measurement must also be part of the TPM.

2.5 THE TRUSTED PLATFORM MODULE

A TPM contains a number of fundamental building blocks in order to implement the functionality of RTS and RTR. Every TPM has a set of keys associated with it. The most important is called the Endorsement Key which is a unique key pair per TPM. All other keys are protected under a key hierarchy under a master key which is called the Storage Root Key. This key is also stored securely inside the card and all other keys can be encrypted under SRK and stored separately.

The other major components of a typical TPM are:

2.5.1 Input / Output:

Manages information flow with encoding / decoding of communications.

2.5.2 Key Generation

Secure Asymmetric and Symmetric Key Generation. Also creates random nonces for use in Authorization protocols.

2.5.3 Cryptographic Engine / Co-processor

Used for encryption / decryption; digital signature generation / validation etc

2.5.4 HMAC Engine

HMACs are used for authorization proofs and command integrity verification if it has been modified during transit.

2.5.5 SHA-1 Engine

SHA-1 engine is also used extensively in lots of applications including authorization protocols, attestation and sealing calls.

2.5.6 Power Detection

Manages the TPM and platform power states

2.5.7 RNG

True Random Number generation for use in cryptographic protocols.

2.5.8 NV Memory

Holds identity information for e.g. the Endorsement Key and Storage Root Key. Also used for holding persistent state.

2.5.9 Volatile Memory

Used for storing active keys in use by the TPM (during encryption / decryption or signature functions).

2.5.10 Opt-In

This component is used when physical presence at the TPM must be demonstrated for e.g. during the ownership or clearing the TPM.

2.5.11 Key Storage:

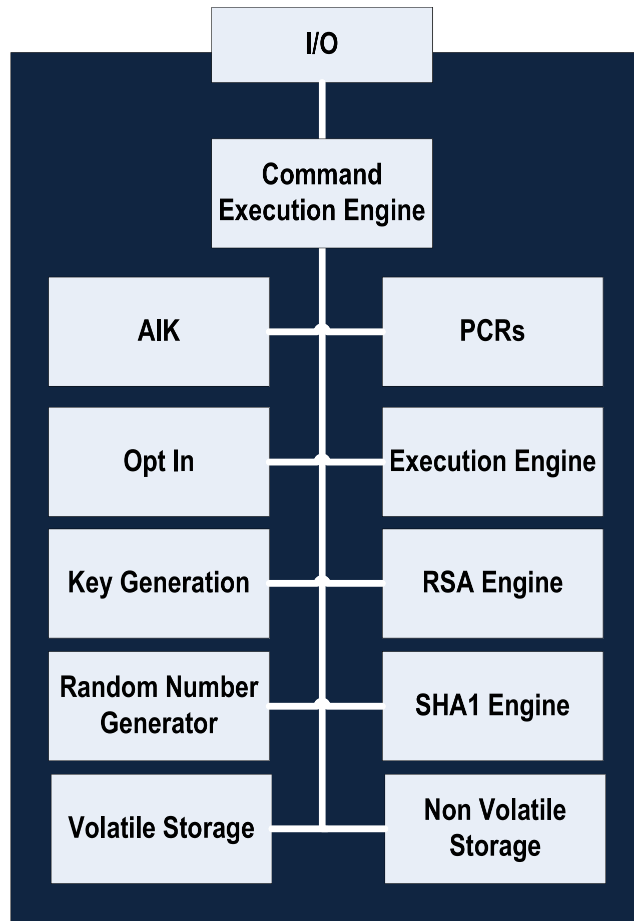
Unique EK, SRK key hierarchy stored permanently

2.5.12 PCRs

16, 20 byte registers used to hold integrity metrics.

2.5.13 Execution Engine

Runs program code for executing TPM commands.



The Trusted Platform Module Components

Figure 2-1 The Trusted Platform Module Components

2.5.14 Authentication and Authorization

Authentication to the TPM is required for most authorized operations. There are two main methods

1. Physical Presence (that requires some physical interaction with the platform)
2. Authorization Data (A secret value shared between the platform and the object owner)

2.5.14.1 Physical Presence

The TCG specifies Physical Presence for some sensitive operations. At the time of this writing[2], the TPMs that are being shipped are disabled by default. To enable them from the BIOS, or even clear them physical presence is required.

[2] February 09

2.5.14.2 Authorization Data

Possession or knowledge of an authorization data in the TPM context proves a complete ownership of a resource. The TCG defines it to be a shared secret between the object / resource and the owner.

Authorization Data are used in Authorization protocols which are used to:

- Handle the confidential creation of Authorization Data
- Provide proof of knowledge of authorization data
- Allow its secure update.

A typical authorization data is 20 bytes. This is equal to the output of a SHA-1 Cryptographic Hash which is shared between the object owner and the TPM.

There are 3 types of authorization data:

1. The Owner Authorization (for user authentication)
2. Migration Auth (for objects declared to be migratable, to be used when migrating the objects).
3. Auth data not associated with any particular object. These are created with every object and are required to allow use of the object in subsequent operations requested from the TPM.

In the main part of the project we will describe how we use a SmartCard to hold, store, generate and provide on demand the authorization data for different objects and making the authorization protocols more strong with multi-factor authentication and secure generation of AuthData.

2.5.14.3 Authorization Protocols

There are a total of 6 authorization protocols. Three of them are used for securely pass the authorization data to the TPM, the other three are related to authorization data creation and update. For details of these please see [**2**]. These protocols are also described in Section 5.2 where smart card is used to enhance the security of these protocols with stronger TPM / SC binding.

A. Protocols used for authorization data usage:

i. Object Independent Authorization Protocol (OIAP)
ii. Object Specific Authorization Protocol (OSAP)
iii. Delegate Specific Authorization Protocol (DSAP)

B. Protocols used for authorization data creation and update:

iv. Authorization Data Insertion Protocol (ADIP)
v. Authorization Data Change Protocol (ADCP)
vi. Asymmetric Authorization Change Protocol (AACP)

2.5.15 TPM Services

This section describes some of the most widely used services. Some of these important services (or their counterparts) would be implemented in the smart card in the TPM / SmartCard co-operative model to provide enhanced TPM services so this section is just to give a general idea to the user of the standard TPM services that are defined in the TCG model. For more details on these please see [**2**], [**3**], [**4**].

The main goals of a TPM are to protect the most sensitive information for example cryptographic keys for leaks, theft and attacks by malicious code. The purpose of storing keys (or at the minimum the parent and root keys) inside the TPM is to ensure that the private and symmetric keys never leave the TPM and hence can survive attacks outside the trusted boundary. The TPM has a very limited storage for keys; hence a lot of keys are encrypted and stored externally.

2.5.15.1 Integrity Measurement, Recording and Reporting

i. Integrity Measurements and SML

Measuring a Trusted platforms integrity means generation of measurement events which contain either measurement values or measurement digests which are the hash values representing the measurements. The measurement values are usually stored in a Stored Measurement Log (SML) outside the TPM [**2**]. The digests / hashes are stored in the platform configuration registers using the RTM, RTR and RTS capabilities. The measurements are carried out through different methods for e.g. an authenticated boot process is created with the CRTM initializing, measuring itself and carrying over the chain of trust measurements for the BIOS, boot loader, MBR, OS etc. The measurement values stored in PCRs are provided to a remote challenger through attestation.

ii. Attestation

Attestation is the reliable reporting of the platform state to a remote challenger. The integrity metrics are measured and stored in Platform Configuration Registers (PCRs) and a signature is created with an Attestation Identity Key (AIK). This TPM_Quote is provided to the remote challenger who makes a decision on the platform state by examining the PCR values containing the platform and software state.

2.5.15.2 Protected Storage

i. Key Hierarchy

As defined earlier, each TPM is equipped with a unique Endorsement Key which is stored in the TPM. All the keys used by the TPM are kept under a protected key hierarchy whose parent key is the Storage Root Key (SRK) which is also stored in the TPM.

ii. Sealing

Sealed messages are bound to a future platform state that must exist before the decryption is allowed. It associates the encrypted message with a set of PCR register values and a non-migratable asymmetric key. The encrypted message is essentially the symmetric key to encrypt the message. A sealed blob is created by selecting a range of PCR register values which hold the measurement data for the platform state and asymmetrically encrypting the PCR values plus the symmetric key used to encrypt the message. The TPM with the asymmetric decryption key may only decrypt the symmetric key when the platform configuration matches the PCR register values specified by the sender. Sealing is a powerful feature of the TPM. It provides assurance that the protected messages are only recoverable when the platform is functioning in a very specific known configuration.

iii. Binding

Binding is not a TPM function, but a TPM Bound object can only be decrypted by that particular platform. In this case an external data is encrypted under a TPM parent bound key.

iv. Wrapping

Wrapping allows an externally generated key to be encrypted under a parent root key. It can be used for a number of purposes; for e.g. creating a non migratable blob by wrapping a key under a non-migratable key or vice versa.

2.5.15.3 Monotonic Counters

A Monotonic Counter is a trusted atomic tamper resistant counter, whose value once incremented cannot be reverted back. This property is used for protection against replay attacks and freshness checks. The current TPM specifications [3] as of this writing limit the total number of base monotonic counters in a TPM to be 4 (which are called the base counters). Also, out of these 4; only 1 counter can be incremented per boot session, the others can only be read. To use another base counter the system has to be rebooted. The motivation for such a design was to have a monotonic counter per trusted operating system (max 4), so each operating system has its own counter which cannot be updated after its boot sequence [4].

Some of the problems and limitations of these Monotonic Counters are discussed in 5.5.1. An implementation of extended TPM counters with Smart Cards bound with a TPM is discussed in detail in 5.5.2.

2.6 SUMMARY

This chapter describes a brief overview of Trusted Computing Technology with special reference to what were the motivations for a trusted platform. We describe the need of a hardware based platform security device that provides integrity and trust assurances. With special purpose crypto-processors niche security services can be provided. We described the structure and functionalities of a TPM device as of TCG version 1.2 specifications, its components and some of its limitations.

Based on the background of these services and the limitations discussed, we study how we could enhance these services with emphasis on providing secure execution environments for a TPM by coupling it with a SmartCard.

3.0 OVERVIEW OF SMART CARDS

This Chapter gives a brief overview of SmartCards, how they differ from conventional computing platforms, their different types of standards and architectures. We describe how they are being used in the industry and what special characteristics of them became the motivation for choosing them as an extended platform for the SmartCard and TPM coupling process. The chapter also discusses some of the application frameworks with special emphasis on the .NET platform available for SmartCards. How we would use the SmartCard for coupling and the services implemented for our applications are detailed in chapter 4.0 and 5.0 . For further reading on SmartCards the reader is encouraged to see [**5**] and [**6**].

3.1.1 Introduction

A smart card (also sometimes known as an Integrated Circuit Card ICC) is a portable, tamper-resistant computer / platform used in many different industries worldwide. They come in different sizes such as SIM cards, Credit Cards etc. Most of the conventional identity and Payment smart cards are the same of a credit card, and it is embedded with a silicon integrated circuit (IC) chip. The chip provides numerous functions from memory to store data, a microprocessor to manipulate that data, persistent secure storage and sometimes a cryptographic coprocessor to perform complex instructions and modular arithmetic.

Smart cards that contain a microcontroller chip are sometimes called chip cards to distinguish them from cards that offer either memory storage only, or memory storage and non-programmable logic as they look quite similar from physical characteristics. These memory cards store data efficiently, but cannot manipulate that data because they do not contain a computing chip. Memory cards depend on host-side applications to perform whatever processing is required on the data that they store.

Chip cards are either fixed command set cards, which are programmed and soft masked in read-only memory (ROM) during manufacturing to interact with security infrastructure systems as portable, secure tokens; or post-issuance programmable cards, which can be used for multiple purposes simultaneously (for example, a card might be used as both as a security token and a rechargeable stored-value card), and which can be upgraded or repurposed while in the field, long after their initial manufacture-time programming and soft masking.

3.2 SMART CARD INDUSTRIES

Smart Cards are one of the most under-estimated technologies present in the world today, yet they are ubiquitously used in a number of different industries for providing tamper resistant security and applications such as

- Telecom Industry (SIM Cards)
- Banking (EMV)

- Transport (e.g. Oyster)
- Identity Cards (Microsoft Key-badges)
- Passports (ICAO)
- Health Care
- Access Control
- Satellite TV
- RFID Tagging products etc.

3.3 SMART CARD CHARECTERISTICS

3.3.1 Portable Platform

Smart cards come in different shapes and unlike some of the other hardware based security modules such as HSM or TPM; they are meant to be portable. The physical characteristics are defined in an ISO standard 7816-1[**7**], Identification Cards - Integrated Circuits(s) Cards with Contacts - Physical Characteristics. The reader is encouraged to refer to them for all the details.

3.3.2 Processing Power:

Just like any other hardware, SmartCards come in various different processing capabilities and variants from 8-bit microcontroller to greater than 32bit architectures. The choice of the processing capabilities entirely depends on the application and the target audience. The card we used in this study (The Gemalto .NET Smart Card [**8**] has a 32bit microcontroller and 47kb of ROM.

3.3.3 Computation and Performance limitations

A smart card is very different from a conventional general purpose computing platform and is highly focused for specialized security applications. Hence it has a number of limitations comparing to other computing devices like PC etc. Only a few limitations are pointed out for the reference to give an idea of the challenges that are faced in using such a platform. The SmartCard relies on clock and power from an external source (reader) hence it is helpless alone and the applications can only run when the card is inserted in the reader. The chip is extremely restricted on power consumption, hence effecting clock speed and directly putting a limitation on processing capabilities. Also, the SmartCard has no “User Interface” for management or administration. Furthermore All I/O is done through the contacts on the chip. The transport protocol is called Application Protocol Data Unit (something similar to layer 7 protocol in TCP/IP) which has a packet size of around 250 bytes. Furthermore most of the conventional smart card platforms and readers offer a synchronous I/O, limiting the speed

3.3.4 Storage and memory

As with processing power the storage capacities of SmartCards also vary. Though newer chip fabrication technologies have taken the storage space up to a gigabyte, a mass majority of smartcards in the industry as of this writing contain a few hundred kilobytes of persistent memory. The memory structure of a SmartCard is usually divided into 3 different types

RAM – *Random Access Memory* which is volatile and mutable. Contents in RAM are not preserved when power to the card is removed.

ROM – *Read Only Memory* is persistent and non-mutable. ROM is mostly used to load with the program code which would never be modified (for e.g. the Card Operating System) .This loading is done during the manufacturing of the card. ROM contains the cards operating system routines, permanent data, and permanent user applications. Contents in ROM are preserved when power to the card is removed.

EEROM - *Electrical Erasable Programmable Read-only Memory* is a persistent and mutable memory used for data storage on the card. Content in EEPROM is preserved when power to the card is removed. It is usually meant to load programmable code and updates later in the smart card lifecycle after post issuance.

3.3.5 Special Purpose Crypto-Processors:

Many of the recent smartcards are equipped with a built-in crypto processor for making the crypto arithmetic operations faster. These special purpose crypto-processors are specially optimized for modular arithmetic of RSA and round operations of symmetric cryptography. Special hardware and software techniques are also implemented for protection against side channel or hardware attacks on these processors against attacks on cryptographic keys [**6**].

3.3.6 Tamper Resistance

Unlike other general purpose computing platforms, SmartCard industry puts a lot of effort, technology and investment in both hardware and software protection techniques for tamper resistance. A smart card plays an important integral role in providing authentication, confidentiality, integrity and non repudiation services in many widely used applications in the industry including telecom, banking, identity management etc. Hence since the smartcard is roaming, it is attacked more often so there is an important need of assurance for the integrity and security of the smartcard platform itself. Most of the smart cards today provide adequate protection both against physical and logical attacks. Listed below are just some of the general security and protection features to give the general reader an idea of the security features implemented by the SmartCard industry for protection against a number of hardware and software attacks. For detailed analysis please see [**6**] and [**5**].

Chip Design: Memory scrambling and added dummy structures to prevent circuit analysis, active current carrying layer for layered protection, memory encryption to prevent bus reading etc.
Physical Protection: Voltage analysis, monitoring of passivation layers, frequency monitoring for timing attacks, temperature monitoring, light sensors etc.
Logical Protection: Encryption of storage, implementation of noise free and constant cryptographic algorithms to defeat side channel analysis. Application frameworks for Operating System Security and applet firewalls for encapsulating applications. Secure data transmission techniques, True Random Number Generation and Error recovery functions etc.

3.3.7 Protected Storage

The smart card provides protected storage for secure storage of keys, applications and other sensitive data. For example the card we use for our sample implementations includes Token and Role based security for file system storing the sensitive data. Further protection mechanisms can be implemented for e.g. PIN authentication and onCard encryption.

3.3.8 Secure Execution

One of the most important features and the reason of success of smart cards is their capability of providing isolated, secure execution of applications. The security of the software is managed at all layers from a minimal operating system to provide isolated environments for application deployment and execution. The details of the software isolation that a .NET Card provides for such functionality are listed in the next section. Some related functionality is also implemented in the hardware to help provide a secure execution environment and protect the program execution from external threats, analysis and side channel attacks. Some of the categories of these attacks are listed in 3.3.6. For more details please see [**6**].

3.4 MULTI-APPLICATION PROGRAMMABLE SMARTCARDS

There are a number of different industry initiatives since the last decade that have been providing these technologies for general and special purpose programming environments for the SmartCard industry. Some of the major frameworks available are JavaCard [**9**], Multos [**10**] and .NET Card [**8**]. For the purpose of this study, the focus is only on .NET Cards as they are more integrated and suited for our requirements and provide all the fundamental building blocks that meet the TPM coupling requirements. Furthermore they are natively supported in Windows and since they provide a .NET environment for programming they are highly well suited over other platforms.

3.4.1 Smart Card Software

The SmartCard software platform typically consists of the following.

3.4.1.1 Operating System

The Operating system in a SmartCard is very similar to operating systems provided for mass market general purpose PCs in terms of the basic responsibilities. The only difference is that the OS of a smartcard is very small and compact. It is the first layer of abstraction on the raw hardware and provides services for managing communication and memory operations for the smart card applications. It is also responsible for all I/O related activities and providing interfaces to the cryptographic processor if any for cryptographic algorithms. There are fundamental differences between an Operating System, application platforms and runtimes. Multos for instance is an OS, GlobalPlatform[3] is a platform and .NET / Java are runtime environments.

3.4.1.2 Applications

Application development for SmartCards has come a long way in the recent few years. Smart Card based applications are used in a number of industries as mentioned earlier in 3.2. Before the JavaCard and .NET Cards became popular, the smartcard based applications were developed for a specific focused industry, chip / IC and written in native code by specialized community of smart card developers. These applications were loaded immutably in the smartcard and could not be changed once loaded. Hence application management was a nightmare and an update usually required a re-issue of card with re-deployment of the application. This was costly both in terms of time and money and the entire smart card lifecycle needed to be known in advance for the application to be built.

These problems motivated the industry and newer application platforms like JavaCard, .NET Card and Multos soon emerged. Writing applications on these platforms shielded the complexity of the native hardware; instruction set and opened the doors for general developers to write their applications on the smart card.

The applications on these platforms are loaded in EEPRROM and can be erased and loaded indefinitely. The post issuance update capability made a huge cost and management difference to the telecom, banking and transportation industries. For instance the telecom industry moved to Over-The-Air updates [**5**] (OTA) for application updates which can update the SIM applications in the hands of the consumers directly.

The Gemalto .NET Card contains an Intermediate Language interpreter that allows users to develop applications for the smart card using the ECMA .NET standard. Applications can be developed in any .NET compliant language that can be compiled to IL.

3.4.1.3 Runtime Environment

A typical runtime environment consists of:

- An Interpreter; which executes the applications loaded on the smartcard.
- Add on Libraries; that in our case are the subset of .NET framework base class libraries provided by the card. JavaCard provides similar class library that includes a strict subset of Java API.

---

[3] http://www.globalplatform.org/

3.4.1.4 Loader

A loader is responsible for safe loading and unloading of applications after post issuance. Following are the basic responsibilities of a typical loader.

- To verify that the code being loaded to the card is safe to run.
- To verify that the assembly being loaded to the card is digitally signed.
- To ensure that all types (classes / objects etc) used by the assembly are already present on the card (either provided by the framework or programmed separately in another application).
- And finally it is also responsible for clean unloading of the applications.

3.4.2 The .NET Smart Card

The Gemalto .NET Card is a post-issuance programmable smart card. It is based on technology that is a subset of ECMA standards[4] (language, CLR, framework) for .NET. The differences between standard .NET Compact framework and the .NET framework provided by the smart card are documented in 0.

Figure 3-1 Gemalto. NET Card V2

As part of .NET solutions Gemalto also provides the Smart Enterprise Guardian which is a smart card based secure storage device. For some of the application scenarios it is more convenient to have a USB based security token as most common PCs/ laptops are equipped with a USB port already but for normal smart cards a separate reader is required.

[4] Standard ECMA-335 Common Language Infrastructure (CLI) 4th edition (June 2006) http://www.ecma-international.org/publications/standards/Ecma-335.htm

Figure 3-2. The Gemalto Smart Enterprise Guardian

Diagrammed below is an architecture component description of the .NET SmartCard. The Smart card runs an operating system on top of the smart card hardware chip (details of the chip and card specifications can be found in Appendix B). On top of the OS, the card provides both a .NET environment libraries and native execution environments for application usage.

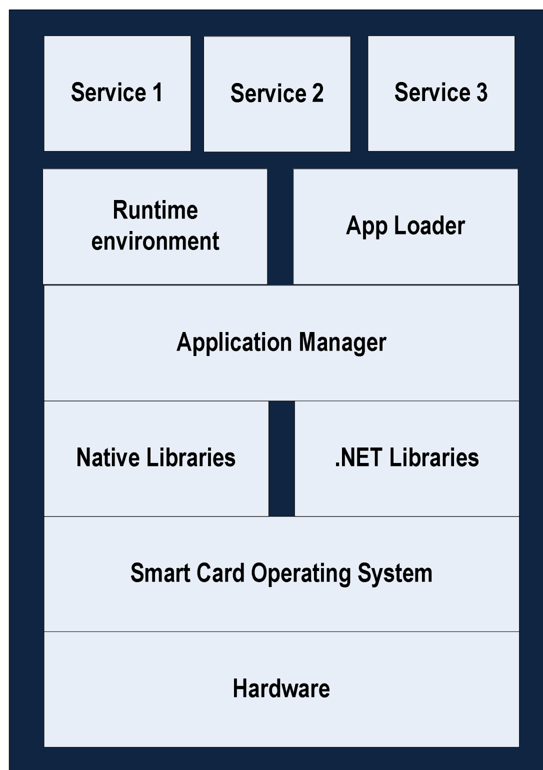


Figure 3-3 Smart Card .NET Architecture

3.4.2.1 The Application Lifecycle

A .NET Card application is managed by the common language runtime (CLR) throughout its lifecycle, beginning when it is converted to a binary format that can be loaded

onto the card. This section briefly describes the application lifecycle in a .NET Card environment.

v. Loading

A .NET Card application can be created using any supported .NET programming language (for example, C# VisualBasic.NET, C++.NET etc). After the development of the code, that code is compiled (using the .NET Compact Framework provided by the Card). The compiled code in .NET environment is called Microsoft Intermediary Language (MSIL) code. The compiler produces a binary which is called an assembly. This assembly is not native code and hence is portable across any platform that provides an implementation of standard .NET framework. Hence the application only needs to be compiled once and can run anywhere.

After being compiled the application needs to be loaded in the SmartCard. The loader microcode coverts this binary from MSIL to card resident binary format which is a much smaller code (almost $1/4^{th}$) comparing to the normal .NET assembly size on the file system. Before loading, it also needs to be strong-named signed otherwise the loader throws an exception.

vi. Installation

The main entry point for all .NET based card applications is the Main function with a declaration like *public static int Main.* This method is executed for application-specific installation after the assembly is loaded on the card. The application also registers the remote types within the .NET Framework in the card to allow remote clients to call methods on the installed application through .NET Remoting. The application is exposed to the outside world in the form of a URI.

vii. Execution

The card based applications follow a client / server model. The application in the card acts as a server and the host application talking to the card acts as a client. The lifecycle of a server application in the card is infinite as the applications do not terminate when the card is powered off or removed from the terminal / reader. The application state does not change from its previous state in either in case of loss of power or even card reset. Hence transactions are implemented to secure persistently all critical data and this need to be taken care of during the development of the application.

viii. Termination

The card application only stops running when a service uri is unregistered. When a service is unregistered the running instance is deleted, the memory is garbage collected and further application cleanup is done as required.

ix. Unloading

After a service has been terminated, the binary containing that service can be removed from the card. A loaded assembly that is still exposing a service cannot be unloaded. The service must be terminated first.

3.4.2.2 The Common Language Runtime and its benefits

.NET Card applications run as managed code within the .NET Smart Card Framework common language runtime (CLR). The common language runtime executes using a CPU-neutral instruction format. A .NET application running under a managed CLR has a number of benefits that provide ease of implementation, separation of security functionality and application isolation capabilities along with numerous security offerings of the .NET framework that can be utilized directly within the applications. This helps us greatly in providing a secure execution environment for applications that are coupled with the TPM to provide enhanced services.

i. **Application lifecycle management**

Manages execution of the code throughout its lifecycle.

ii. **Application domain management**

Supports multiple applications running simultaneously with safety and integrity assurances on a single .NET Card. The security of the application domains ensures that data in one application domain cannot directly reference data in another domain.

iii. **Garbage collection**

Eliminates the burden from the programmers to explicitly free memory when an application no longer needs an object. This ensures there are no memory leaks in an application.

iv. **Remoting management**

Provides an integrated foundation for secure communications between applications using a subset of the .NET remoting architecture.

v. **Exception handling**

Provides standard exception handling with a subset of exception handling classes implemented in the .NET Compact Framework.

vi. **Evidence-based security**

Ensures the integrity and authenticity of .NET Card assemblies during load to the card and during execution of the loaded applications.

vii. **Transaction management**

Ensures the integrity of data and applications in the Card, despite frequent and sometimes unpredictable physical removal of the card from the system or terminal with which it is communicating.

viii. **Code access security**

Very similar to data security; that is, a public key token is required for an assembly to access a dependent library. To enable a library to be shared with another assembly, the corresponding public key token must be added as an attribute. Security policy is determined by the Access Manager.

3.5 SUMMARY

In this chapter we described an overview of SmartCards, how they differ from conventional computing platforms, their different types of standards and architectures, how they are being used in the industry and what special characteristics of them became the motivation for choosing them as an extended platform for the SmartCard and TPM coupling process. These included their portability, processing and storage capabilities, extensive cryptographic support, tamper resistant hardware and protected storage and finally and most importantly secure program execution capabilities.

We discussed some of the application frameworks with special emphasis on the .NET platform available for SmartCards. The different layers of software were described with the responsibilities of each. These included the Operating System, Applications, Runtime environment, loader and unloader microcode provided by the card.

Later in the chapter we described the .NET SmartCard architecture with its application lifecycle, how a typical application management takes place from start to end which includes the loading, installation, execution, termination and unloading phases. We discussed the post issuance update and re-programming capabilities and how different industries utilize them for card based application management.

Finally a few points were mentioned over the benefits of using a .NET based application that would be executed by a Common Language Runtime environment. A .NET application running under a managed CLR has a number of benefits that provide ease of implementation, separation of security functionality and application isolation capabilities along with numerous security offerings of the .NET framework that can be utilized directly within the applications. This helps us greatly in providing a secure execution environment for applications that are coupled with the TPM to provide enhanced services.

The smart card capabilities hence provide us with an attractive platform which has a number of similar capabilities like a TPM, and some different characteristics and capabilities that makes it different from conventional computing platforms. The secure execution, storage and process isolation capabilities provided by a general programming platform like .NET makes it an ideal choice for using it to provide or extend secure execution environments. In the next chapter we will discuss how we utilize a SmartCard for extending secure execution environments of a platform based security device, i.e. the Trusted Platform Module.

4.0 TPM & SMARTCARD COUPLING FOR ENHANCED SECURITY SERVICES

4.1 INTRODUCTION:

Trusted Platform Modules (TPMs) are secure cryptographic processors built into trusted platforms. Combined with the Core and Dynamic Root of Trust (CRTM & DRTM), they provide a rich and powerful secure environment for platform security. The CRTM and DRTM are responsible for reliably informing the TPM of the platform state and software that is running by taking cryptographic measurements. Hence the TPM (acting as a Root of Trust for Reporting RTR and Root of Trust for Storage RTS) can provide security services that depend on the identity and state of the reported software and make decisions based on the platform state. Some of the most common TPM services that are used in this process include:

Attestation: Reliable reporting of the platform state to a remote challenger. The integrity metrics are measured and stored in Platform Configuration Registers (PCRs) and a signature is created with an Attestation Identity Key (AIK).

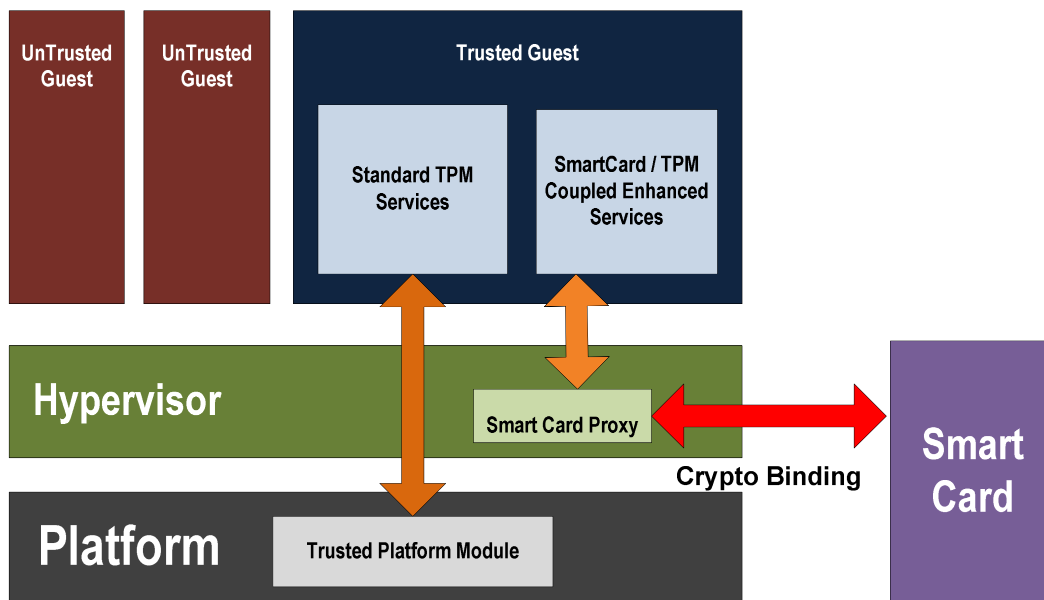
Sealing: Protected Storage/ Encryption of data that ensures release / decryption only to authorized software when it is in a particular configuration and state.

Thus the TPM bootstraps a rich and powerful execution environment running on the main CPU from the small set of functions that it provides. However even though the TPM provides a lot of cryptographic capabilities and tamper resistance, it is not meant to perform general purpose program execution. The current mass market Operating Systems, Hypervisors and general purpose applications only use the TPM services for platform integrity measurements, code measurements and data protection. The applications still run on the mainstream processor executing all code and the secrets held in primary or secondary storage of memory. For many security sensitive applications this normal code execution environment (main-CPU, memory etc) is much less secure (to both hardware and software attacks) than that offered by the TPM, so in gaining flexibility much security is lost. This problem is evident from some of the recent attacks [**11**] on applications utilizing TPMs such as Bitlocker.

One way to avoid this tradeoff is to build a secure execution environment inside the TPM or propose new hardware and instruction set models for general purpose secure execution for e.g. [**12**]. But platforms like these do not exist, would take a lot of time to be built and be available for general usage and most importantly are not envisaged in the near future. There are also some other studies and implementations of providing high assurance secure environments. *AEGIS* [**13**] describes a single-chip architecture for a secure processor which can be used to build computing systems secure against both physical and software attacks. *Terra* [**14**] uses a trusted virtual machine monitor (TVMM) that partitions a tamper-resistant hardware platform into multiple, isolated virtual machines. *Thin Clean Client* [**15**] by IBM Research proposes a modified minimum Linux distribution with trusted computing technology for a secure environment. *Trusted Execution Module* [**16**] puts forward a high level specification for a chip that can execute user supplied procedures in a trusted environment. *Flickr* [**17**] uses the new general purpose

hardware extension for secure OS loading, late launch and attestation. It provides an infrastructure for executing security sensitive code in complete isolation while trusting as few as 250 lines of additional code but requiring special hardware.

In this project we propose and implement a different architecture. Instead of making expensive changes to the hardware and adding complex functionality to the TPM specifications which would not only break the existing applications, would also increase the effort and cost of writing new ones. We evaluate the extent to which other platforms provide similar degree of hardware tamper resistance and secure execution and couple them with a TPM to provide extended TPM services not possible with the current specifications either of TPM or smart card alone. For this study we use multi-application programmable .NET Cards that provide adequate tamper resistance, a programmable environment with application isolation and crypto blocks for building confidentiality and integrity services. This enables greater levels of protection for information stored, processed and exchanged across different systems.



A TPM / Smart Card Co-operative Model

Figure 4-1 A Smart Card TPM Cooperative Model

Figure 4-1 A Smart Card TPM Cooperative Model shows an example schematic of the smartcard-enhanced TPM design that is implemented in this project. The hypervisor is shown for definiteness, but similar design has been developed and tested when no hypervisor is present. An authenticated cryptographic tunnel is established between the smartcard and the TCB. Enhanced smartcard services are exposed through this tunnel. The smartcard proxy exposes a number of enhanced TPM/SC coupled services to the outside world along with the standard TPM Services.

We first discuss the challenges we face in the coupling process and the platform differences between the TPM and a Smart Card. Then we describe our implementation of a

secure TPM / Smart Card cryptographic binding. We move forward to describe our implementations of some of the enhanced TPM / Smart Card coupled services that were not possible with a TPM or Smart Card alone. With these enhanced TPM services we implement some applications that change the way conventional TPM or Smart Card applications are perceived. Finally we shed some light on potential future applications and future work.

4.2 A TPM ARCHITECTURE FOR SECURE EXECUTION

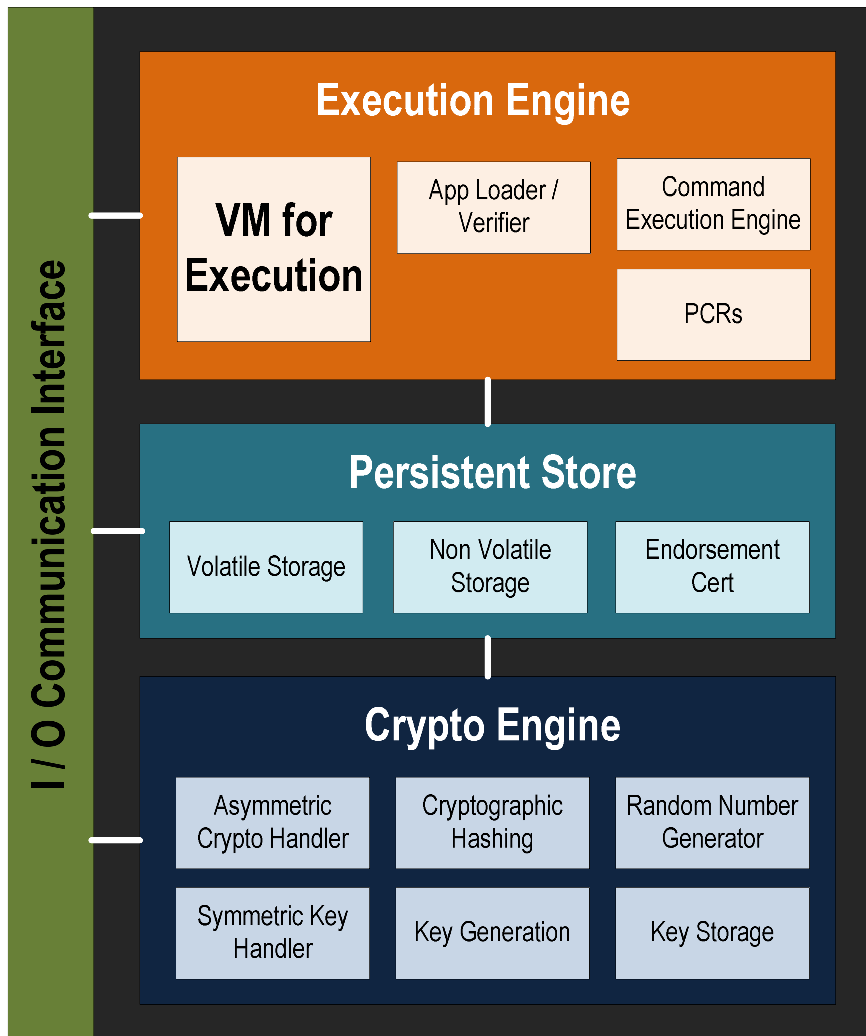
A TPM provides many services for platform attestation, security and data protection. A typical TPMs architecture is described in 2.4 but as described earlier it does not provide any secure execution environment on its own which leads to tons of vulnerabilities and attacks on the operating system and applications. Some of the noted attacks on the solutions and services utilizing the conventional TPM are [**11**] and [**18**]. To counter these attacks, the secrets, encryption keys and sensitive pieces of code has to be isolated within trusted boundaries providing confidentiality and integrity services.

We first discuss and propose how an ideal prototype of an extended TPM that provides code execution isolation and guarantees of confidentiality would look like. This model would have a trusted execution architecture integrated within the TPM. An architecture like this though does minimize the total Trusted Computing Base (TCB), but also introduces its own challenges including but not limited to building code execution blocks specifications, performance, and support for different platforms. Similar work has been proposed and developed in [**16**]. But platforms like these do not exist, would take a lot of time to be built and be available for general usage and most importantly are not envisaged in the near future.

4.2.1 A hypothetical architecture of the TPM-Internal Execution Environment

Assume an execution environment for applets or code modules inside the TPM. The TPM provides a Measured Trusted Virtual Machine for providing isolated trusted environments for code execution. Loader microcode in the modified TPM loads a reserved PCR register with the hash of the class / program/ AppDomain being executed. The assembly is also granted a special "locality." The extension program can then authenticate itself (quote), store private data (seal), and any other TPM operation. The extension program can also provide additional services to external callers.

This model provides a nice open extensibility model because the extension program is granted few additional privileges beyond those granted to platform macrocode (the extension program is treated in large measure as an external program – just with a different locality and reserved PCR). On the other hand, because the extension programs cannot access genuine TPM private data like the SRK or EK private keys, there are limits to the functionality of the extension programs. We will explore whether additional privileges are necessary and can be granted.



A TPM with Trusted Execution

Figure 4-2 A TPM Architecture for Trusted Execution

From the next section we discuss the implementation of our smartcard and TPM co-operative model which uses general purpose commodity hardware.

In Chapters 2 and 3 we explored the differences of Trusted Platforms and smart cards in detail, the different computing and servicing categories they fit in, the different TPM-OR-SmartCard applications they provide and how the businesses are utilizing them with their current state of technology. This Chapter is to explore an alternative architecture: To utilize the best of both worlds in smart cards and TPMs, and explore some of the new services and applications that can be offered by a TPM-And-SmartCard cooperative model.

4.3 SMART CARD CAPABILITY REQUIREMENTS

To participate fully in a TPM and Smart Card co-operative model, a smart card at the minimum should provide at least the following

i. Random Number Generation
ii. Symmetric Cryptography (AES / Rijndael)
iii. Asymmetric Cryptography (RSA up to 2048-bit)
iv. Hash algorithms (SHA-1 and HMAC)
v. Implementations of XOR over a piece of data

Most of the programmable (Java and .NET Card)s available as of this writing, have most of these capabilities and provide native true random number generation, basic RSA and Symmetric algorithms including DES, 3DES and AES. There are differences and limitations however on the different padding schemes supported for RSA (PKCS/ OAEP) and native HMAC capabilities. The choice on what the smart card is capable of doing entirely depends on what applications and services would the smart card be providing. The building blocks not provided by the card natively can be programmed and loaded as serviceable objects, JavaCard applets or onCard assemblies in .NET Cards.

4.3.1 Platform Differences between TPM and the Smart Card

There are some fundamental differences between the services that a TPM provides and the solutions that a smart card addresses. First a TPM is bound to a platform but a smart card is “roaming”, hence a smart card is more vulnerable to attack. Second it does not share some of the interesting properties of the TPM. A TPM is reset when the platform is reset, but the application lifecycle for a .NET application is infinite. Furthermore the TPM acts as the Root of Trust for Measurement for a particular platform but smart card has no such capabilities. Also the ownership administration of the TPM lies with the platform owner but for most of the smart cards (for e.g. SIM Cards, Identity cards etc) it lies with the issuing authority.

Our major interest areas for SmartCards differences are its functionality for providing the execution environments for code modules and cryptographic primitives. The specifications of a TPM for all operations (e.g. Sealing / Binding / Wrapping / Certifying) keys have strict requirements of how the data is formatted, serialized, how the keys are presented what cryptographic algorithms are used, what key types are used, what encryption / decryption methods are used, what signature schemes are supported and how is the message encoded. So a Smart Card needs to participate with the required crypto primitives of the TCG Model else the supporting operations need to be taken care of manually either inside the card as a separate code module or outside the card in a proxy application.

4.3.2 Coupling Challenges

These fundamental differences and assumptions of un-trusted hardware, software and channels introduce some interesting challenges of how to bind the two architectures together.

Hence we need to build strict authentication, confidentiality and integrity services if we want to offer coupled services. We use cryptographic primitives provided by the smart card and the TPM to achieve this process which is described later. For a smart card to verify the platform state we implement a service that uses the hosts TPM_Quote primitive, hence verifying the platform state by generating an attestation, and the TPM identifies a card by loading its identity key under its key hierarchy.

4.4 EXPERIMENTAL PLATFORM

Our experimental platform consisted of:

i. Gemalto .NET Cards (v2.1.161) implementing version 2 of .NET Compact Framework. It supports SHA1, HMAC, RSA up to 2048bits, DES, 3DES and Rijndael for cryptographic services and a subset of .NET Compact Framework v2.
ii. The TPM is Infineon 1.2 built into Dell Optiplex 755.
iii. We use the Windows Vista TPM Base Services Library (TBS)[5] for all communications to the TPM.

4.5 SECURE TPM AND SMARTCARD BINDING

The first problem that we need to solve is how the TPM and the Smart Card identify or recognize each other. This problem goes beyond just simple identification as we need to secure end-to-end our smart card and TPM communication for confidentiality and integrity services. To strengthen binding between the smart cards and the TPM, we create session based symmetric keys which are securely transferred by using Asymmetric keys of the smart cards and the TPM. There are 2 parts of the application: the OnCard Service and the OffCard Client Proxy that acts as an interface with the TPM and the SmartCard communication. The process for secure binding goes as follows:

- A TPM generates an Attestation Identity Key (AIK) which we also use as the Hosts Identity Key.
- The Card Generates an RSA Key Pair which we use for its identity.
- The public key of AIK generated in step 1 are exported and loaded securely[6] in the card.
- The public key of Cards Identity key generated in step 2 is exported and loaded in the TPM as a Loadable Key.
- We use these Asymmetric keys to create a secure tunnel between the Host and the Card through extending the .NET remoting architecture (described in the next section) to communicate to the card.
- One time session keys are generated and transferred securely across the host and the card.

This is an out of bound marrying step for TPM and smart card coupling but once married both platforms can authenticate, identify and co-operate with each other to provide enhanced coupled services not possible with each platform alone. In an enterprise this out of bound marrying step can be performed by the IT administrators while issuing Identity smart cards or

[5] http://msdn.microsoft.com/en-us/library/aa446796(VS.85).aspx
[6] The Gemalto card offers token and role based security for files stored in the persistent storage.

network operators during production of SIM cards to hold public keys of the TPM(s) they are bound to.

4.5.1 Extended Crypto Remoting for End-to-End Encryption

The .NET Card uses an extended .NET Remoting architecture to allow a program executing on the Host (PC) to communicate with a process running in the smart card and also allows a program running in one application domain to access code or data in a process running in another appDomain within the card. The conventional way of talking to a smart card is through the APDUs (Application Protocol Data Units). However as the industry around the smart card hardware and software is maturing, the smart card manufacturers are adopting the new language and framework constructs to shield the programming complexity and utilize the newer frameworks offerings more to write optimized, secure applications faster and make them more manageable. To solve the problem of distributed communication and remote object access there are a number of technologies that have been constantly updated with richer, newer, more stable technologies. Java for instance solves this problem through RMI [**19**] (Remote Method Invocation). .NET introduced an architecture called Remoting (described in detail in 4.5.1.1) which is superseded by Windows Communication Foundation (WCF) in .NET 3.0.

4.5.1.1 .NET Remoting Overview

Remoting [**20**] is the Microsoft's approach for Inter Process Communication (IPC) in the .NET Framework. The real strength of the remoting framework, resides in its ability to enable communication between objects in different application domains or processes using different transportation protocols, serialization formats, object lifetime schemes etc. Contrary to some other technologies for e.g. Web Services, remoting enables us to work with stateful objects. Remoting is superseded by a technology called Windows Communication Foundation (WCF) in .NET 3.0. Our implementation restricts us to use .NET 2.0 as the .NET Card we are using at the time of this writing only supports .NET 2. We use the remoting services provided by the .NET Smart card framework for cross channel and platform communication.NET remoting can be compared to similar technologies such as Java Remote Method Invocation (RMI), CORBA etc.

The .NET Smart Card Framework extends standard .NET remoting and allows a program executing on a PC to communicate with a process running on a Gemalto .NET Card, and also allows a program running in one application domain to access code or data in a process running in another application domain within the Gemalto .NET Card.

Basic Remoting in the .NET Smart Card Framework:

The .NET Remoting Architecture consists of 5 core object types.

i. **Proxies:** These objects impersonate as remote objects and forward calls.
ii. **Messages:** contain the necessary information to execute a remote method.
iii. **Message sinks:** These allow custom processing of messages during a remote invocation.

iv. **Formatters:** These work like message sinks as well but are also capable of serializing a message to a transfer format like for e.g. SOAP.
v. **Transport channels:** These also work like message sinks, but they also transfer the serialized message to a remote process, for example, via HTTP or TCP Channels.

Remoting works by having a server application (in our case the On Card application works as a server) expose an object to the external world by registering the object as a service either through the `RemotingConfiguration.RegisterWellKnownServiceType()` method or through the `RemotingServices.Marshal()` method. In order for an object to be registered as a service in .NET, that object must inherit from one of the base marshalling classes. Although the .NET framework supports marshalling either by reference or by value[7], the .NET Smart Card Framework supports only marshalling by reference (i.e. a class extending `System.MarshalByRefObject`). For detailed references to .NET remoting please see [**20**]. After the server has registered the object, the object becomes available to clients that connect to the server. When the client connects to the server, it creates a local proxy of the server object. When the client wants to call a method on the remote object, the proxy object passes the method call to the system remoting mechanism, which is responsible for marshalling (described in the next section) the parameters and return value of the method. The current implementation of the .NET Smart Card Framework[8] does not support the marshalling of classes. However, it does support the marshalling of all value types (including structs) and supports both out and ref parameters. Types that can be marshalled include the basic value types (byte, short, char, int, long, string, etc), structs, arrays of basic types, and MemoryStreams[9].

---

[7] Marshalling objects by value means to serialize their state including all objects referenced to some persistent form from which they can be deserialized in a different context.
[8] Gemalto .NET SDK v.2.1.161.8583
[9] MemoryStream class creates streams that have memory as a backing store instead of a disk or a network connection.

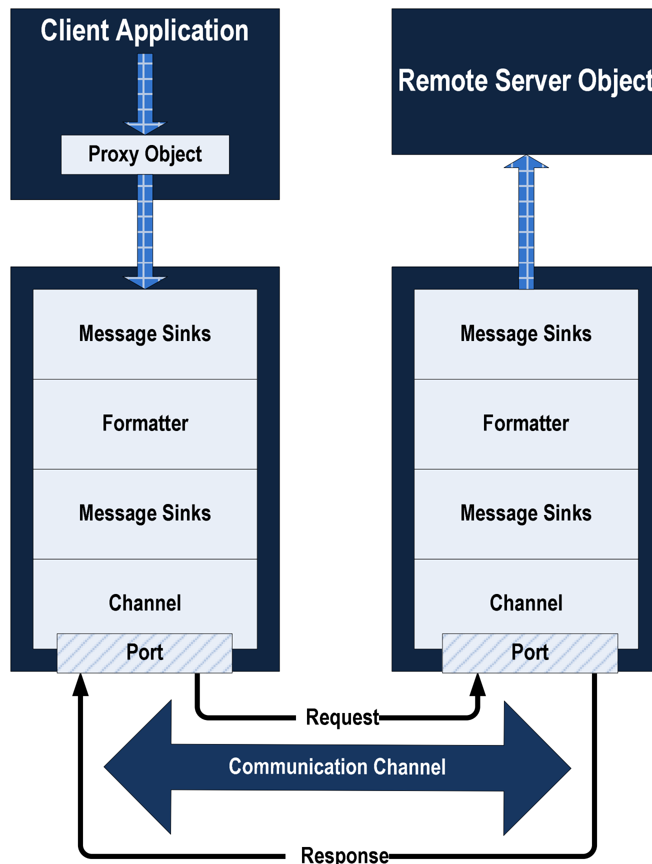


Figure 4-3 Simplified .NET Remoting Architecture.

The mechanism by which a client connects to the server is completely isolated from the marshalled object. Conventional .NET remoting applications generally use either TCP or HTTP as the transport protocol for applications. The .NET Smart Card Framework uses ISO 7816-4 as a transportation protocol for communication. However, because the transportation protocol is isolated from the service object, we don't have to worry about the actual protocol of ISO 7816-4 communication.

All communication between the client and server takes place through a channel. A channel is just simply a series of sinks through which the remoting call is made. That could include custom formatters, serializers etc. The .NET Smart Card Framework defines a new type of channel known as an APDUChannel. This is referenced on the server (card) side through the `APDUServerChannel` class and on the client (PC) side through the `APDUClientChannel` class. The APDUChannel is responsible for encoding method calls to a binary format and transporting them from the client to the server using the ISO 7816-4 protocol.

Channels and Ports

When a remote-able class is created in the .NET framework, it needs to have a definition of a channel which has to be registered with a port number to make an associative pair. (This is

something similar to TCP/IP where a server listens on a specific port). This channel/port combination registered in the .NET infrastructure listens on that port for messages intended for that particular channel. When a message arrives, the framework routes it to the correct server object. See Figure 4-3 Simplified .NET Remoting Architecture.

Server Code

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using SmartCard.Runtime.Remoting.Channels.APDU;

namespace SCInterfaceManagerServer
{
    /// <summary>
        /// Server class to register the service and Secure Session Establishement Channel.
        /// </summary>
        public class OnCardServer
        {
                /// <summary>
        /// Registers the on Card service
        /// </summary>
        /// <returns></returns>
        public static int Main()
        {
            IServerChannelSinkProvider newProvider = new SessionEstablisherSinkProvider(null, null);
            newProvider.Next = new APDUServerFormatterSinkProvider();

            APDUServerChannel channel =
                        new APDUServerChannel (newProvider , Constants.serviceConnectionPortNormal );

            // Register the channel the server will be listening to.
            ChannelServices.RegisterChannel(channel);

            // Register this application as a server
            RemotingConfiguration.RegisterWellKnownServiceType( typeof(
                                    SecureSessionEstablishmentService), Constants.ServiceName,
                                    WellKnownObjectMode.Singleton);

            return 0;
        }
    }
}
```

In the .NET Framework, the client code (in our case the host proxy application) also creates a channel associated with a specific port, and then uses the `Activator.GetObject` method to obtain a reference to the remote object. A remote object is identified with the (Uniform Resource Locator) URL of the computer on which it is located, the name of the remote class, and a (Uniform Resource Identifier) URI that was assign.

The APDUChannel supports URL's of the format:

"apdu://<name of the smart card reader>:<the port on which the service is registered>/<the name of the service>"

For example: "apdu://Gemalto Reflex USB v2:2222/CardService"

In addition to explicitly naming the reader to connect to, you can also use the reserved names "promptDialog" and "selfDiscover". The promptDialog mechanism will display a dialog box and allow the user to select which reader to use. The selfDiscover mechanism attempts to find the requested service by attempting to connect to any .NET smart cards attached to the machine.

Client Code

```
APDUClientChannel channel = new APDUClientChannel("clientChannel", null);
ChannelServices.RegisterChannel(channel);

// Connect to the service on the clear channel.
ChannelCardInsertionStateChangedEventHandler eventHandler = OnCardInsertionRemoval;
channel.RegisterCardInsertionStateChangedEventHandler(eventHandler);


// Connect to the service on the clear channel.
string SmartCard_OnCardService = "apdu://" + readername + ":" + Constants.serviceConnectionPortNormal +
"/" + Constants.ServiceName;

SecureSessionEstablishmentService sessionEstablishmentService = (SecureSessionEstablishmentService)
Activator.GetObject(typeof (SecureSessionEstablishmentService), SmartCard_OnCardService, "clientChannel");
```

4.5.2 Securing Remoting communication between the TPM and the SmartCard

Conventional .NET remoting applications either use TCP or HTTP as the transport protocol for applications. The .NET Smart Card Framework uses ISO 7816-4 as a transportation protocol for communication. But since the transportation protocol is isolated from the service object, we don't have to worry about the actual protocol of ISO 7816-4 communication. .NET remoting does not offer any security services on its own. Some of the fundamental problems are lack of confidentiality (the remoting traffic is sniff able until encrypted and the methods exposed to the outside world through the proxy can be called by anyone). However, remoting does offer an open interface architecture so any of these services can be implemented by extending the remoting architecture through custom sinks and sink providers. This not only enables the use of encryption over custom sinks, it also gives the ability to use different transport methods. Other reasons for using custom sinks are to use compression or custom formatting methods. For details on extending the remoting architecture see [**20**].

We use the session based symmetric keys to encrypt and decrypt all communication between the TPM and the Smart Card. The encryption and decryption of exchanges between service and client are delegated to the pluggable custom sinks instead of handling it within the application itself. This makes code in service smaller, portable and independent of cryptographic algorithm used. The Remoting components can also be hosted in IIS to utilize windows authentication protocols, SSL/TLS for transport level encryption easing in deployments in large enterprises using active directory services. See 4.5.4.

We start the process of securing the path between the smart card and the TPM by generating an Asymmetric key pair for each entity. For the TPM, we generate an Attestation Identity Key (AIK), and we do an Asymmetric RSA Key generation inside the card. The public

key certificate of the card is exported and stored securely[10]. The public part of the AIK is also exported and stored securely inside the card[11]. For all practical purposes and large scale distributed environment an enhanced key distribution mechanism and protocols (for e.g. Diffie Hellman / Internet Key exchange etc) should be used. The choice of the mechanism would depend and vary from different security requirements. It is generally accepted a good practice to generate and store keys within a protective hardware[**21**]

Once the TPM and the smart card(s) are loaded with a trusted copy of each other's public key we can create a secure session between them. We achieve this by generating a random symmetric key that would only be used per session[12]. The symmetric key can be generated either by the client or the smartcard depending upon the application scenario and usage, since both the TPM and the Smart Card are able to generate True random numbers that could be used to generate initialization vectors and key streams to be used for symmetric encryption. This symmetric key is called the session key.

The simplified protocol goes as follows:

[10] There are a number of methods that can be used including encrypting it under the SRK. Storing the key inside the TPM or just sign it with the generated AIK so that a trusted copy can be fetched later.

[11] Data files in the Gemalto .NET card are protected using a public key token system that is very similar to that used by applications. For details see 3.3.6

[12] Session in the smart card terminology means the communication between two resets.

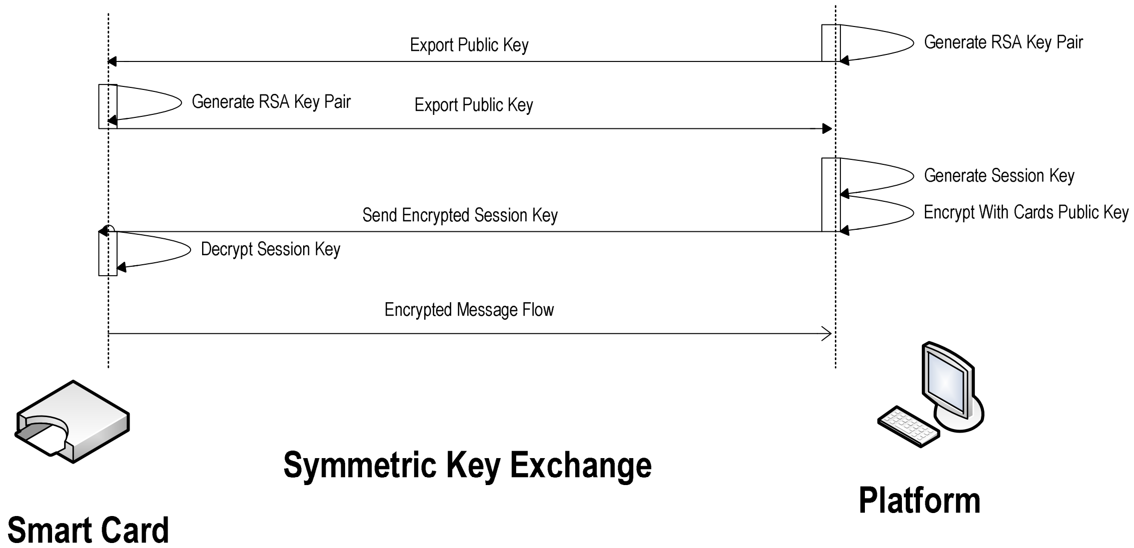


Figure 4-4 Session based symmetric key exchange.

We omit the details of content of messages to show a simplified version of authenticated key exchange. The platform user can authenticate to the card by authentication mechanism implemented in the card. In our case we use the default Access Manager implemented in the .NET Card that offers role based security and Pin Authentication. Similarly the platform needs to be authenticated as well. In our case we generate a platform attestation based on the TPM_Quote primitive on a nonce generated by the card to verify that we are talking to the correct TPM. The message flow in Figure 4-4 can be noted to be quite similar to that used in SSL and gives mutual authentication.

4.5.2.1 Custom Sinks (Extending .NET Remoting to support encryption)

When implementing custom sinks, we have to take care that a particular sink behaves properly as part of chin of sinks. Depending upon the application there can be a client and server, or client or server sink. For encryption, naturally there has to be a sink pair. The server and client sinks differ slightly in implementation as described in detail below.

The Session Sink on the client side implements the IClientChannelSink

```
public class SessionSink: BaseChannelSinkWithProperties,IClientChannelSink
```

keeps an attribute of next sink in chain and the session key

```
/// <summary>
/// Next Sink in Chain
/// </summary>
private IClientChannelSink nextSink;
/// <summary>
/// The session key to encrypt the communication
/// </summary>
private byte[] sessionKey;

public SessionSink(IClientChannelSink next,byte[] sessionKey)
{
    nextSink = next;
    this.sessionKey = sessionKey;
}
```

The key method of the implementation is the ProcessMessage method where the outbound processing is performed before passing the message to the next sink in the chain. When the message returns from the next sink, we perform inbound processing

```
/// <summary>
/// Requests message processing from the current sink.
/// </summary>
/// <param name="msg">The message to process.</param>
/// <param name="requestHeaders">The headers to add to the outgoing message heading to the server.</param>
/// <param name="requestStream">The stream headed to the transport sink.</param>
// <param name="responseHeaders">When this method returns, contains a <see
cref="T:System.Runtime.Remoting.Channels.ITransportHeaders"/> interface that holds the headers that the
server returned. This parameter is passed uninitialized.</param>
/// <param name="responseStream">When this method returns, contains a <see cref="T:System.IO.Stream"/>
coming back from the transport sink. This parameter is passed uninitialized.</param>
/// <exception cref="T:System.Security.SecurityException">The immediate caller does not have
infrastructure permission. </exception>
[SecurityPermission(SecurityAction.LinkDemand, Infrastructure=true)]
public void ProcessMessage(IMessage msg, ITransportHeaders requestHeaders, Stream requestStream,
                                out ITransportHeaders responseHeaders, out Stream responseStream)
{
    requestStream = SymmetricCrypto.ProcessOutboundStream(requestStream,"Rijndael",sessionKey);
    // forward the call to the next sink
    nextSink.ProcessMessage(msg, requestHeaders, requestStream, out responseHeaders, out responseStream);
    responseStream = SymmetricCrypto.ProcessInboundStream(responseStream,"Rijndael",sessionKey);

}
```

Note that we pass a hardcoded string "Rijndael". The argument to a SymmetricAlgorithm could be any depending upon the requirements and availability at both ends of the sink.

We also have to implement System.Runtime.Remoting.Channels.IClientChannelSinkProvider[13] which is responsible for creating the sink object and for calling other providers to create other sinks in the chain.

```
public class SessionSinkProvider : IClientChannelSinkProvider
```

The class holds the attribute for the next provider in sink chain and the symmetric key

```
/// <summary>
```

[13] http://msdn.microsoft.com/en-us/library/system.runtime.remoting.channels.iclientchannelsinkprovider(VS.80).aspx

```
/// Next Sink in Chain
/// </summary>
private IClientChannelSinkProvider nextProvider;
/// <summary>
/// The symmetric key to encrypt the sink
/// </summary>
private byte[] sessionKey;

[SecurityPermission(SecurityAction.LinkDemand, Infrastructure = true)]
public IClientChannelSink CreateSink(IChannelSender channel, string url, object remoteChannelData)
{
    // create other sinks in the chain
    IClientChannelSink next = nextProvider.CreateSink(channel, url, remoteChannelData);

    // put our sink on top of the chain and return it
    return new SessionSink(next, sessionKey);
}
```

The session Sink on the Server Side implements the IServerChannelSink

```
public class SessionSink : IServerChannelSink
```

which has the same attributes as the Client, i.e. the next message in chain and the symmetric key

```
/// <summary>
/// Next Sink in Chain
/// </summary>
private IServerChannelSink nextSink;
/// <summary>
/// The symmetric key to encrypt the sink
/// </summary>
private byte[] sessionKey;
```

The key method of the implementation is the ProcessMessage method, which is responsible for implementing whatever transformations the sink is responsible for. A server sink may perform either inbound transformations, outbound transformations, or both. It is critical that the server sink also call the next sink in the sink chain between processing its inbound and outbound data.

```
public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack, IMessage requestMsg,
                ITransportHeaders requestHeaders, Stream requestStream, out IMessage responseMsg,
                out ITransportHeaders responseHeaders,  out Stream responseStream)
{
   Logger.Log(Logger.LogLevel.Info, "Process Message Entry");
   // decrypt the inbound messsage
   requestStream = SymmetricCrypto.ProcessInboundStream(requestStream, "Rijndael", sessionKey);

   // mark that we are on coming from sessionestalishersink
   SecureSessionEstablishmentService.onChannelEstablishment = false;

   ServerProcessing srvProc = nextSink.ProcessMessage(sinkStack,requestMsg, requestHeaders, requestStream,
                                             out responseMsg, out responseHeaders, out responseStream);

   // encrypt the outbound message
   responseStream = SymmetricCrypto.ProcessOutboundStream(responseStream, "Rijndael", sessionKey);

   Logger.Log(Logger.LogLevel.Info, "Process Message Exit");
   // returning status information
   return srvProc;
}
```

We also have to implement System.Runtime.Remoting.Channels.IServerChannelSinkProvider[14] which is responsible for creating the sink object and for calling other providers to create other sinks in the chain.

```
    public class SessionSinkProvider : IServerChannelSinkProvider
```

The SessionSinkProvider contains an attribute for the next provider in chain

```
    private IServerChannelSinkProvider nextProvider;

/// <summary>
/// Creates a sink chain.
/// </summary>
/// <param name="channel">The channel for which to create the channel sink chain. </param>
/// <returns>The first sink of the newly formed channel sink chain, or a null reference, indicating that
this provider will not or cannot provide a connection for this endpoint.</returns>
    public IServerChannelSink CreateSink(IChannelReceiver channel)
        {
            // create other sinks in the chain
            IServerChannelSink next = nextProvider.CreateSink(channel);

            // put our sink on top of the chain and return it
            return new SessionSink(next, sessionKey);
        }
```

The limitations

This Gemalto.NET card supports only the use of MemoryStreams and FileStreams within a custom sink. We cannot use either a CryptoStream or a CustomStream as the basis for sink manipulation. Attempting to use an unsupported stream results in a NotSupportedException.

i. Usage of Custom Sinks

The encryption and decryption of exchanges between service and client are delegated to the custom sink instead of handling it within the application itself. This makes code in service smaller, portable and independent of cryptographic algorithm used.

[14]http://msdn.microsoft.com/en-us/library/system.runtime.remoting.channels.iserverchannelsinkprovider(VS.80).aspx

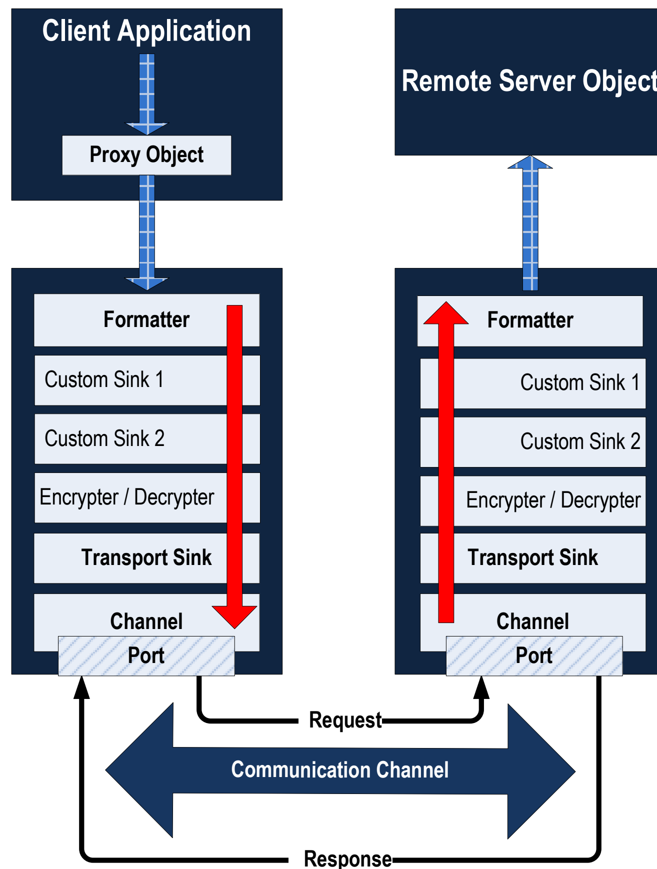


Figure 4-5 .NET Remoting with pluggable custom sinks.

To Use the custom sink we just built, we have to insert it into the sink chain both on the server and on the client.

On the client side, the sink provider is created and placed after the APDUClientFormatterSinkProvider. Then, when we register a channel, we register the APDUClientFormatterSinkProvider as the parameter to the channel we will be using.

```
// Set up a Sink Provider with a SessionSink attached to it using the sessionKey as a parameter for
creating the SessionSink.
Hashtable properties = new Hashtable();
properties["key"] = sessionKey;

IClientChannelSinkProvider provider = new APDUClientFormatterSinkProvider();
provider.Next = new SessionSinkProvider(properties);

// Create and register a new channel using the sink provider that we've just created.
string channelName = "SecureChannel_" + DateTime.Now.Ticks;
APDUClientChannel apduClientChannel = new APDUClientChannel(channelName, provider);
ChannelServices.RegisterChannel(apduClientChannel);
```

On the server side, the sink provider is created and placed after the APDUServerFormatterSinkProvider. Then, we register a channel, we register the APDUServerFormatterSinkProvider as the parameter to the channel we will be using.

The class is inherited from MarshalByRefObject for reason defined in 4.5.1.1 and contains an array of the channels that it has already created.

```
public class SecureSessionEstablishmentService : MarshalByRefObject
internal static ArrayList channels = new ArrayList();


    // Set up the encryption sink properties.
     Hashtable properties    = new Hashtable();
     properties["key"]       = sessionKey;

     IServerChannelSinkProvider newProvider = new SessionSinkProvider(properties, null);
     newProvider.Next = new APDUServerFormatterSinkProvider();

     APDUServerChannel channel = new APDUServerChannel(newProvider,port);

     // Register the channel the server will be listening to.
     ChannelServices.RegisterChannel(channel);

     channels.Add(channel);
```

To achieve the secure communication, the service should have an APDUServerChannel listening at a pre-determined port and should provide methods which should not be invoked over channels using the SecureSessionSink. The problem with .NET remoting architecture is that there is no explicit link between the transport channel and service. The system will not prevent the invocation of any service method on any channel in a given AppDomain. Hence ExecuteOverSecureChannel() can be invoked on the predetermined port which does not have any security or custom encryption sink. To solve this problem, APDUServerChannel should also include a custom sink whose sole purpose is to mark the fact that the remote method is invoked at channel listening at the pre-determined port. This is done by setting a static boolean variable accessible by the service. If a method such as ExecuteOverSecureChannel() is invoked over the pre-determined port (and without being authenticated), the boolean variable is set to true and implementation of ExecuteOverSecureChannel() method expects this flag to be false.

ii. Summary of Steps

Here are mentioned the steps for establishing and communicating over a secure channel

i. The Client Host Proxy creates & registers an APDUClientChannel without any custom sink.
ii. Client Invokes the GetPublicKey() method of the Smart Card service at APDUServerChannel listening on a predetermined port in the card. The remote method returns the public modulus and exponent which is imported as a public key into an RSACryptoServiceProvider[15].
iii. Client generates a random session key of 128 bits (16bytes). (This can be delegated to card as well)

---

[15] http://msdn.microsoft.com/en-us/library/system.security.cryptography.rsacryptoserviceprovider(VS.80).aspx

iv. Client encrypts the session key and the PIN of the OnCard service using the RSACryptoServiceProvider.
v. Client invokes EstablishSecureChannel() method of service at APDUServerChannel listening on the pre-determined port. The arguments passed to this method are the port number (any random) at which new created channel should listen, the encrypted pin and the encrypted session key.
vi. EstablishSecureChannel() method in card decrypts the PIN and session key using the private key. It validates if the PIN passed is correct. If the PIN was correct, a new APDUServerChannel listening at the port number passed in the method call is created and registered with the custom sink called SessionSink.
vii. Client now creates and registers an APDUClientChannel with SessionSink using the session key.
viii. Client invokes ExecuteOverSecureChannel() method of service at APDUServerChannel listening at the new negotiated port. The data sent passes through the SecureSessionSink of APDUClientChannel that was registered in vii and is encrypted with the session key.
ix. SessionSink of APDUServerChannellistening at the new port receives the data and decrypts it with the session key.
x. ExecuteOverSecureChannel () method is invoked. The method checks if invocation is done on secure channel by checking the boolean flag.

xi. This method does all the processing of messages sent by the TPM via Client Proxy through the SessionSink of APDUServerChannel listening at the random port that was negotiated. All communication is encrypted with the session key.
xii. SessionSink of APDUClientChannel receives the returned responses from the card and decrypts it with the session key.

Same steps will be repeated for any client that wants to communicate with the service. The APDUServerChannels with SessionSink are unregistered and destroyed on a reset (end of smartcard session).

In a distributed environment a number of client applications might be communicating with the OnCard Server application, hence the card application needs to have a different communication channel for each client. We achieve this by negotiating a unique port number for the channel to be created between the client and the server. The channel are created and registered dynamically before the secure session and unregistered at the end of each session to free up resources.

Secure Session Establishment Code

```
// Generate a 128 bit session key (can be generated from the card as well using the same service).
byte[] sessionKey = RNG.GenerateRandomBytes(16);

// Get the public key from the card (Which is the RSA Modulus and the Exponent)
byte[] cardPKmod = sessionEstablishmentService.GetPublicKey();
byte[] cardPKexp = sessionEstablishmentService.GetExponent();

// Put the public key from the card into an RSACryptoServiceProvider
RSACryptoServiceProvider rsaProvider = new RSACryptoServiceProvider();

RSAParameters rsaParam = new RSAParameters();
rsaParam.Modulus = cardPKmod;
rsaParam.Exponent = cardPKexp;
```

```
    rsaProvider.ImportParameters(rsaParam);

    // This is the pin that we share with the card
    byte[] pin = Encoding.ASCII.GetBytes(Constants.CardPin);

    // Encrypt the pin and session key using the public key of the card
    byte[] encryptedPin = rsaProvider.Encrypt(pin, false);
    byte[] encryptedSessionKey = rsaProvider.Encrypt(sessionKey, false);

    // Now call the EstablishSecureChannel method of the card using the encrypted PIN and session key.  The
card will set up an encrypted channel using the provided session key.

    try
      {
        sessionEstablishmentService.EstablishSecureChannel(Constants.ServiceConnectionPortSecure,
                                                            encryptedPin, encryptedSessionKey);
      }
    catch (Exception ex)
      {
                Console.WriteLine(ex.Message);
                throw new AuthenticationException("PIN incorrect");
      }
      //All communication from here onwards are on the encrypted channel
       ExecuteOverSecureChannel(readername, sessionKey);
```

4.5.3 Data Marshalling (Data Transformation between the TPM and the SC)

Marshalling is the process where the memory representation on an object is transformed into a format which is suitable for storage and transmission. The reverse process is called Un-marshalling. Marshalling and Un-marshalling are extensively used in Remote Procedure calls and Inter Process Communication, hence data transferred in .NET remoting has to be marshalled before sending and unmarshalled on receiving. Another term serialization is sometimes used synonymously with Marshalling however .NET clearly differentiates between the two. Since the Smart Card is providing a lot of cryptographic services, the methods exposed need to transfer objects on the remoting layer that include cryptographic keys, TPM primitive structures etc. Hence for e.g. if we need to transfer an RSA key from the Card to the server, the RSACryptoServiceProvider object would need to be marshalled into a sequence of bytes (byte array), transferred over a remoting channel and then the byte stream unmarshalled into the RSACryptoServiceProvider object on the client end again. As described in 4.5.1.1 the current implementation of the .NET Smart Card Framework does not support the marshalling of classes. It only support the marshalling of basic primitive value types that include (byte, short, char, int, long, string, etc), structs, arrays of basic types, and MemoryStreams.

There are special challenges in marshalling / unmarshalling objects inside the smart card as the .NET Framework of the card does not support Reflection[16], hence the code has to be written separately to marshall / unmarshall each new type of object.

4.5.4 Securing Remoting Components in a Distributed Environment

The method to strengthen security of remoting described in the earlier section is just one step towards secure building blocks of a TPM-SmartCard cooperative environment. The solution

[16] http://msdn.microsoft.com/en-us/library/f7ykdhsy(VS.80).aspx

proposed does provide confidentiality and to some extent semi-mutual authentication and authorization but is still susceptible to many vulnerabilities, threats and risks. Furthermore in a distributed environment, where clients may not necessarily be within the same domain, there are a lot of challenges in authenticated key distribution, cryptographic binding or end to end tunneling. In this section we explore a more scalable solution for security of our remoting architecture for a distributed environment. In large distributed environments, it would make sense to utilize the authentication and access control mechanisms already set in place in the enterprise. Hence we would explore the different security services provided by the Windows and how to best use them for our needs.

4.5.4.1 Authentication and Encryption with IIS[17]

The IIS is strongly integrated within the Microsoft Windows Environment and provides many services that are of interest to us. In the following section we discuss some of the security services provided by IIS and how they fit in our objective of securing remoting applications in a distributed environment. The easiest way for securing .NET Remoting components is hosting them in IIS. IIS provides us with authentication against Windows accounts as well as transport-level security through SSL. It can also provide a rudimentary form of access control by restricting calling IP addresses. To utilize the IIS Security, we can host the .NET Remoting components (in our case, the host proxy application) within the ASP.NET runtime infrastructure. IIS gives a number of authentication methods tightly integrated within Windows that includes Anonymous, Basic, Digest and Windows Authentication (that includes NTLM and Kerberos). For details of these protocols please see [**20**]

Furthermore SSL/TLS can be enabled for transport level security.

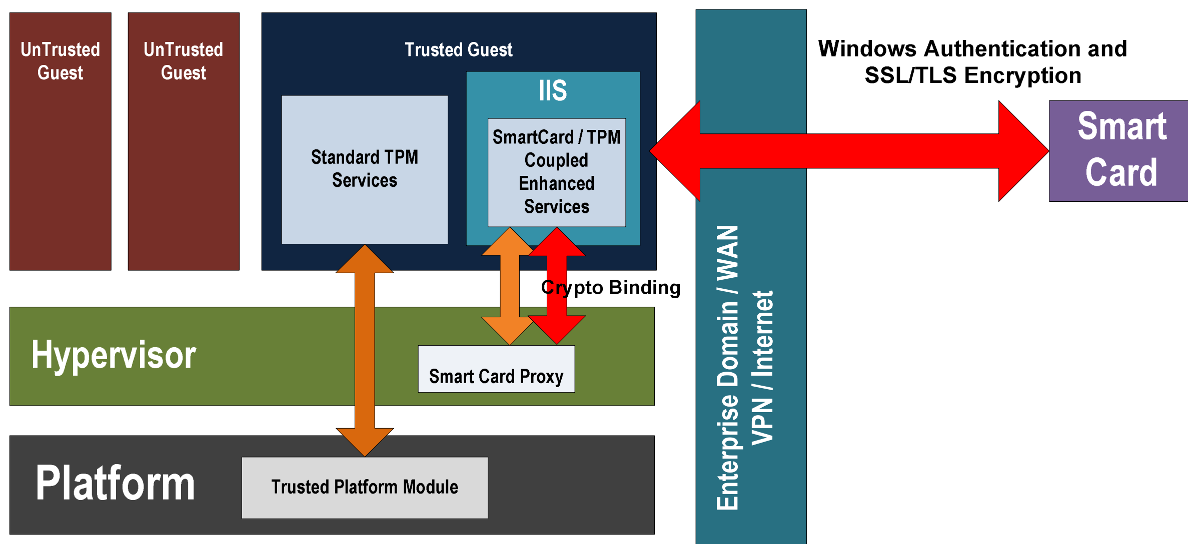


Figure 4-6 A Distributed TPM / Smart Card co-operative model

---

[17] Internet Information Services (formerly Internet Information Server) is a set of Internet-based services for servers using Microsoft Windows that includes web, ftp, news and mail server.

4.6 SUMMARY

This chapter includes a detailed explanation of the enhanced TPM and Smart Card coupled services that were implemented in some of the major functional areas of the TCG specifications. We discussed the fundamental differences in both TPM and the SmartCard platforms. We also discussed in detail some of the major challenges that arise during the coupling process and our solutions for addressing the major security issues in coupling these architectures together. We discussed the requirements and limitations of coupling with smartcards and put forward some recommendations regarding the non supported scenarios.

We started the chapter by discussing what would be an ideal architecture and components of a TPM with internal secure execution environment which can provide VM based code isolation and security guarantees. We discussed some of the similar projects, their propositions and their solutions for extending hardware and software to provide similar offerings for secure code execution. However, most of these projects do not focus exclusively on either providing a secure / trusted execution environment within the TPMs tamper resistant hardware boundary or put forward a solution that can be implemented without changing the current state of hardware and software based platform security. Furthermore most of them require hardware extensions which are time consuming, require changes in the software which is complex and costly process both in terms of time and money.

Hence we put forward an alternative architecture that uses the services of a TPM and couple them with a platform that does provide secure execution environments. In this way we utilize the already established strengths of both the platforms without requiring any additional or special hardware extensions. We discuss in detail how we achieve this target with multi-application post issuance programmable smart cards with a detailed implementation of the coupling process. We described our motivation for choosing SmartCards as the coupling platform and its characteristics that made it a natural choice for this process in Chapter 3.0 . The SmartCard capability requirements for different coupling levels and scenarios are clearly stated with a description and references of our chosen .NET SmartCard platform. Furthermore we also discuss the core platform differences between the SmartCard and the TPM that build up a number of challenges in the process.

For the actual implementation we start off by discussing how to cryptographically bind a TPM and a SmartCard in order for them to identify each other and provide other coupled confidentiality and integrity services. We utilize the TPMs attestation keys and SmartCard generated Identity keys for TPM and SmartCard binding and follow a model similar to SSL for generating / using session based symmetric keys for all calls to and from the TPM / SmartCard. We also extend the .NET Remoting architecture to add encryption to the channels and sinks for making an end-to-end encrypted tunnel of communication between the SmartCard and the TPM. This is important as .NET remoting does not offer any security services on its own. We also shed some light in securing the remoting component / proxy application in a large enterprise distributed environment by utilizing the established security infrastructure provided by Windows

for e.g. utilizing different windows authentication methods, access control mechanisms and encryption / security features offered by the Internet Information Services Server.

The next Chapter would discuss the services that are built and programmed into the SmartCard for providing enhanced TPM coupled security services and secure execution functionality for the applications.

5.0 EXTENDED TPM SERVICES PROVIDED BY THE SMART CARD

5.1 INTRODUCTION

Trusted platforms (like the TPM) are integrated onto a system to provide hardware based platform security. They provide niche security services such as attestation (reliable reporting of platform state), sealing (encrypting some piece of data so it can be only decrypted by authorized software), integrity protection, shielded locations, protected storage and secure management of keys. They act as the Root of Trust (for integrity measurement reporting and storage). For a platform (state) to be trusted, the integrated TPM should be trusted.

SmartCards on the other hand are removable tokens and are usually carried along with the user (for example Identity cards). They are used for multi-factor authentication and different services than that of a TPM. Programmable smartcards also provide an environment and platform where general (or special) purpose applications can be built and loaded before the SmartCard is handed out to the customer. Examples include EMV applications for the Payment Card Industry, SIM applications for telecom etc. Much of the security of the system lies on the tamper resistance of the SmartCard.

As such, the SmartCard is not capable of making integrity measurements of a platform nor does it enjoy the other characteristics of the TPM. Similarly the TPM lacks some features such as providing a secure / trusted environment for general purpose code execution and is not roaming like a SmartCard. Hence we proposed an architecture where we couple both the platforms to provide extended services and combine the best capabilities of both the platforms.

This Chapter discusses some of the enhanced services that can be offered when the TPM and SmartCard are coupled together. A simple TPM and SmartCard binding process is described in Chapter 4.0 TPM & SmartCard Coupling for Enhanced Security Services and these services are based on extending the same binding process further to design and implement extended security services in the SmartCard which utilizes the capabilities of both the SmartCard and the TPM. Code snippets and function implementation details are provided where necessary. These services include from simple TPM/SC Authorization Model to complex flexible sealing, binding and attestation services and hence open new possibilities and ways for providing strong security services for security sensitive applications.

5.2 FLEXIBLE AUTHORIZATION

5.2.1 Introduction:

This section describes the authentication and authorization methods and protocols in the current TCG specifications. It then describes some of the shortcomings and finally a description and implementations of a stronger Smart Card and TPM Authorization Model. The combined authorization model describes simple authentication schemes like authorization storage to

complex smartcard participation in calculating parts of the protocols for stronger binding. We discuss 3 different implementations for different authorization protocols in increasing complexity order.

The TCG specifies different protocols and mechanisms for authentication and access control over the TPM protected objects. These objects and other authorized functions are performed through different authorization protocols defined earlier 2.5.14. These authorization protocols prove to the TPM that the user holds the permission to access those protected objects or perform the authorization functions. The proof comes from the knowledge of a shared secret between the user and the TPM.

The current TPM specifications do not clearly identify the concept of a “user”. The only entities described are the owner and the operator. Hence most of the authentication for resources and objects are defined as “ownership authentication” [**22**]. The TCG doesn’t address any security from the user point of view, as the specifications only deal with the security of the platform. The proof of ownership of Owner Authentication data implies complete control over TPM resources including the ability to clear the TPM.

As with all systems and solutions relying totally on password based schemes, the biggest problem is secure storage of the authorization data. Humans are long known for their poor capabilities in remembering passwords, hence most of the TPM administration products available today including Windows Vista TPM Management Console implements a simple password based scheme for TPM control. The password could be arbitrary length (sometimes enforced by a password policy) and the Authentication Data (AuthData) is computed by taking the SHA-1 hash of the password.

$$OwnerAuthorizationData = SHA\text{-}1(password)$$

Figure 5-1. Password based TPM Administration in Windows Vista

There are a number of problems with the authorization scheme for TPM ownership as described earlier in 2.5.14.21 are highlighted below:

- One factor only authentication
- Subject to brute force and dictionary attacks
- Snoop able
- Guessable
- Easy to lose and forget
- Shareable

These problems are so common even in the current TPM context that the TPM manufactures had to implement lockout and response degradation mechanisms to protect from repeated password entry failures and automated brute force attacks. Some of the solutions addressing this problem include storing passwords and keys in a secure e-wallet such as PGP or some other container (generally persistent secondary storage). However this solution again depends upon knowledge of a master password that protects all the secrets within and is subject to other statistical and algebraic key finding attacks such as mentioned in [**23**]. The response degradation is also implemented in the TPM_IncrementCounter to limit the monotonic counters increment

and save them from being exhausted within a small time frame. This is described in detail in 5.5.1

5.2.2 Smart Card participation in Authorization Protocols

There can be a number of solutions possible depending upon the smart cards capabilities and the security level required. These are listed in increasingly secure order

1. Simple storage of Authorization Data
2. Computation of shared secret in Object access protocols (e.g. OSAP)
3. Computation of the authentication values generation and verification.
4. Encryption of the newly generated Authorization Data (ADIP).

5.2.2.1 Simple Storage of Authorization Data

The simplest (and hence the least secure) method for smart card to participate in the authorization protocols is to store that 20-bytes of authentication data in the smart card itself. This authorization data (or a number of authorization data sets) can be stored securely and provided by the user during the Authorization protocols such as OIAP or OSAP. The user can be authenticated to the smart card by some other means for example PIN. Secure binding between the smart card and the host platform with secure PIN transfer is described in detail in the earlier section 4.5.2.

i. The Process:

i. The user wishes to execute an authorized command and sends it to the platform.
ii. The platform (host) recognizes this as an authorized command and sends a request to the user to enter his PIN.
iii. The user inserts his smart card and enters his PIN to get access to the requested object.
iv. The smart card validates the PIN and sends a response to the platform
v. The platform requests Authorization Data from the Smart Card
vi. The smart card checks whether the authorization data is the same the user requested for. If yes the smart card reads the corresponding authorization data from the card and sends it over to the platform.
vii. The platform receives the authorization data, combines this in the authorization command the user requested in the first step and sends it to the TPM through an authorization protocol.
viii. The TPM receives the command and continues the authorization protocol and allows the user application to access the TPM protected object.

The scenario is illustrated diagrammatically below:

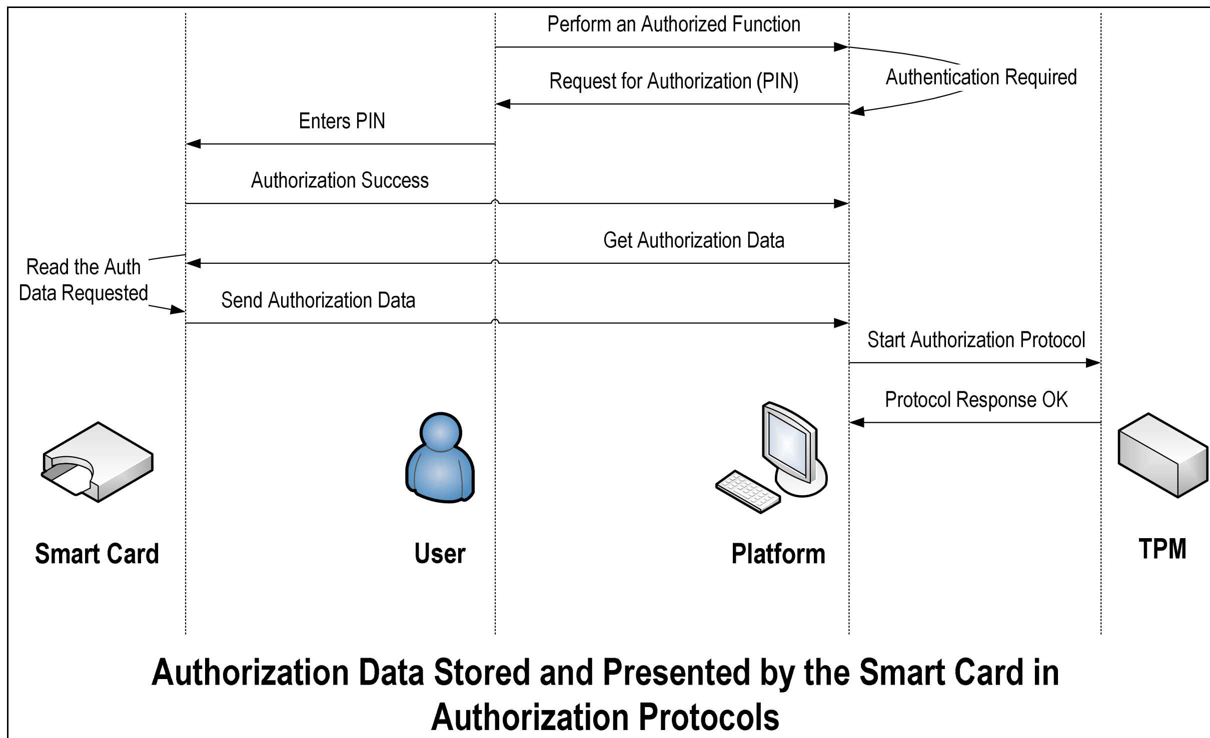


Figure 5-2 Authorization Data Stored & Presented by SmartCard in Authorization Protocols

This method even though not 100% secure, still gives us the following advantages:

- We have two (or even more) factors of authentication. The user needs to be in possession of a smart card that holds the authorization data and knowledge of its PIN to access it. Furthermore it is now at user's discretion whether he wishes to participate in the authorization session or not by presenting or holding the smart card.
- The authorization data that is now protected by the smart card can be assigned to different users who are authenticated and authorized. Furthermore only the smart card that holds the specific authorization data can access the TPM protected object.
- By securing it inside a smart card we ensure that it is now very hard (though not impossible) for the attacker to get hold of the authorization data since the smart card is not easy to duplicate
- Since the smart card provides true random number generation. We can generate and use truly random values for authorization which need not be derived from a password or seeded from some existing value
- Since all the smart card is doing is storage (and protection). This method can be implemented by even most of the cheapest smart cards available today.

5.2.2.2 Computation of Shared Secret

The previous method though providing some benefits is not so convincingly secure. If the path between the smart card and the platform is not protected the authorization data can be

sniffed, snooped or attacked in other similar ways. We further strengthen this model by building a scheme where the authentication data never leaves the card and the smart card computes parts of the protocol requiring the authentication data.

During the smart card participation in the authorization protocol, the smart card performs the value of inAuth and returns to the platform. The platform takes this inAuth value and calculates the remaining part of the authorization data. In addition the smart card is also used to verify the resAuth value. This scheme offers a much higher security then the previous section since the authorization data never leaves the card. The binding for the card is more secure and this method is even less vulnerable to reply attacks since each time the random nonces are transferred from the smart card and the platform and vice versa.

i. The Two way OSAP participation:

As described earlier in OSAP is used to provide proof to the TPM, the ownership of the authorization data for a single specific object. It can be used by allowing multiple commands within the same authorization session but only for that specific object.

i. The TPM generates a handle to the object to track authorization sessions.
ii. The TPM generates an OSAP nonce (nonceEvenOSap) and a TPM replay nonce (nonceEven).
iii. The TPM generates a shared secret by taking the HMAC of the callers nonce (nonceOdd) and the TPMs OSAP nonce

SharedSecret = HMAC(ObjectAuthData, nonceEvenOSAP, nonceOdd)

iv. The HMAC key (K) is the authorization data needed for the use of the key.
v. The AuthHandle, nonceEvenOSAP and nonceEven are sent to the SmartCard.
vi. The SmartCard also computes the shared secret since he knows the authorization data K. It HMACs the same way as TPM in step iii and returns to the host.
vii. The host now calls a TPM_Command for e.g TPM_LoadKey for key (K) which is an authorized command hence requiring to the host to prove knowledge of authorization data for key K that is required to access the protected object.
viii. The Smart Card calculates the inAuth parameter which is consisted of
    a. First take the SHA1 Hash of (TPM Command + Input Arguments + nonceEven + a flag to continue session). We call this value the inputParams
    b. Then take an HMAC with the Key K over the inputParams, the Shared Secret calculated earlier and the parameters for the setup.

inAuth = HMAC(SharedSecret ,SHA1(inputParam), inAuthSetupParam)

ix. The Smart Card hence sends to the TPM the KeyHandle, the inAuth, and the plaintext on which HMAC was generated.
x. The Command is executed by the TPM and a new nonce is generated to replace the last nonce for preventing replay.
xi. The message returned to the SmartCard includes:

a. A SHA1 Hash of the return code by the TPM for the command executed + The TPM Command Code Ordinal + Authorization Session Handle + nonce generated by the TPM (nonceEven) + nonce associated with the shared secret (nonceEvenOSAP) + whether to keep the session open). We call this the outputParams.
b. Then take an HMAC with the Key K over the output Params, the shared secret and the plaintext elements of outputParams.

responseAuth = HMAC(SharedSecret, SHA1(outputParmams), outputParamsSetup

xii. The Smart card now verifies the response Auth and checks the return code whether the command was successfully executed or not.
xiii. The session can be end or continued here as was setup earlier.

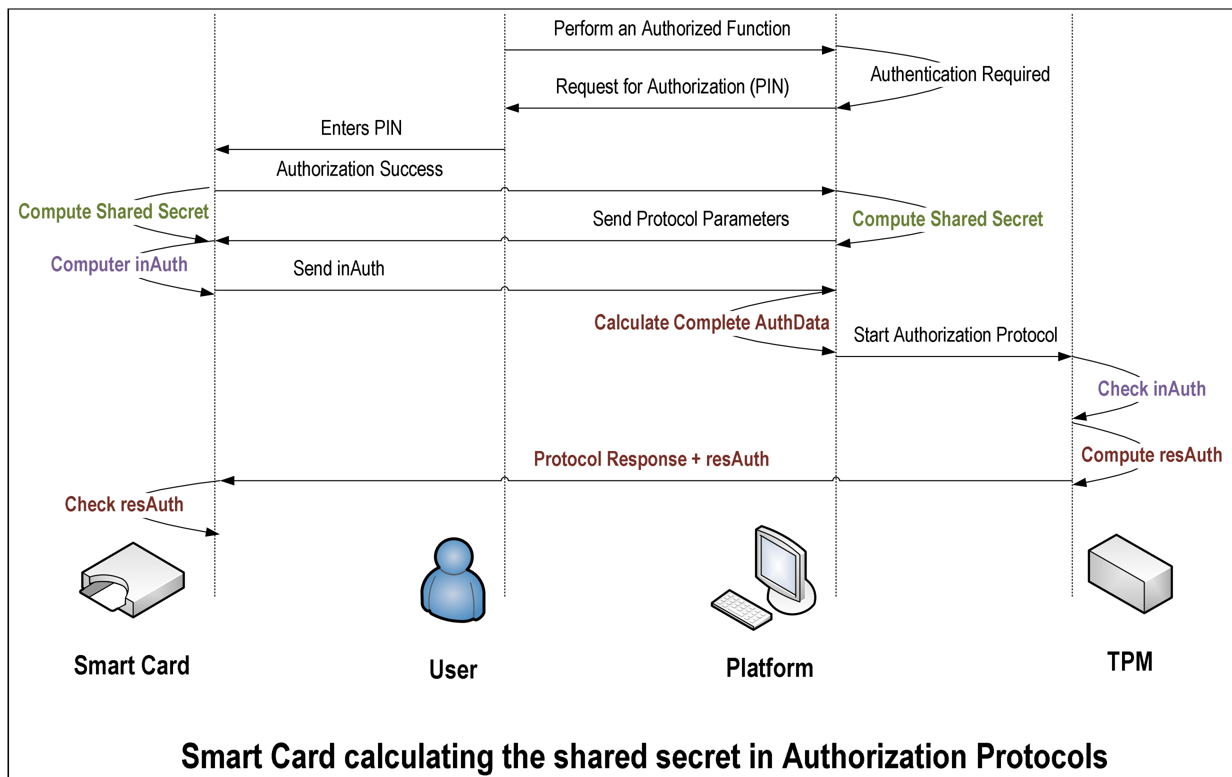


Figure 5-3 SmartCard Calculating the shared secret in Authorization protocols

5.2.2.3 Secure Generation and Insertion of Authorization Data.

In the previous example of two directional OSAP implementation, we let the inAuth value computed by the platform and only allow the smart card to calculate the shared secret. In most cases the shared secret is used per session only so even if it is sniff able or intercepted in between it is not a huge problem, however for cases like ADIP and ADCP protocols it becomes a bigger problem. The ADIP protocol for example uses the shared secret calculated as part of the secure

session creation to further encrypt the new authorization data as they are loaded in the TPM. Hence we need a way to strengthen the smart cards participation more for this scenario. To strengthen the ADIP scheme with smart cards we take the following steps

i. The Smart Card holds the parent key authorization data (which in our example case is the Storage Root Key)
ii. The smart card creates the temporary shared secret for ADIP usage in the same way as described earlier for OSAP
iii. The Smart card securely generates (with true randomness) new authorization data & encrypts it.
iv. In the same way as described for OSAP, the smart card computes the inAuth values.

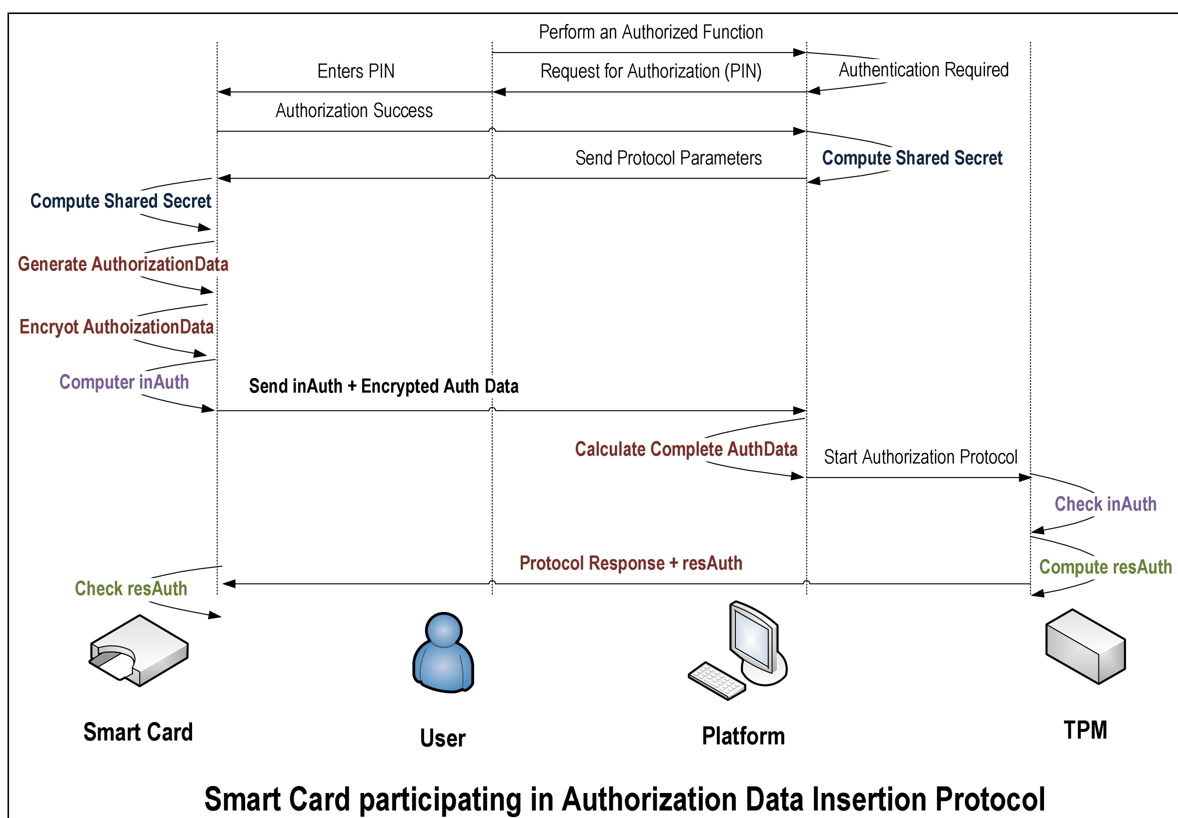


Figure 5-4 Smart Card participating in ADIP

5.2.3 Benefits of Enhanced Authorization.

i. **Separation of Privileges:** Privilege separation partitions a program into two parts, a privileged program which is the monitor and the unprivileged program called the slave [**24**]. The monitor is responsible for all control, trust and privileges which result in a more secure trusted base. In this case the Smart Card acts as a trusted monitor for authorization data and the host proxy as the slave for calling TPM authorization commands.

ii. **Two factor authentication**: The authentication is now based on knowledge of a secret (password) i.e. something you know, and the possession of a smartcard i.e. something you have. Furthermore the smartcard s authentication itself is protected by a PIN and can be replaced by custom authentication.

iii. **User credential portability:** In day to day activities users interact more and more with different computing environments. Platform credentials remain attached to their host computer however user credential usage goes beyond those boundaries. Just one of the advantages of using a smart card based user credential is that if a users credential or authentication data is locked in a TPM, they would be unavailable to the user if the TPM machine is offline or under maintenance.

iv. **Administration simplification:** Think of a scenario where a TPM is used for authenticating users to a platform operation for example machine logon. As of current TPM specifications, there are serious concerns on the TPMs storage space, which raises questions on TPM to be used in a large multi user environment for such a scenario.

v. **A layer of privacy:** Smart Card protected credentials always remain under user control. Users can explicitly decide to use or not to use their credentials on some operation by inserting or not inserting the card and check what elements of their identity is to be revealed if they chose to do so. Credentials stored, managed and being used in a TPM raises a lot of concerns and suspicions to the end users how their credentials are being used.

vi. **Tamper resistant storage:** The SmartCard provides tamper resistance for any data stored in it. Even the most basic smartcards require atleast a modest amount of effort in order to clone them. Furthermore, most of the mass market produced cards are Common Criteria evaluated. The .NET Smart card also provides other security services including role based security. The security services of the card are described in detail in 3.4.1.

5.3 VALIDATING TPM GENERATED QUOTE (ATTESTATION)

5.3.1 Introduction:

The following section describes the Smart Card implementation of ValidateTPMQuote. The smart card uses this service to validate a TPM generated attestation just like any remote challenger. This service can be used in a number of different methods and combined with other protocols depending upon the application. We describe one such application using ValidateTPMQuote in Section 6.1 Enhanced Digital Signature.

A TPM_Quote[**3**] is essentially a Digital signature on the platform state. The platform state is detailed in a log of software events which are also called integrity metrics and are stored in the

Platform Configuration Registers (PCRs). For normal applications scenario a remote server typically sends a request to a particular platform to quote its platform state (also called generating attestation). The server based on this attestation makes a decision on how to move forward depending upon the application. The attestation includes the integrity metrics of both the platform and the software state. The server is assumed to know the public key of the TPM it requested the quote from which is used to verify the digital signature.

As we described in the secure TPM / SmartCard Binding process we generate an Attestation Key from the TPM and securely store the public key in the smart card to which it is bound. Hence we can extend the same attestation idea for the SmartCard which can request the platform to generate a *TPM_Quote* over its state and the SmartCard would verify it like any remote challenger. We therefore implement a service in the smart card of *SC_ValidateTPMQuote* which can be used in a variety of products and protocols for validating a platforms identity and state. An example application detailing its usage is described in the next chapter in section 6.1.

5.3.2 Function Details for SC_ValidateTPMQuote

Pre-Req: SmartCard holding TPMs AIK publicKey *aikpub*
Input: TPM_Quote *t* , Data_Quoted *d*
Output: A Boolean value *b* stating the signature verification was true or false.
Description: return *aikpub*.Verify(*t*,*d*)
Code:

```
/// <summary>
/// Validates the quote RSA enc OAEPP.
/// </summary>
/// <param name="tpmQuote">The TPM quote.</param>
/// <param name="dataQuoted">The data quoted.</param>
/// <param name="publicKey">The public key.</param>
/// <returns></returns>
public bool ValidateQuoteRSAEncOAEPP(byte[] tpmQuote, byte[] dataQuoted, byte[] publicKey)
{
    RSAParameters rsaParams = new RSAParameters();
    //Convert TPM key to RSA key

    RSACryptoServiceProvider rsaKey = new RSACryptoServiceProvider();
    rsaKey.ImportParameters(rsaParams);
    Debug.WriteLine("Parameters imported");

    Debug.WriteLine("Going to verify");
    return rsaKey.VerifyData(dataQuoted, SHA1.Create(),tpmQuote);

}
```

An important note and difference in the TPM and .NET encryption and signature schemes are the different padding and signature schemes supported. In this scenario we omit the details and the code but we have to add / remove the OAEP padding [**25**] from the data that is signed to match the signature byte arrays and verify them.

5.3.3 The Process:

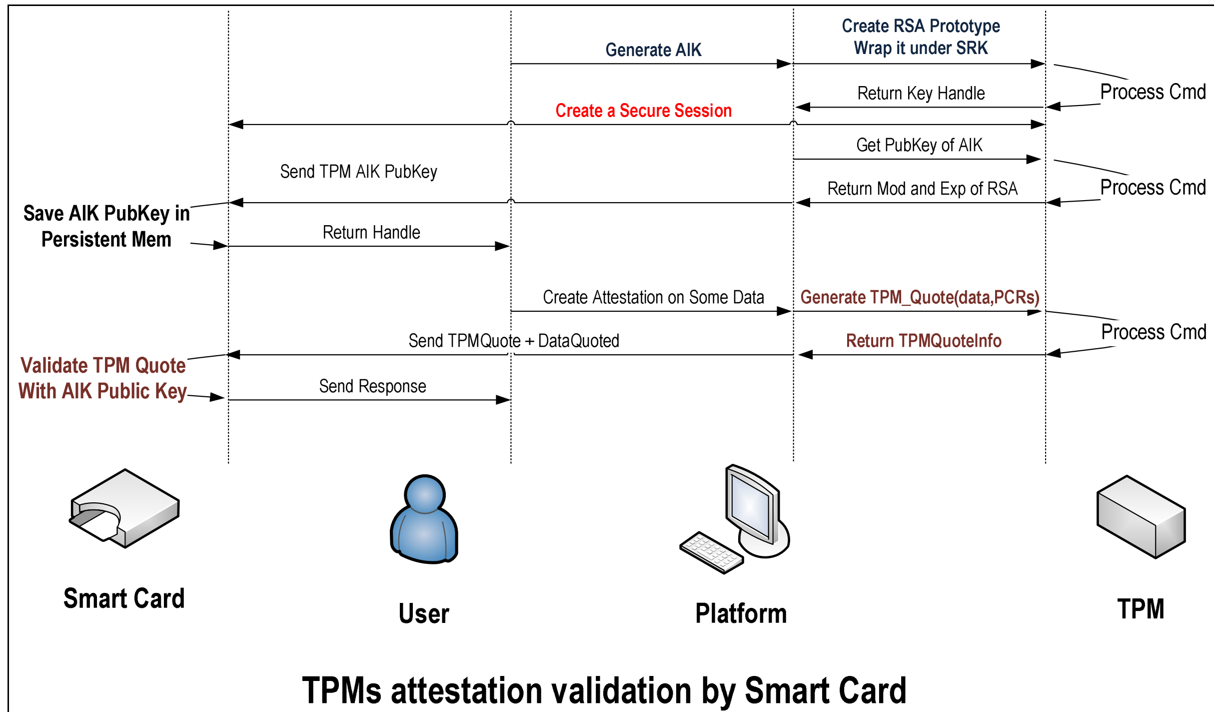


Figure 5-5 TPM Attestation Validation by the Smart Card

5.4 ENHANCED SEALING AND BINDING

Sealing binds (encrypts) some data to a particular platform specifying a trusted future platform state so it can only be decrypted by the platform when it is in the expected configuration. The concept of sealing is one of the most powerful and widely used features of the TPM as it provides strong confidentiality and integrity services. However it comes with certain restrictions and the specifications are very strict over specifying the current and future platform state. The existing TCGs proposed model for sealed data is very simple: data is bound to a specific platform by specifying the future platform state stored in *digestAtRelease* which is a SHA1 over a specific PCR combination holding integrity metrics. However the *digestAtCreation* and *digestAtRelease* need to be specified before the *TPM_Seal* is called and hence much flexibility is lost if the data has to be sealed over a number of different platforms and configurations.

We extend the simple TCG sealing and binding model so that more flexible disclosure options are possible for example:

- Sealing to multiple TPMs and multiple configurations
- Sealing to a complex logic expression based on PCRs
- Sealing to any configuration authorized by a signature with a known public key

We present an enhanced sealing application in 6.2 which is coupled with some other services provided by the smart card.

5.4.1 SmartCard Un-Bind

The Binding flexibility goes further than sealing. Since keys used for *TPM_Data_Bind* can be migratable we can generate and load the entire RSA key securely in the card. This way either the smart card or the TPM can bind and/or unbind the data depending on the application. The SmartCard can hold multiple keys from multiple platforms and Bind / Seal Data to any or all of those based on any policy, signature or configuration required. We present an enhanced sealing application in the next section which is coupled with some other services provided by the smart card. If the smart card is loaded with the private key of a migratable bind key, the smart card can Unbind (decrypt) *TPMBoundData* generated by the particular TPM as well

5.4.1.1 Function Details for SC_Unbind

Pre-Reqs: Migratable TPM Private Key to decrypt *pvtKey* (pre-loaded / securely generated in the card).
Input: Data to Unbind *d*, Policy *p* or a digital signature *ds*
Output: Decrypted data blob *data*
Description: The function assumes a private TPM key (for e.g. a migratable BindKey) already loaded in the card. This key can also be transferred securely depending upon the application. It checks the pre-requisite items before performing *SC_Unbind* (that may include checking for a policy, validating a TPM generated attestation, or validating a digital signature over some data), checks for authorization values and if everything is successful performs the *SC_Unbind* and returns the decrypted data

i. Enhanced Bind Code:

```
public bool TestSmartCardBind(SecureSessionEstablishmentService smartCardService)
        {
            TpmRawCommands tpmCmd = TpmDevice.RawCommand;
            OsapSession osapSession = tpmCmd.Osap(TpmEntityType.KeyHandle, (int)ReservedKeyHandle.Srk,
                                                                           srkAuth, false);
            osapSession.ContinueAuthSession = false;

            OiapSession srkSession = tpmCmd.Oiap(srkAuth, false);
            OiapSession session = tpmCmd.Oiap(ownerAuth, true);

            //Get from Card
            AuthorizationData useAuth = new AuthorizationData(smartCardService.GetUsageAuth());
            AuthorizationData migrationAuth = new
                AuthorizationData(Encoding.Default.GetBytes(smartCardService.GetMigrationAuth()));

            //Generate a random key .. we can do this in the card as well .. it just takes longer
            TpmKey randomGenKey = CreateBindingKeyPrototype();
            //Wrap it with SRK
            TpmKey randGenBindKey = tpmCmd.CreateWrappedKey(TpmKeyHandle.SrkHandle, useAuth,
                                                   migrationAuth, randomGenKey, osapSession);
            //Now load it in the TPM for use
            TpmKeyHandle bindingKeyHandle = tpmCmd.LoadKey(TpmKeyHandle.SrkHandle, randGenBindKey,
                                                                           srkSession);

            byte[] dataToBind = Encoding.Default.GetBytes("Sample Data To Bind  ... ");


            byte[] boundDataWithSmartCard =
```

```
                smartCardService.CreateTPMBoundData(randGenBindKey.StorePubKey.key, dataToBind);
    PrintArray(boundDataWithSmartCard, " SC Bound");

    byte[] scUnboundData = tpmCmd.Unbind(bindingKeyHandle, boundDataWithSmartCard, session);
    PrintArray(scUnboundData, " SC UnBound");

    // and see if the answer is correct
    if (!ArraysAreEqual(dataToBind, scUnboundData))
    {
       throw new Exception("Unbind Error");
    }
    tpmCmd.FlushSpecific(bindingKeyHandle);
    return true;
}
```

ii. Create TPM Bound Data

```
/// <summary>
/// Creates a TPM bound data.
/// </summary>
/// <param name="bindingKey">The binding key.</param>
/// <param name="dataToBind">The data to bind.</param>
/// <returns></returns>
public byte[] CreateTPMBoundData(byte[] bindingKey, byte[] dataToBind)
{
    RSAParameters rsaParams = new RSAParameters();
    rsaParams.Modulus = bindingKey;
    rsaParams.Exponent = new byte[] { 1, 0, 1 }; //This one just works both in and outside the
card

    //rsaParams.Exponent = Encoding.ASCII.GetBytes("65537");
    RSACryptoServiceProvider rsaKey = new RSACryptoServiceProvider();
    rsaKey.ImportParameters(rsaParams);

    byte[] tpmSerializeForBound = new byte[] { 1, 1, 0, 0, 2 };
    byte[] serializedBoundData = new byte[tpmSerializeForBound.Length + dataToBind.Length];
    Array.Copy(tpmSerializeForBound, serializedBoundData, tpmSerializeForBound.Length);

    for (int i = 0; i < dataToBind.Length; i++)
    {
        serializedBoundData[i + tpmSerializeForBound.Length] = dataToBind[i];
    }

   // byte[] cipherText2 = RsaEngine.RsaTpmEncodeAndEncrypt(rsaP.Modulus, serializedBoundData);

    //byte[] parms = new byte[] { (byte)'T', (byte)'C', (byte)'P', (byte)'A' };
    byte[] tcpaParams = Encoding.ASCII.GetBytes("TCPA");
    byte[] EncodedMessage = RSAesOAEP.Encode(serializedBoundData, tcpaParams, 255);

    //byte[] pHash = Sha1Engine.ComputeHash(parms);

    byte[] boundData = rsaKey.EncryptValue(EncodedMessage);

    return boundData;
}
```

This scenario is detailed in the diagram below:

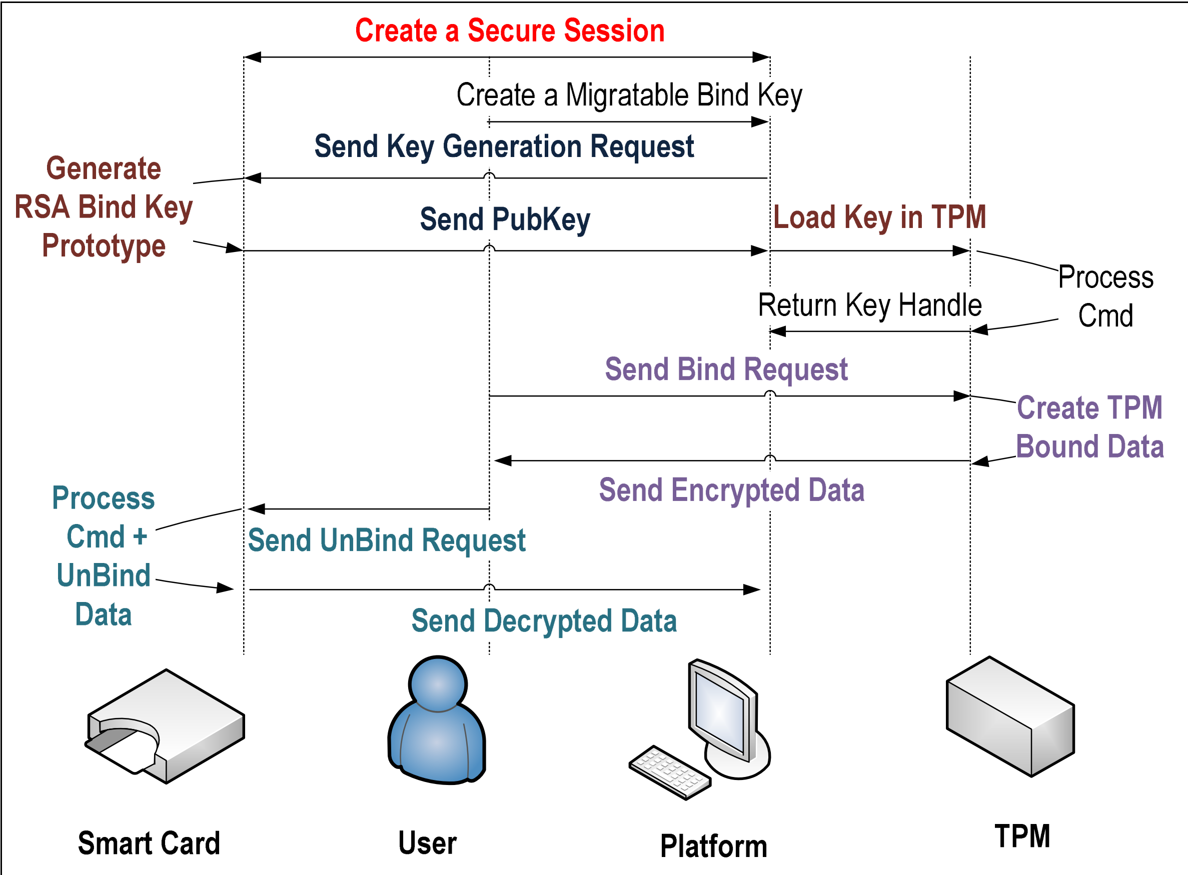


Figure 5-6 TPM Bound Data decrypted by a SmartCard by SC_UnBind.

5.4.2 SmartCard Bind / Seal

The reverse operation (where a smartcard can seal / bound data to a particular platform / set of platforms) is relatively simpler. All we need is to have the public Key loaded in the card to perform *SC_Bind.* The implementation of *SC_Seal* is a little more complex as the program logic to create future trusted state (i.e. creating *digestAtRelease* values for the platform) for the data to be decrypted will depend on how the sealing is implemented. Diagrammed below is a simple implementation of SmartCard binding data to a particular platform and the platform holding the correct key decrypting it on the system.

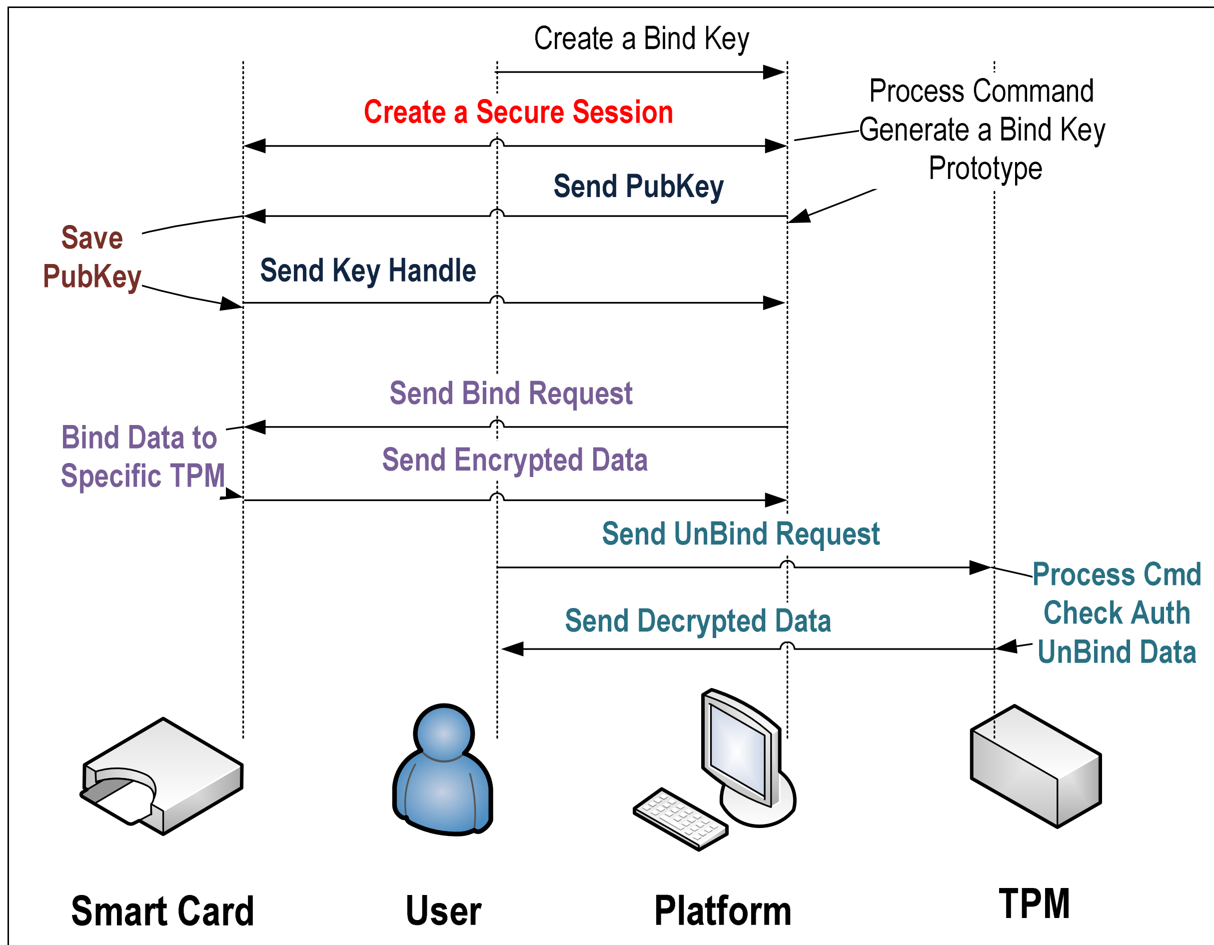


Figure 5-7 Sample scenario for SmartCard Seal / Bind data for a particular TPM

5.4.2.1 Function Details for SC_Bind

Pre-Reqs: Public Key to encrypt *pubKey* (pre-loaded sealPubKey, securely generalted migratable BindKey)
Input: Data to Bind *d*, Future state specification *hash*
Output: A Smart Card generated TPM Bound Data *encblob*
Description: The function takes a public TPM key and encrypts some data. It creates a *TPM_BoundData* and specifies the future state of the platform for the data to be decrypted just like *TPM_Seal*. The future state is essentially a SHA1 hash which is taken over the platform state that would be expected for the data to be unsealed.

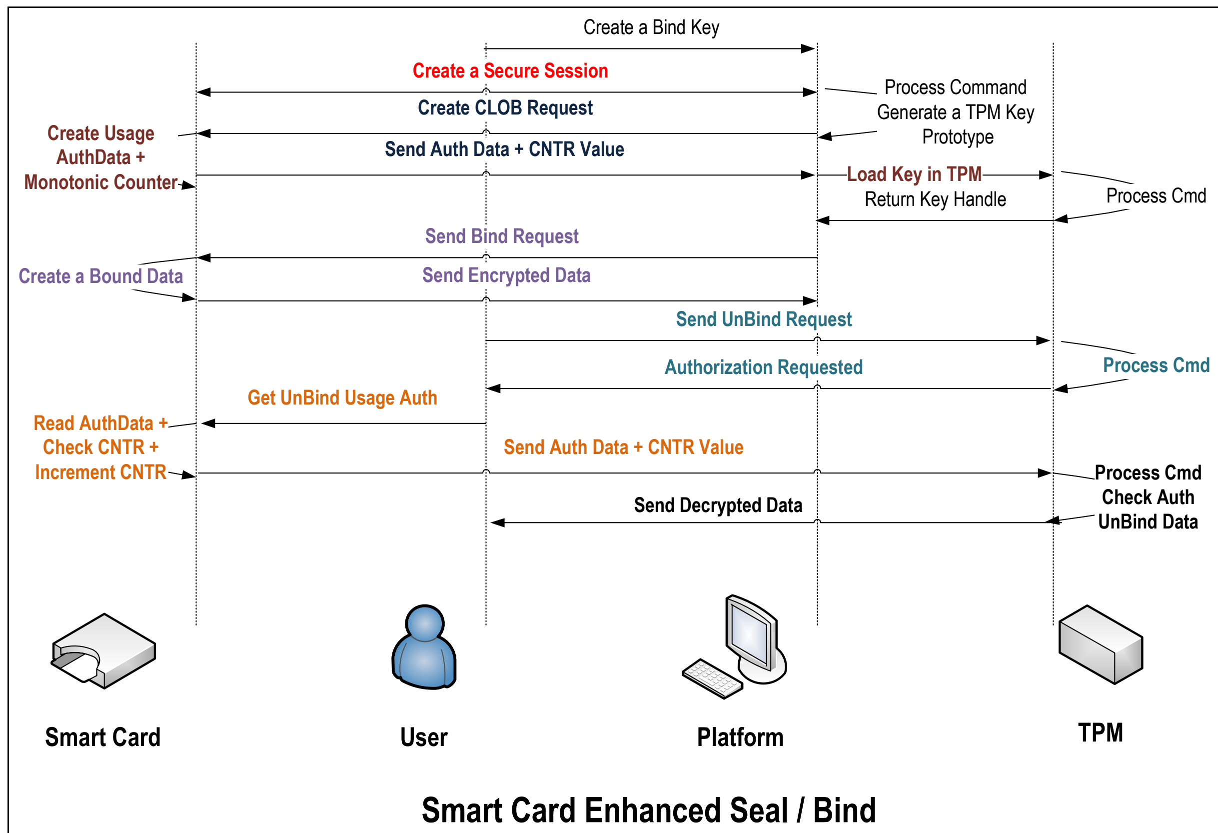


Figure 5-8 Smart Card Enhanced Seal / Bind

5.5 ENHANCED VIRTUAL MONOTONIC COUNTERS

A Monotonic Counter is a trusted atomic tamper resistant counter, whose value once incremented cannot be reverted back. This special property (that also allows for protection against replay attacks) allows for many different interesting applications that include but not limited to DRM technologies, e-cash tokens, count limited objects etc.

5.5.1 Issues with the TPM Monotonic Counters:

The current TPM specifications [**3**] as of this writing limit the total number of base monotonic counters in a TPM to be 4 (which are called the base counters). Also, out of these 4; only 1 counter can be incremented per boot session, the others can only be read. To use another base counter the system has to be rebooted. The motivation for such a design was to have a monotonic counter per trusted operating system (max 4), so each operating system has its own counter which cannot be updated after its boot sequence [**4**].

Different TPM Manufacturers limit the speed of incrementing a monotonic counter to counter against denial of service attacks and prevent a counter rollover as a single monotonic counter is a 32bit value whose maximum value can be exhausted within 2^{32} -1 values. The

TPM_IncrementCounter[**3**] call has a throttling limit of 5 seconds that would ensure that even if continuously updated the update cycle for a complete counter overflow is atleast 7 years.[**4**]

Furthermore the monotonic counters are not integrated directly with any of the other services that the TPM provides. Hence their use (and implementation of any extended counters pointing to the base) has been left on the implementation of the OS.

These specification of the TCG for monotonic counters naturally limit the ability of the user mode applications to utilize directly, the full potential of monotonic counters provided by the TPM, hence there have been a number of efforts to enhance the use of these counters for example [**26**].

5.5.2 Smart Card Enhanced Counters:

To provide an easy, usable framework and shielding the complexity of the Monotonic counters provided by the TPM, we implement some user and programmer friendly functions for implementing an extended counter system within the smart cards which have the following benefits.

1. We can use arbitrary length counters using Big Integer based classes. For demonstration purpose we implement the services using a both 32bit and 64bit counters (having a max value of 18446744073709551615), which can be extended as required.
2. The extended counters can be bounded to (and hence taken a count of) any object (that includes but not limited to Authorization Data, Cryptographic keys etc, hence giving an effective implementation of Count Limited Objects or Clobs[**26**].

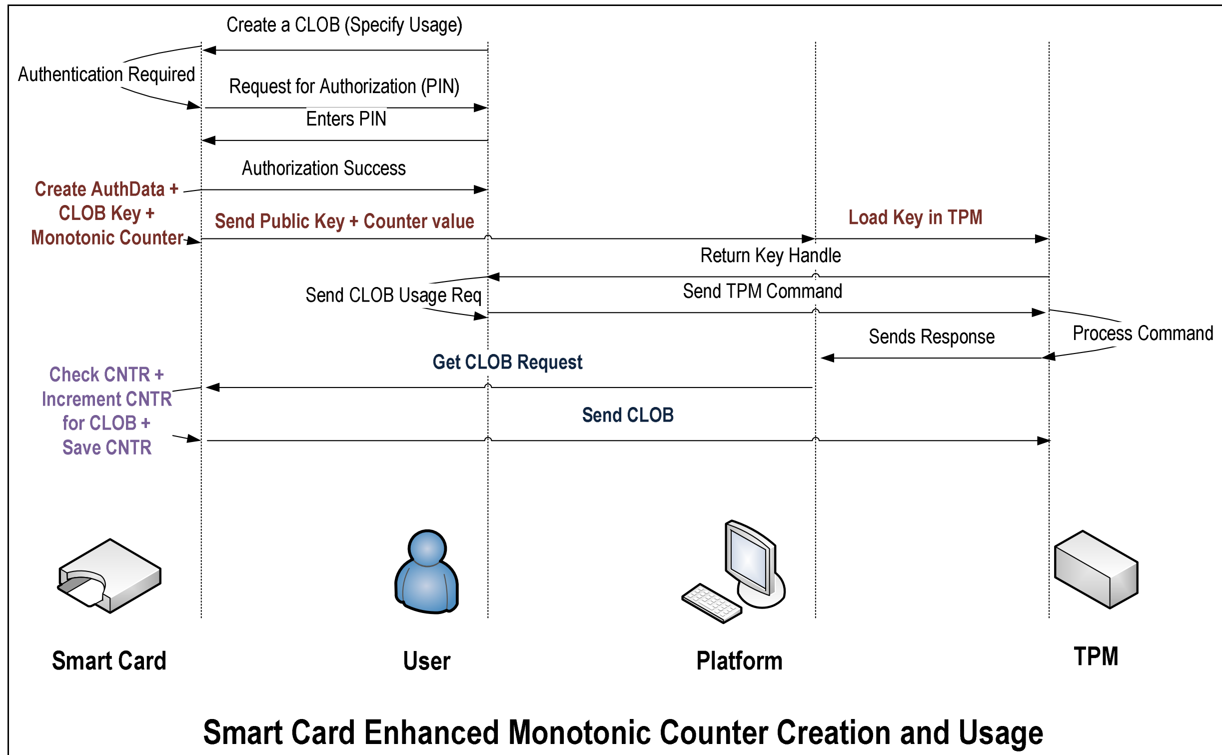


Figure 5-9 Smart Card Enhanced Monotonic Counter Creation and Usage

5.5.2.2 Implementation Details and Function Calls:

1. *SC_CreateCounter*
   Input: A CounterName *n*, AuthrozationData *auth,* Object to Bind *clob* (optional)
   Output: A Monotonic Counter *mc*
   Description: Creates a Monotonic Counter and optionally binds it with an object. Authorization data is provided for authorized operations including increment and release.

2. *SC_IncrementCounter*
   Input: The CounterName *n*, AuthrozationData *auth,* Object to Bind *clob* (optional)
   Output: Incremented Counter Value *mc+1*
   Description: Increments a particular monotonic counter. Increment Authorization is required as this is an authorized operation
   Usage:

```
/// <summary>
/// Increments the counter.
/// </summary>
/// <param name="counterName">Name of the counter.</param>
/// <returns></returns>
public int IncrementCounter(string counterName, AuthorizationData authData)
{
    if CheckAuth(counterName, authData){


    if (MonoCounters.ContainsKey(counterName))
```

```
        {
            int originalValue = (int)MonoCounters[counterName];
            originalValue++;
            MonoCounters[counterName] = originalValue;
            SaveCounterInPersistentMem(counterName);
            return (int)MonoCounters[counterName];
        }
        else
        {
            return -1;
        }

        }
        }
```

3. *SC_GetCounterValue*

   Input: The CounterName *n*

   Output: The Counter Value *v*

   Description: Gets the Int64 value of a monotonic Counter

   Usage:

```
    /// <summary>
    /// Gets the counter value.
    /// </summary>
    /// <param name="counterName">Name of the counter.</param>
    /// <returns></returns>
    public int GetCounterValue(string counterName)
    {
        if (MonoCounters.ContainsKey(counterName))
        {
            return (int)MonoCounters[counterName];
        }
        else
        {
            return -1;
        }
    }
```

4. *SC_ReleaseCounter*

   Input: The CounterName *n* , AuthrozationData *auth*

   Output: Boolean value indicating release success.

   Description: Releases a counter, deletes the memory and persistent storage acquired by the counter. Release Authorization is required.

   Usage:

5.5.2.3 Helper Functions:

```
/// <summary>
        /// Saves the counters in persistent mem.
        /// </summary>
        /// <returns></returns>
        [Transaction]
        private bool SaveCountersInPersistentMem()
        {
            FileStream persistentCounterStorage = new FileStream(@"D:\Pub\Counters.dat",
                                             FileMode.OpenOrCreate, FileAccess.ReadWrite);
            BinaryFormatter bf = new BinaryFormatter();
            bf.Serialize(persistentCounterStorage, MonoCounters);
            persistentCounterStorage.Close();
            return true;
```

```
}
```

5.5.2.4 Implementing .NET Transactions in the Smart Card

The .NET Smart Card Framework supports a persistent transaction model that ensures the integrity of data on the Gemalto .NET Card, despite frequent and sometimes unpredictable physical removal of the card from the system or terminal with which it is communicating.

As we mentioned before, unlike the TPM which is bound to a particular platform, a smart card is roaming. It has more chances of being stolen, lost or get into malicious hands. Furthermore, the smart card takes the power from the reader it is inserted in to perform all the calculations. A smart card can be removed from a reader at unpredictable times (maliciously or otherwise is not the primary concern), What we need are atomic operations on creation, updates and deletes etc. When the removal occurs, the card will lose power immediately. This is a serious problem if you were in the middle of updating a sequence of object fields. For example, we might be updating some balance in e-cash tokens in an object. If are not able to update a monotonic counter correctly, (say we return the authorization data in the count limited object) but the card is removed before we update the "UsageCount" field, we would end up our Count Limited Objects being defeated. In a transaction based model, we want to be sure either we do all the operations in entirety or none. A number of power attacks are known including Static and Differential Power Analysis (SPA / DPA) that have been very successful in the past to recover secret keys from the card [**5**] .

Card removals aren't the only interruption that can cause a problem. We are also concerned with a code or data failure which could cause an exception to be thrown in the middle of an update operation. This could leave the card in an inconsistent state (or at times even corrupt the memory and persistent storage). In this case, you need a mechanism for rolling back any field updates to the original state.

i. How transactions work

Transactions work by ensuring that changes are not committed until the end of a transaction. When you create a transaction, the card preserves the state of the object before the transaction began, and will revert to this state at power up if the transaction was unable to complete. Any method (including the method initially called by the client) can be marked as a sub-transaction by the addition of a special transaction attribute, Transaction. Also, any method that is called by a method that is under transaction is also considered to be under transaction. If the transaction method returns an uncaught exception, the transaction is not committed, and objects and static data fields are returned to their previous state.

Example

In the example, the Increment method is marked as a transaction using the Transaction attribute.

```
[Transaction]
/// <summary>
/// Increments the counter.
/// </summary>
```

```
/// <param name="counterName">Name of the counter.</param>
/// <returns></returns>
public int IncrementCounter(string counterName)
{
    if (MonoCounters.ContainsKey(counterName))
    {
        int originalValue = (int)MonoCounters[counterName];
        originalValue++;
        MonoCounters[counterName] = originalValue;
        SaveCounterInPersistentMem(counterName);
        return (int)MonoCounters[counterName];
    }
    else
    {
        return -1;
    }
}
```

If the method is partially executed (say the card is pulled out and loses power) or there is an uncaught exception, the changes to the variables are never committed or are rolled back because of the Transaction attribute.

ii. Out-of-Transaction objects

Although in general you would like to roll back any modifications made to your card if the operation is interrupted, there may be cases where you might want the method to be under transaction, but for a particular field of an object to be "out of transaction". One motivation for this is the PIN class. You can imagine that the logic for a PIN class might be for the caller to send PIN data to a method, and the method would then pass the data to the PIN object. If the data does not match the PIN, the number of remaining tries on the PIN is decreased, and the method returns. What we would want to avoid is for an attacker to be able to try a PIN, cut power to the card if it fails, and have the number of remaining tries reset by a transaction.

To avoid this type of attack, the .NET framework provides an OutOfTransaction Attribute that can be applied to the fields of an object. Fields annotated by this attribute are always considered to be "out of transaction". That means that even if it is used inside a method that is under transaction, the field will not be rolled back to its previous state if the transaction is interrupted. The PIN class of the card is built using an OutOfTransaction attribute.

iii. Storage

The Gemalto .NET Card contains both persistent memory and volatile memory that are used for data storage. The persistent memory acts as persistent storage for the card - data persists in it even after the card is removed from a smart card reader. Volatile memory is reset when the card loses power and cannot be used for persistent storage.

iv. Data stored in persistent memory

The persistent memory of the card is used for objects that are created by your application. Any object created with a new keyword (whether this is done by the developer or by an underlying software package) is created in persistent memory, and will remain on the card until it is no

longer referenced and has been garbage collected. In addition, any fields of an object will be stored in persistent memory. Data stored in the file system is always stored in persistent memory.

5.6 SUMMARY

In this chapter we discuss some of the enhanced services that are possible with the TPM-and-SmartCard collaborative model and discussed our detailed implementations of them. These services are broadly categorized into 4 categories:

Flexible Authorization: Smart Card participates in the TCG authorization protocols with different methods depending upon the security level required and smartcard capability requirements. Those include from simple AuthorizationData storage in the card to complex HMAC based shared secret calculations for OSAP and ADIP protocols.

Validating TPMs Generated Quote: The smart card uses this service to validate a TPM generated attestation just like any remote challenger. This service can be used in a number of different methods and combined with other protocols depending upon the application. We use this service exposed by the SmartCard to check the authenticity and genuineness of the TPMs generated attestations. We can use this service in a number of ways, some of the applications using this service are described in Chapter 6.0 .

Flexible Sealing and Binding: We extend the simple TCG sealing and binding model so that more flexible disclosure options are possible for example Sealing to multiple TPMs and multiple configurations, Sealing to a complex logic expression based on PCRs or Sealing to any configuration authorized by a signature with a known public key etc. A SmartCard can Bind / encrypt data to any platform (or set of platforms) for which it is bound to. This also helps us build some interesting DRM applications, a couple of them are described in Chapter 6.0 .

Enhanced Monotonic Counters: The Smart Card also provides an easy, usable framework and shielding the complexity of the Monotonic counters provided by the TPM. We implement some user and programmer friendly functions for implementing an extended counter system within the smart cards using arbitrary length or 64bit counters. We bound those counters with different objects (such as keys) to provide an implementation of Count Limited Objects, hence controlling the use of keys and objects. All counter implementation functionality including the number of counters, their maximum values, the objects they are bound to etc can be controlled by the user.

Based on these enhanced services, that are implemented in the SmartCard, we have built the basic building blocks that are needed for smart card and TPM coupling. Hence any general purpose application can now be developed targeting the .NET framework and executed securely inside the smartcard. Moreover since the SmartCard is coupled with a TPM using cryptographic services described earlier we can build any application that utilizes the TPM protocols and primitives as well. Hence in the next chapter we discuss some of the applications utilizing this extended TPM and Smart Card model and describe how these applications change the way conventional TPM and Smartcard applications were perceived. We also shed some light on similar applications and further work.

6.0 SAMPLE APPLICATIONS FOR THE TPM & SC CO-OPERATIVE MODEL

This chapter details some of the sample applications developed from the building blocks described in Chapter 5.0 Extended TPM Services Provided by the Smart Card. For demonstration we provide a complete implementation of 2 major applications which utilize a number of small applications and building blocks themselves. Brief descriptions of some other application ideas are also provided for the reader's interest.

6.1 ENHANCED DIGITAL SIGNATURE

The first sample application that is developed in the smart card and TPM co-operative model is The Enhanced Digital Signature application. We call it enhanced because it enhances the conventional digital signature process by combining the best capabilities of smart cards, trusted computing and digital signatures together and solves a number of problems and issues with the digital signature systems present today. It uses the SC_ValidateTPMQuote primitive provided by the smartcard and TPM co-operative model and uses the Digital Signature Services provided by the SmartCard.

We start the applications design motivation by describing a normal digital signature process used in the enterprises.

i. A user wishes to sign a document. For an example scenario let's say that the user is a company's representative signing the document of a business proposal on his laptop.
ii. The user initiates the request to his system to sign the document with his smart card.
iii. The system treats this as an authenticated operation for a smart card and requests the user to enter his PIN before the smartcard can sign the document with its private key.
iv. The smart card authenticates the user with his PIN
v. The smart card creates a digital signature on the document which is passed to it by the host application and returns it.
vi. The user submits the signed document to the other party for records.

There are a number of problems with this scenario that includes both practical and security issues that have been discussed over the past. We further strengthen this scheme with our enhanced digital signature scheme and the step of operations now goes as follows:

i. A user wishes to sign a document. We take the same earlier scenario where the user is a company's representative signing the document of a business proposal on his laptop.
ii. The user initiates the request to his system to sign the document with his smart card.
iii. The system treats this as an authenticated operation for a smart card and requests the user to enter his PIN before the smartcard can sign the document with its private key.
iv. The smart card authenticates the user with his PIN
v. The smart card now instead of creating a digital signature on the data requests the platform to generate an attestation of its state.

vi. The TPM generates a TPM_Quote on its platform state, the current PCR values which may include the software running on the system, some nonce provided by the card which the card can later verify.
vii. The host application sends this attestation to the card and the card validates it since it holds the public key of the key used to create an attestation.
viii. If the TPM_Quote is verified by the card, the smart card now creates a digital signature on the TPM_Quote and the Document that needs to be signed
ix. The user submits the signed data to the other party for records.

As we can see, this scheme holds a number of interesting characteristics which were not possible in the normal Digital Signature methods. This method also further strengthens the trust relations and eliminates the attack schemes on digital signatures as described in [**27**].

First, now the signature created on the data not only proves that a particular smart card signed the data, it also proves that the document was signed on a particular machine of which the attestation was generated. Second, using a stronger smartcard and TPM coupling and trusted I/O this method can be further strengthened eliminating the problems where a person later disputes the digital signature on a note that he didn't 'see' the document he was signing.

The enhanced digital signature scheme is described diagrammatically below:

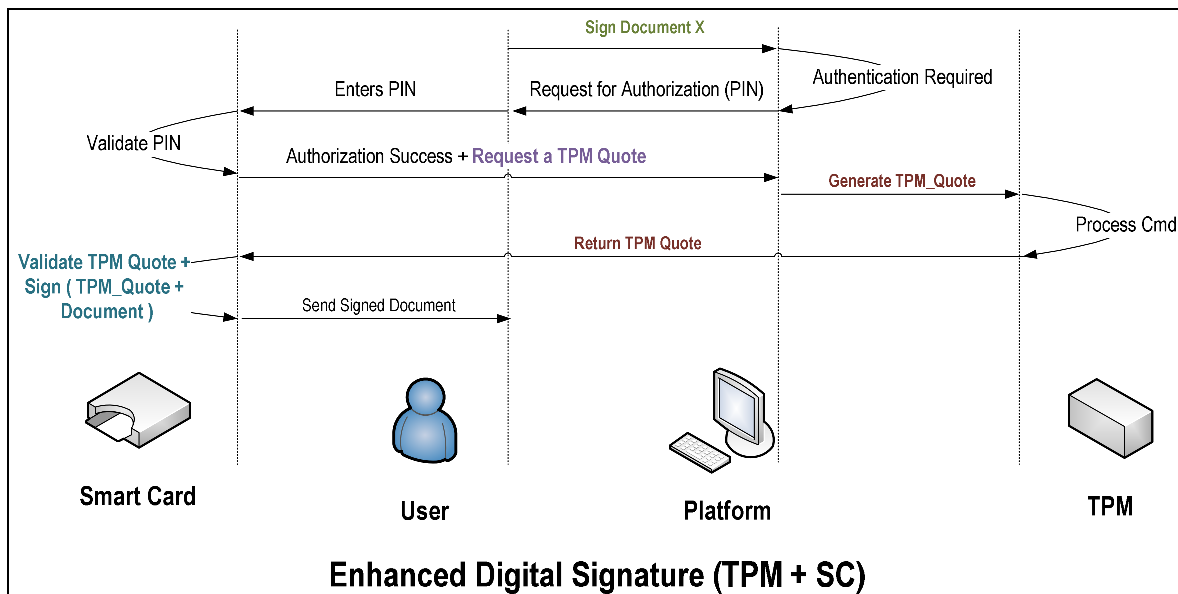


Figure 6-1 Enhanced Digital Signature by TPM and SC

```
<summary>
/// Generates the quote.
/// </summary>
/// <param name="service">The service.</param>
/// <returns></returns>
public static TPMQuoteInfo GenerateQuote(SecureSessionEstablishmentService service)
```

```
{
   // GenerateQuote();
    TPMQuoteInfo tpmGeneratedQuote = new TPMQuoteInfo();
    TpmContext TpmDevice = new TpmContext();
    TpmRawCommands tpmCmd = TpmDevice.RawCommand;

    //We will limit the use of this key by a monotonic Counter inside the card
    TpmKey idKey = (TpmKey)TpmStructureBase.XmlDeserializeFromFile("IdentityKey.xml", typeof(TpmKey));
    //idKey.AlgorithmParms.SignatureScheme = TpmSignatureScheme.;

    int numTimesUsed;
    AuthorizationData quoteUsageAuth = new AuthorizationData();

    quoteUsageAuth.authData = service.GetQuoteUsageAuth(idKey.StorePubKey.key, out numTimesUsed);

    if (quoteUsageAuth.authData != null)
    {

        string counterName = Convert.ToBase64String(SHA1Engine.ComputeHash(idKey.StorePubKey.key));
        //Debug.WriteLine(numTimesUsed);
        Debug.WriteLine("Testing Counter Value by Name "+ service.GetCounterValue(counterName));


        if (numTimesUsed >= 0)
        {
            OiapSession signingKeySession = tpmCmd.Oiap(quoteUsageAuth, true);
            TpmKeyHandle signingKeyHandle = tpmCmd.LoadKey(TpmKeyHandle.SrkHandle, idKey,
                                                           signingKeySession);

            TpmHash nonce = new TpmHash(); // 20 byte anything
            //nonce.hash = SHA1Engine.ComputeHash(randomNonceGen);
            nonce.hash = service.GetRandomNonce();

            PcrSelection pcrsToQuote = new PcrSelection(new int[] { 0, 1, 2, 3, 4 }); //export ?
            PcrComposite pcrSignedData;

            tpmGeneratedQuote.TPMQuote = tpmCmd.Quote(signingKeyHandle, nonce, pcrsToQuote,
                                                      signingKeySession, out pcrSignedData);

            TpmQuoteInfo quotInfo = new TpmQuoteInfo();
            Array.Copy(Encoding.ASCII.GetBytes("QUOT"), quotInfo.Fixed, 4);
            quotInfo.ExternalData = nonce;
            byte[] tpmRep = pcrSignedData.GetTpmRepresentation();

            quotInfo.CompositeHashDigestValue = new TpmHash(Sha1Engine.ComputeHash(tpmRep));

            tpmGeneratedQuote.DataQuoted = quotInfo.GetTpmRepresentation();

            TpmPubKey signingPubKey = tpmCmd.GetPubKey(signingKeyHandle, signingKeySession);
            tpmGeneratedQuote.VerificationKey = signingPubKey.PubKey.key;
            return tpmGeneratedQuote;
        }
        else
        {
            throw new Exception("Counter Error");
        }

    }
    else
    {
        throw new Exception("Generate a new Quote Key.");
    }

}
```

6.2 ROAMING DRM

This sample application implements a number of enhanced smart card coupled TPM services developed earlier including:

- Enhanced Monotonic Counters.
- Count Limited Objects.
- Flexible Sealing.

TCG defines four classes of protected message exchange; Binding, Signing, Sealed-Binding (Sealing) and Sealed-Signing. As described earlier for the Seal and Bind operation 2.5.15.2 a data can be bounded to a particular TPM and the data would only be decrypted if the platform is in the particular expected state. These are powerful functionalities not offered by conventional platforms and we extend them further to provide more secure and flexible services. We use these services to provide flexible seal or bind operations with count limited objects to provide a roaming digital rights management platform.

The Smart Card application exposes the functionality of Monotonic Counters, SC_Seal or SC_Bind and creates a CLOB (Count Limited Object), which bounds the use of a binding or unbinding key with a monotonic counter. Hence we can control the authorization data of a particular key to be used n number of times.

The sample application encrypts a file (which was an MS Word file as an example) but any media can be encrypted and stored the same way depending upon the need.

The Smart card can either be used to encrypt (Bind) data to a particular / or a set of TPMs by encrypting data for each public key, or can be used to decrypt (unbind) data encrypted by the smart card. This is only true for binding keys as they can be made migratable within the TCG. For Sealing operations, the Smart card can only seal data for a particular TPM and the TPM would be able to decrypt it.

Description:

i. The user requests through the platform to the smart card to create a Count Limited Object (Clob)
ii. The Smart Card requests for authentication (PIN)
iii. On successful authorization the Card checks what kind of Clob is requested, for example we create a Binding Key that would be used 10 number of times which is specified as part of the request.
iv. The usage specifications can be based on some part of policy (for example each Quote Generation key could be used a max of 10 times). This policy can be pre-loaded in the card as an XML file.
v. The Card creates a Binding Key prototype (RSA Key pair), Generates a random Authorization Data to be used to request an authorized command related to the key.
vi. The card also creates a Monotonic Counter through the SC_CreateCounter primitive and inks the Monotonic Counter with the Binding Key Authorization data.

vii. The Card returns to the platform, the public part of the Binding Key, the authorization data and the counter value, which by default starts from 0.
viii. The platform generates an AIK prototype and loads the key returned from the card in the TPM for future use.
ix. When needed, the key binding key just loaded in the TPM would be used to Bind Some piece of data. The authorization for key usage is fetched from the card, however we don't increment the counter since the AuthData is only being used to encrypt the data.
x. Whenever the platform wants to unbind (decrypt) the data for processing, it sends the request to the TPM to UnBind it. The platform sends the AuthData and an UnBind request to the Card.
xi. The SmartCard holding the private key of the RSA pair which is to be used for decrypting the data, it checks the AuthData sent from the platform, check's whether the counter created with the key has crossed its usage limit or not.
xii. If the counter is still valid for use and below the required threshold, the card increments the counter and decrypts the data.
xiii. The decrypted (UnBound) data is sent to the platform along with the updated counter value.
xiv. The update of the counter and the decryption process has to be atomic; otherwise the application can be targeted for different kinds of attacks.
xv. We implement the atomicity of the IncrementCounter through the transaction services provided by the card and are described in detail in 5.5.2.

The application flow is described diagrammatically below.

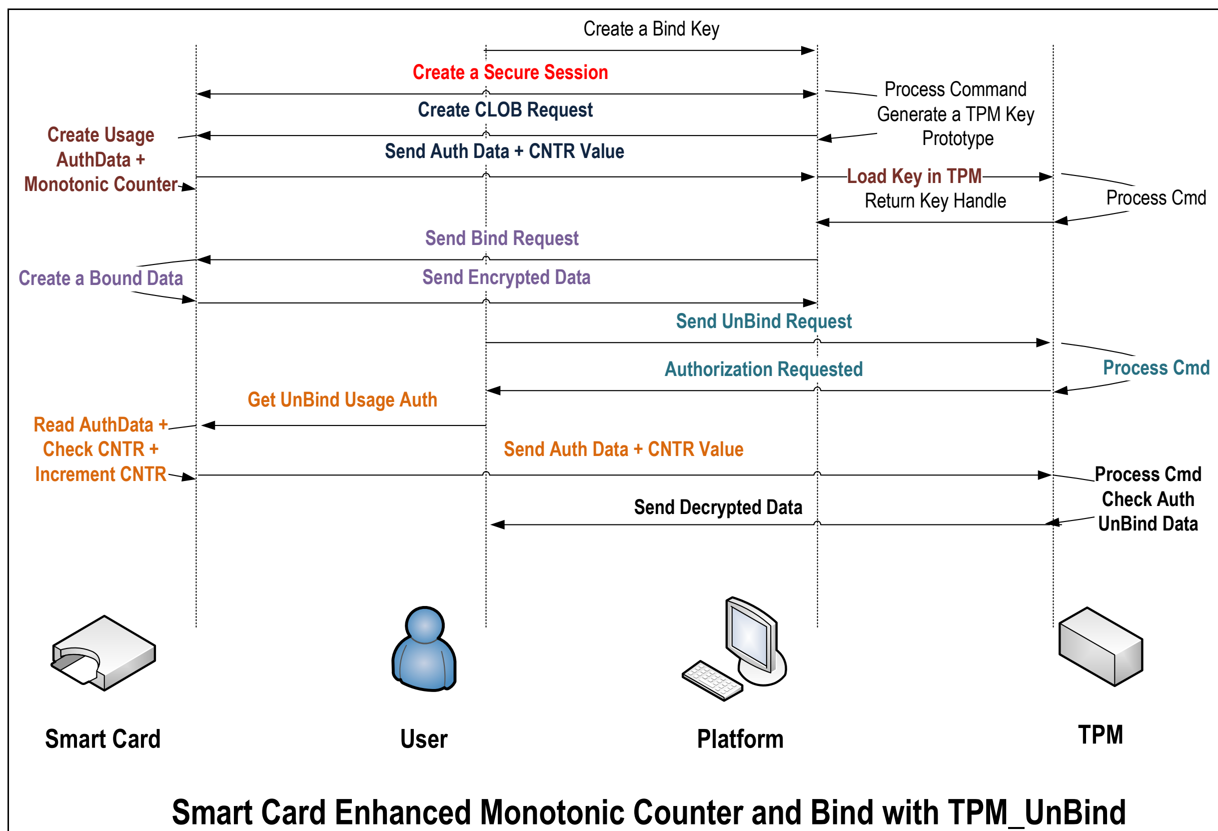


Figure 6-2 Smart Card Enhanced Flexible Sealing

```
        /// <summary>
        /// Tests the unbind from smart card.
        /// </summary>
        /// <param name="smartCardService">The Smart Card Service.</param>
public bool TestSmartCardBind(SecureSessionEstablishmentService smartCardService)
        {
            TpmRawCommands tpmCmd = TpmDevice.RawCommand;
            OsapSession osapSession = tpmCmd.Osap(TpmEntityType.KeyHandle, (int)ReservedKeyHandle.Srk,
                                                  srkAuth, false);
            osapSession.ContinueAuthSession = false;

            OiapSession srkSession = tpmCmd.Oiap(srkAuth, false);
            OiapSession session = tpmCmd.Oiap(ownerAuth, true);

            //Get from Card
            AuthorizationData useAuth = new AuthorizationData(smartCardService.GetUsageAuth());
            AuthorizationData migrationAuth = new
                AuthorizationData(Encoding.Default.GetBytes(smartCardService.GetMigrationAuth()));

            //Generate a random key .. we can do this in the card as well .. it just takes longer
            TpmKey randomGenKey = CreateBindingKeyPrototype();
            //Wrap it with SRK
            TpmKey randGenBindKey = tpmCmd.CreateWrappedKey(TpmKeyHandle.SrkHandle, useAuth,
                                                  migrationAuth, randomGenKey, osapSession);
            //Now load it in the TPM for use
            TpmKeyHandle bindingKeyHandle = tpmCmd.LoadKey(TpmKeyHandle.SrkHandle, randGenBindKey,
                                                                        srkSession);

            byte[] dataToBind = Encoding.Default.GetBytes("DataToBind");

            byte[] boundDataWithSmartCard =
                        smartCardService.CreateTPMBoundData(randGenBindKey.StorePubKey.key, dataToBind);

            byte[] scUnboundData = tpmCmd.Unbind(bindingKeyHandle, boundDataWithSmartCard, session);
            if (!ArraysAreEqual(dataToBind, scUnboundData))
            {
               throw new Exception("Unbind Error");
            }
            tpmCmd.FlushSpecific(bindingKeyHandle);
            return true;
        }
```

6.3 SOME FURTHER APPLICATIONS

The applications described above are just to give a general idea how our proposed architecture can be used to provide extended TPM services with assurance of secure execution. Some similar and connected applications that are very intuitive to our design are described below.

6.3.1 E-cash Tokens that only work with authorized platforms

One of the major problems with payment tokens is that there is no trust relationship between the card and the reader. The Roaming DRM application can be simplified to make a SmartCard / Token only work with an authorized platform. This includes payment tokens, EMV based applications, transportation cards, identity cards etc. This is very simple as we can restrict applications inside the smart card to talk to only authorized platforms for whose key or key hierarchy is already loaded in the smartcard and the smartcard can refuse to disclose secrets or even its identity if it cannot verify or build a trust relationship with the smartcard.

6.3.2 Ease of data migration between trusted platforms

We already implemented a subset of this scenario in the Roaming DRM application. As the SmartCard is able to seal / bind data to the platforms it is bound to, and unbind data with migratable keys, we can bind data when exporting from trusted platform A and unbind it using a SmartCard on another trusted platform B. This model can be extended to 'n' platforms and unlimited number of files which can be sealed from one platform to be unsealed on another trusted one.

6.3.3 Crypto schemes not supported by the TPM

One of the criticisms on the current specifications of the TPM is the strict limitations on the cryptographic algorithms and primitives supported. NIST already started advising the industry not to use SHA1 anymore because of different cryptanalysis and collision attacks on SHA-1, but the newer hashing algorithms like SHA256, SHA512 etc are not available which puts a severe limitation on security sensitive applications which have to comply with certain minimum requirements and comply with standards. With our solution, as the smart card can provide general purpose secure execution, any cryptographic primitive not supported natively by the TPM and the SmartCard can be built as an applet / onCard application and be used as a replaceable algorithm. The calls to and from the applet can be encrypted using the custom remoting and encryption sinks as described in the secure binding section 4.5. Any data can be sealed by the smart card to be exported only to the trusted platform it is bound to. Hence a TPM / platform could also encrypt any calls and data to be sent to the SmartCard specifying the crypto operation it wants the SmartCard to perform and the extended crypto algorithm operations can be carried out by the smartcard.

6.3.4 Flexible Authorization Applications

The flexible authorization described in 5.2 can be extended beyond any two factor authentication where both the presence of a token and involvement of a platform needs to be assured for a process. The applications can include IPSec / VPN authentication where user password is provided by the authorization of a smartcard and keys for encryption can be released from a TPM by a TPM/SC collaborative model hence giving a remote authentication of both a platform and the user.

6.3.5 Strengthen Remote Attestation

A number of studies have worked on more flexible remote attestation such as [**28**]. This smartcard and TPM co-operative model can help in strengthening trust in a distributed environment by implementing the Secure Binding between the SmartCards / TPM and using the SmartCards for authentication and validating TPM generated attestations given a flexible distributed attestation environment.

6.3.6 Smart Card SIM Bound with Mobile TPM

The telecom industry faces numerous challenges for protecting the state of their hardware and software applications. With increasing competitive and diverse market it is becoming very common (and with automated tools, much easy) to tamper the state and protection of any handset. It is quite common to see a partnership between an equipment manufacturer and the network operator to provide exclusive services with a new model release for e.g. [**29**]. However as soon as a new phone is hacked it is a severe damage both in terms of finance and credibility of both the network operator and the handset manufacturer plus the stakeholders.

Now consider a mobile phone equipped with a TPM. The network operator and the equipment manufacturer can create a TPM/ SIM binding for stronger DRM and enhanced services. So even if the phone is cracked and a SIM replaced with some other network operator, the enhanced operator services will not be available. This can be extended to even disable the phone completely depending upon the choice of application, regulations and other dynamics. The TPM can offer an authenticated boot operation which would fail to disclose secrets since the expected SIM and keys are not in place for authenticated binding

7.0 SUMMARY AND CONCLUSION

In this project we discussed some of the shortcomings and limitations of secure execution with the current state of the TCG (Trusted Computing Group) specifications. Though we feel that the various industry initiatives taken by the TCG and CPU manufacturers for hardware based platform security are a step in the right direction, the problem of secure isolated code execution and TCB minimization still remains unsolved. In this project we propose and implement an alternative architecture for secure code execution. Rather than proposing recommendations for hardware changes or building isolated execution environments inside a TPM (Trusted Platform Module), we use a platform that provides related, yet different services for secure / trusted code execution, couple its functionality and bind it to a TPM using cryptographic primitives. For the purpose of this study we used multi-application programmable SmartCards but similar work can also be implemented on other platforms as long as they meet some pre-requisites described in.

7.1 THE TRUSTED PLATFORM MODULE

We started with a description of the Trusted Platform Module, The Trusted Computing Group, their working models, and working groups. We discussed the motivation and history of trusted computing and how the industry players are moving together to work for making all computing platforms safer, trusted and trustworthy. We described the different components and services provided by a TPM for platform security. We discussed the ownership, activation, initialization and clearing process. Next we discussed some of the authorization protocols for accessing TPM protected resources. And finally we discussed the services provided by the TPM including attestation, sealing, quoting, counters, and other cryptographic primitives that are important for our discussion. We discussed the limitations of the execution engine provided by the TPM and discussed in general what problems the industry faces without a TCB minimized secure execution environment.

7.2 THE SMARTCARDS

In Chapter 3 we described an overview of SmartCards, how they differ from conventional computing platforms, their different types of standards and architectures, how they are being used in the industry and what special characteristics of them became the motivation for choosing them as an extended platform for the SmartCard and TPM coupling process. These included their portability, processing and storage capabilities, extensive cryptographic support, tamper resistant hardware and protected storage and finally and most importantly secure program execution capabilities. We discussed some of the application frameworks with special emphasis on the .NET platform available for SmartCards. The different layers of software were described with the responsibilities of each. These included the Operating System, Applications, Runtime environment, loader and unloader microcode provided by the card.

Later in the chapter we described the .NET SmartCard architecture with its application lifecycle, how a typical application management takes place from start to end which includes the loading, installation, execution, termination and unloading phases. We discussed the post issuance update and re-programming capabilities and how different industries utilize them for card based application management. Finally a few points were mentioned over the benefits of using a .NET based application that would be executed by a Common Language Runtime environment. A .NET application running under a managed CLR has a number of benefits that provide ease of implementation, separation of security functionality and application isolation capabilities along with numerous security offerings of the .NET framework that can be utilized directly within the applications. This helps us greatly in providing a secure execution environment for applications that are coupled with the TPM to provide enhanced services.

These smart card capabilities hence provided us with an attractive platform which has a number of similar capabilities like a TPM, and some different characteristics and capabilities that makes it different from conventional computing platforms. The secure execution, storage and process isolation capabilities provided by a general programming platform like .NET made it an ideal choice for using it to provide or extend secure execution environments. Hence we chose a roaming platform like SmartCard for extending secure execution environments of a platform-bounded security device, i.e. the Trusted Platform Module.

7.3 THE SMARTCARD AND TPM COUPLING

Next we discussed the details, architecture and design of enhanced TPM and Smart Card coupled services that were implemented in some of the major functional areas of the TCG specifications. We also discussed the differences in both TPM and the SmartCard platforms and detailed some of the major challenges that arise during the coupling process and our solutions for addressing the major security issues in coupling these architectures together. We discussed the requirements and limitations of coupling with smartcards and put forward some recommendations regarding the non supported scenarios.

In chapter 4 and 5 we described a detailed explanation of the enhanced TPM and Smart Card coupled services that were implemented in some of the major functional areas of the TCG specifications. We discussed the fundamental differences in both TPM and the SmartCard platforms. We also discussed in detail some of the major challenges that arise during the coupling process and our solutions for addressing the major security issues in coupling these architectures together. We also discussed the requirements and limitations of coupling with smartcards and put forward some recommendations regarding the non supported scenarios.

We briefly described what would be an ideal architecture and components of a TPM if an internal secure execution environment is built into the TPM itself which can provide VM based code isolation and security guarantees for applications. We discussed some of the similar projects, their propositions and their solutions for extending hardware and software to provide similar offerings for secure code execution. However we concluded that most of these projects do not focus exclusively on either providing a secure / trusted execution environment within the TPMs tamper resistant hardware boundary or put forward a solution that can be implemented without changing the current state of hardware and software based platform security.

Furthermore most of them require hardware extensions which are time consuming, require changes in the software which are complex and costly both in terms of time and money.

Hence we put forward an alternative architecture that uses the services of a TPM and couple them with a platform that does provide secure execution environments i.e. the SmartCards. In this way we utilize the already established strengths of both the platforms without requiring any additional or special hardware extensions. We discuss in detail how we achieve this target with multi-application post issuance programmable smart cards with a detailed implementation of the coupling process. We already described our motivation for choosing SmartCards as the coupling platform and its characteristics that made it a natural choice for this process earlier. The SmartCard capability requirements for different coupling levels and scenarios are clearly stated with a description and references of our chosen .NET SmartCard platform. Furthermore we also discuss the core platform differences between the SmartCard and the TPM that build up a number of challenges in the process.

For the actual implementation we start off by discussing how to cryptographically bind a TPM and a SmartCard in order for them to identify each other and provide other coupled confidentiality and integrity services. We utilize the TPMs attestation keys and SmartCard generated Identity keys for TPM and SmartCard binding and follow a model similar to SSL for generating / using session based symmetric keys for all calls to and from the TPM / SmartCard. We also extend the .NET Remoting architecture to add encryption to the channels and sinks for making an end-to-end encrypted tunnel of communication between the SmartCard and the TPM. This is important as .NET remoting does not offer any security services on its own. We also shed some light in securing the remoting component / proxy application in a large enterprise distributed environment by utilizing the established security infrastructure provided by Windows for e.g. utilizing different windows authentication methods, access control mechanisms and encryption / security features offered by the Internet Information Services Server.

7.4 ENHANCED TPM AND SMART CARD SERVICES

Next we discuss some of the enhanced services that are possible with the TPM-and-SmartCard collaborative model and discussed our detailed implementations of them. These services are broadly categorized into 4 major categories:

Flexible Authorization: Smart Card participates in the TCG authorization protocols with different methods depending upon the security level required and smartcard capability requirements. Those include from simple AuthorizationData storage in the card to complex HMAC based shared secret calculations for OSAP and ADIP protocols.

Validating TPMs Generated Quote: The smart card uses this service to validate a TPM generated attestation just like any remote challenger. This service can be used in a number of different methods and combined with other protocols depending upon the application. We use this service exposed by the SmartCard to check the authenticity and genuineness of the TPMs generated attestations. We can use this service in a number of ways, some of the applications using this service are described in Chapter 6.0 .

Flexible Sealing and Binding: We extend the simple TCG sealing and binding model so that more flexible disclosure options are possible for example Sealing to multiple TPMs and multiple configurations, Sealing to a complex logic expression based on PCRs or Sealing to any configuration authorized by a signature with a known public key etc. A SmartCard can Bind / encrypt data to any platform (or set of platforms) for which it is bound to. This also helps us build some interesting DRM applications, a couple of them are described in Chapter 6.0 .

Enhanced Monotonic Counters: The Smart Card also provides an easy, usable framework and shielding the complexity of the Monotonic counters provided by the TPM. We implement some user and programmer friendly functions for implementing an extended counter system within the smart cards using arbitrary length or 64bit counters. We bound those counters with different objects (such as keys) to provide an implementation of Count Limited Objects, hence controlling the use of keys and objects. All counter implementation functionality including the number of counters, their maximum values, the objects they are bound to etc can be controlled by the user.

Based on these enhanced services that are implemented in the SmartCard, we have built the basic building blocks that are needed for smart card and TPM coupling. Hence any general purpose application can now be developed targeting the .NET framework and executed securely inside the smartcard. Moreover since the SmartCard is coupled with a TPM using cryptographic services described earlier we can build any application that utilizes the TPM protocols and primitives as well. This model hence utilizes the security features of both the SmartCards and the TPMs to offer secure execution and other enhanced TPM services for security sensitive applications.

7.5 SOME APPLICATIONS DEVELOPED WITH THE COUPLED SERVICES

Based on the building blocks described in Chapter 4 and 5, we discuss some of the applications utilizing this extended TPM and Smart Card model and describe how these applications change the way conventional TPM and Smartcard applications were perceived.

The first major application is the Enhanced Digital Signature application. The Smart Card forces the platform to generate an attestation of the platform state before signing any document. The signing keys can also be bound to particular platforms used for signing. The SmartCard verifies the TPM Generated Attestation and signs the Document and the attestation. This signature hence proves that not only a the smartcard was used in signing, but also which particular platform was used in the signing process and what platform state it was in at the time of signing.

The second application that is described in detail was the Roaming DRM application. We call it roaming because the DRM data / protection keys can be carried to different platforms and places because the smart card is roaming. This application uses the flexible sealing, flexible authorization, enhanced monotonic counters and the count limited objects services provided by the SmartCard. Depending upon the usage the SmartCard can be used to either Seal or Bind data to chosen platforms. In case of bind, a key can be made migratable and a smartcard can also be used to Unbind data encrypted by a particular TPM (with the assumption that the decryption key of the migratable key is already pre-loaded in the card during the TPM / SC coupling process).

Some further application descriptions include crypto-schemes not supported by the TPM to be built as programmable onCard applications for SmartCards. Electronic Cash and payment tokens that can be used only with authorized platforms. Ease of data migration facilities provided by the smartcard / TPM collaboration and stronger DRM applications for mobile protection by a SIM and TPM binding for cell phones.

7.6 SIMILAR WORK

We describe in this section some similar solutions and studies that focus on solving similar problems. There are a number of interesting solutions around extending the secure environments by either extending or working around the current TCG specifications.

AEGIS [**13**] describes a single-chip architecture for a secure processor which can be used to build computing systems secure against both physical and software attacks. The AEGIS processor is the only trusted component. The trusted computing base (TCB) consists of the processor chip and optionally a part of an operating system. The trusted core part of the operating system is called the security kernel which operates at a higher protection level than other parts of the operating system in order to prevent attacks from untrusted parts of the operating system such as device drivers.

Terra [**14**] uses a trusted virtual machine monitor (TVMM) that partitions a tamper-resistant hardware platform into multiple, isolated virtual machines. The software stack in each VM can be tailored from the hardware interface up to meet the security requirements of its application. The hardware and TVMM can act as a trusted party to allow closed-box VMs to cryptographically identify the software they run. Terra allows applications to run in an "open box" VM with the semantics of a modern open platform, or in a "closed box" VM with those of dedicated, tamper-resistant hardware. The key primitive that Terra builds on is a trusted virtual machine monitor (TVMM). The TVMM mechanisms allow Terra to partition the platform into multiple, isolated VMs. Each VM can tailor its software stack to its security and compatibility requirements.

Thin Clean Client [**15**] by IBM Research proposes a modified minimum Linux distribution with trusted computing technology for a secure environment. It allows users to use a single PC for two or more purposes: an ordinary PC for everyday use, and a part-time secure environment for highly sensitive tasks, e.g. dealing with personal information. TCC is built on top of Linux which is modified to support Trusted Computing technology such as the integrity measurement and reporting by the Trusted Platform Module.

Trusted Execution Module [**30**] puts forward a high level specification for a chip that can execute user supplied procedures in a trusted environment. It introduces the concept of partially encrypted closures which express arbitrary computation logic. Anyone who knows the public key of the chip can generate these code closures. The trusted execution module specifications described in the paper differ from both the smart card and the TPM. It introduces a new different style of programming. Hence a smart card need not be pre-programmed with a limited set of domain or application specific commands.

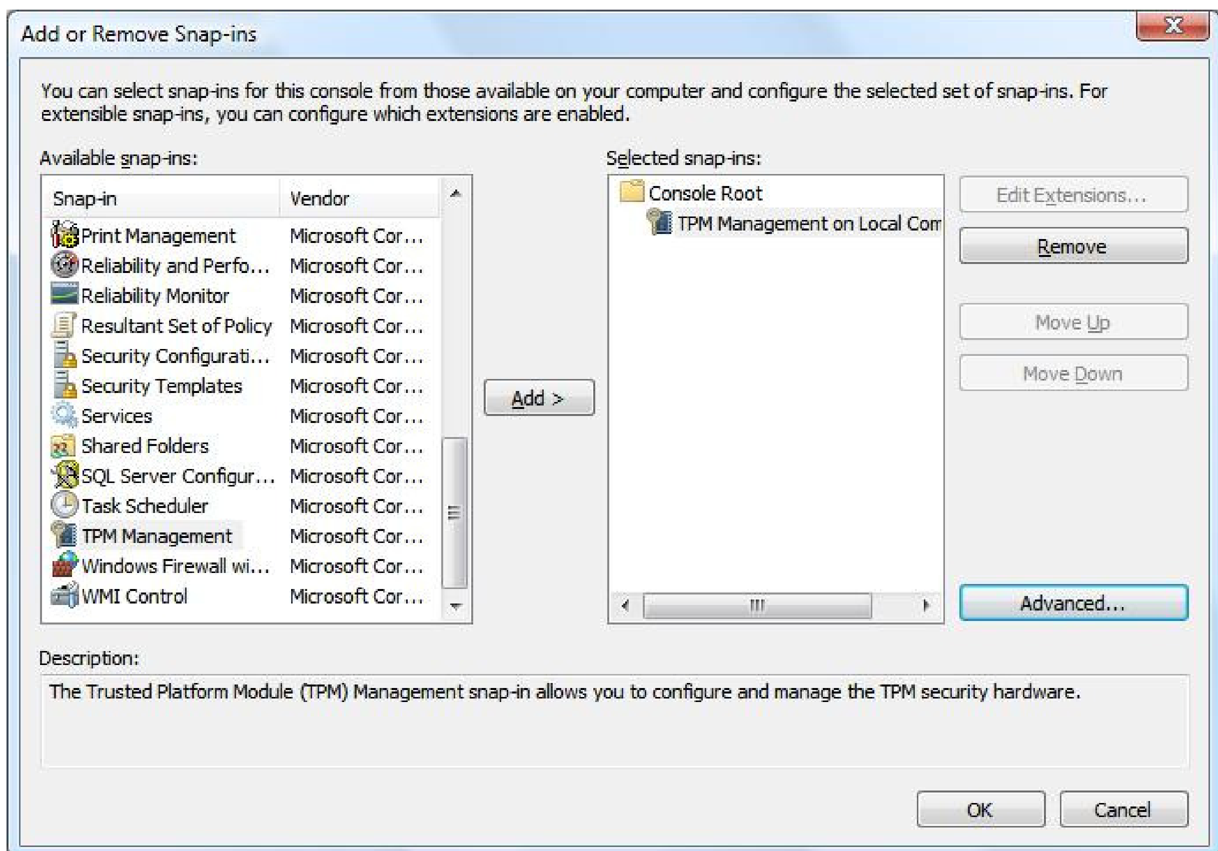
Flickr [**17**] uses the new general purpose hardware extension for secure OS loading, late launch and attestation. It provides an infrastructure for executing security sensitive code in complete isolation while trusting as few as 250 lines of additional code but requiring special hardware. Flicker leverages new processors and hardware extensions from AMD and Intel but does not require a new Operating System or Virtual Machine Monitor. It uses the Late Launch[**1**] and its Secure Virtual Machine Extensions [**31**] to provide such an environment. The exact code executed (and its inputs and outputs) are attested to an external party. For example, a piece of server code handling a user's password executes in complete isolation from all other software on the server, and the server can convince the client that the secrecy of the password was preserved.

However our solution differs from all of the above, as we do not require any changes in the current general purpose available hardware or software. And we combine the best services from already available general purpose SmartCards and Trusted platforms offering niche functionality and security features by coupling them together for all general purpose computing and security sensitive applications.

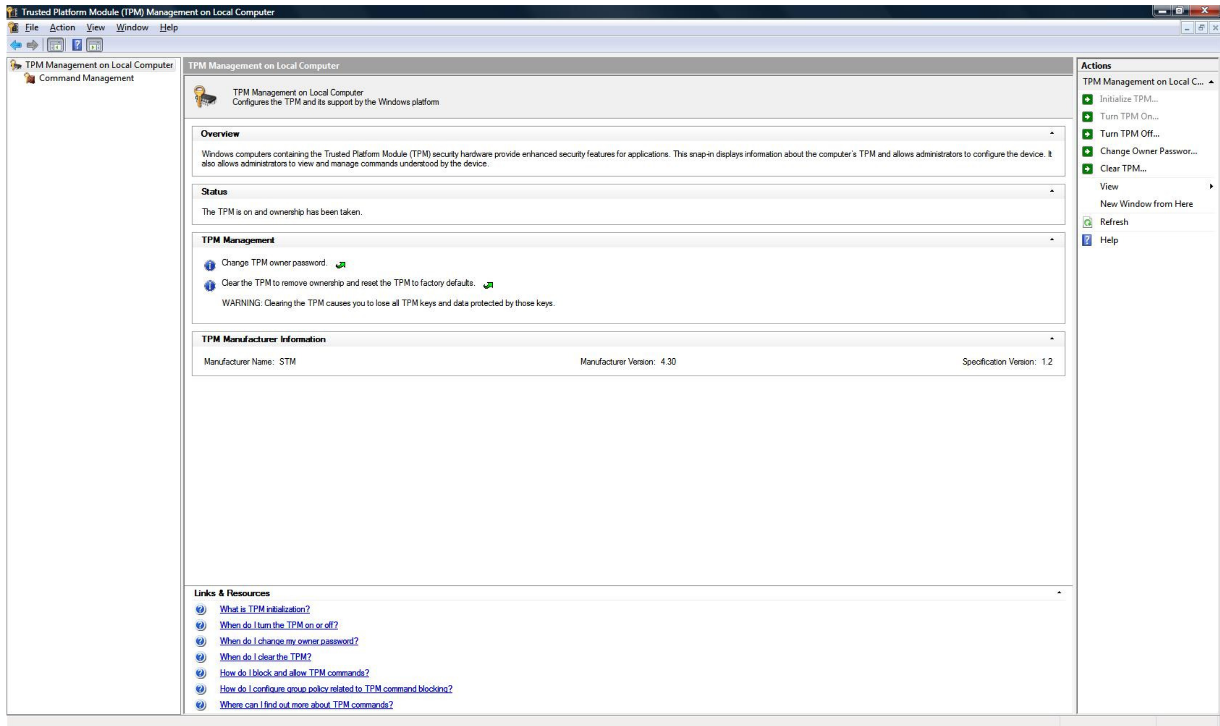
Appendix A – Using TPM Services in Windows

This appendix mentions some related functionality and administration environment for TPMs in Windows Vista. Relevant screenshots are included where suited and special instructions are mentioned which would help eliminate common caveats and problems faced for common troubleshooting.

The TPM administration in Windows Vista and Windows Server 2008 is included in the Management Console and can be accessed from typing in the command tpm.msc on the Run Dialog or running mmc (Microsoft Management Console) and adding the TPM snap-in.

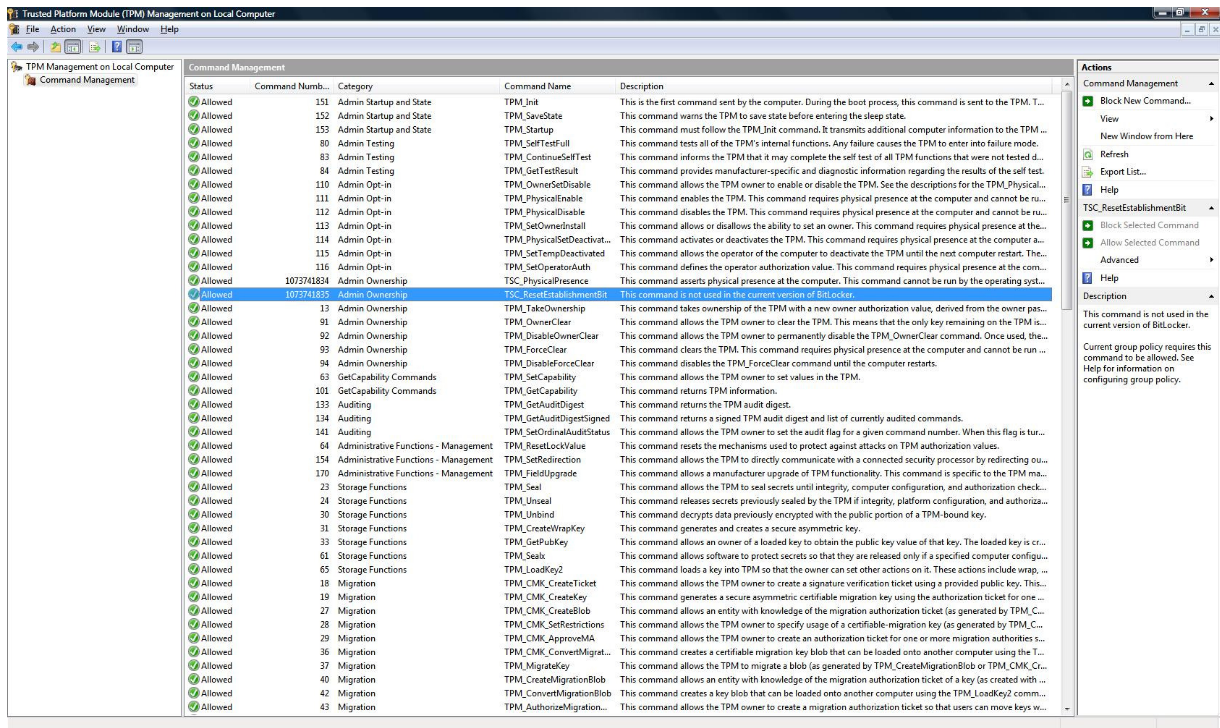


The TPM can be initialized/ cleared/ or turned on/off from this management console. It also allows for password backup.

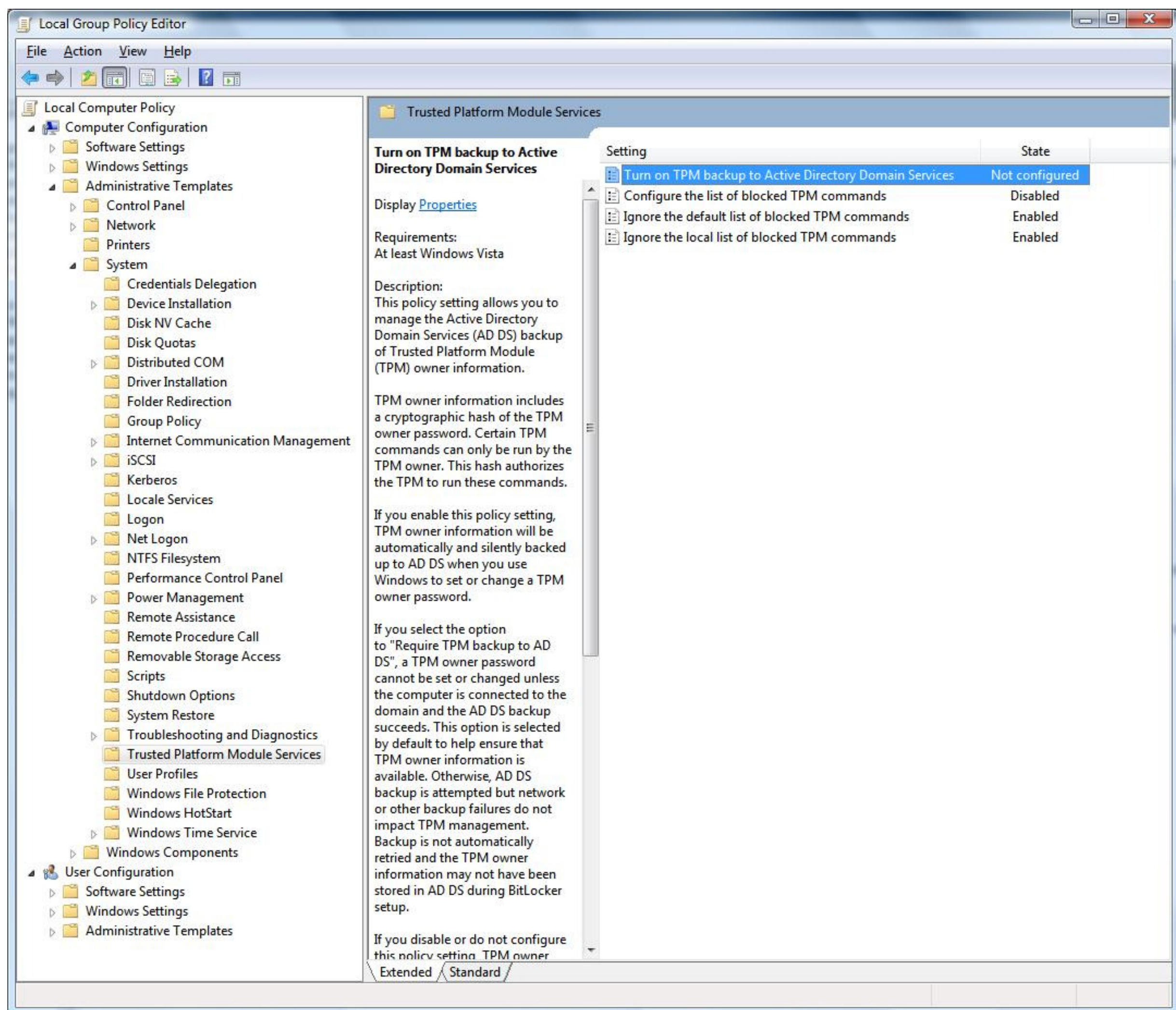


A User can take ownership of a TPM by clearing it and initializing it with a password.

The panel on the left includes the administration of the TPM Commands. Individual TPM Commands can be enabled and disabled from the snap-in as shown in the screenshot below.



Some of the commands are blocked by default and the calling those commands from the TBS API causes a blocked command exception. Before the commands are allowed from the snap-in they also need to be enabled from the local or domain group policy



For programming the TPM and utilizing it in user applications we use the TPM Base Services library[18] provided by Microsoft as part of Windows Vista and Windows Server 2008.

[18] http://msdn.microsoft.com/en-us/library/aa446796(VS.85).aspx

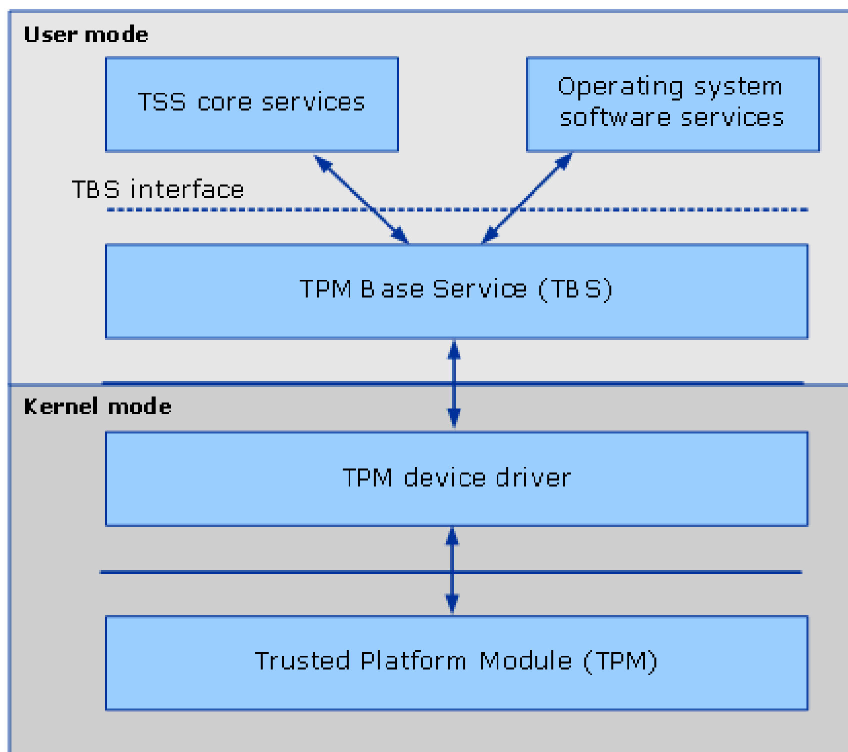


Figure 7-1 The Trusted Base Services Library [19]

[19] http://msdn.microsoft.com/en-us/library/aa446792(VS.85).aspx

Appendix B - .NET Card Specifications [8]

This appendix lists the technical details and capabilities of the .NET smart card. This would give the reader a general idea of the platform used, its capabilities and limitations. Later in the appendix are included some screenshots of the .NET Card Crypto Service Providers and Microsoft Windows built-in driver proxy, mini drivers and PS/SC capabilities built into Windows Vista.

Technical details I

Silicon features

- Infineon Chip SLE88CFX4000P (400 KB Flashmask)
- 32-bit micro-controller
- Cryptographic co-processor (faster RSA and 3-DES)
- True random number generator

Cryptographic capabilities

- RSA signature and verification up to 2048-bit keys
- DES, 3-DES (CBC, EBC), AES, HMAC, SHA1,SHA2 and MD5
- Customizable authentication framework and secure channel capabilities

Standards

- ISO 7816-1-2-3-4 (partial)
- ECMA 335 / ISO/IEC 23271 – Common Language Interface

File system

- Secure data storage
- Role-based access control
- Enables assembly and data separation
- Enables Assembly update with data preservation

Application development

- .NET compatible and programming language independent (CLI)
- 75KB expandable to 90KB memory available for applications
- Legacy compatible application development
- On-card XML parser
- Support for int-64

Security

- Off-card application verification integrated in tool chain
- On-card verifier to check type structural integrity and type safety of applications
- Only strong-name signed assembles can be loaded ensuring integrity and authenticity

Communications

- Standard I/O transfer speed up to 223 Kbps
- Negotiable PPS
- T=0 protocol
- SConnect
- .NET Remoting

The diagram below shows the hardware components of the SmartCard chip. We can see the separate crypt-processor and the mainstream CPU for general purpose execution engine all within the tamper resistant boundaries of the SmartCard.

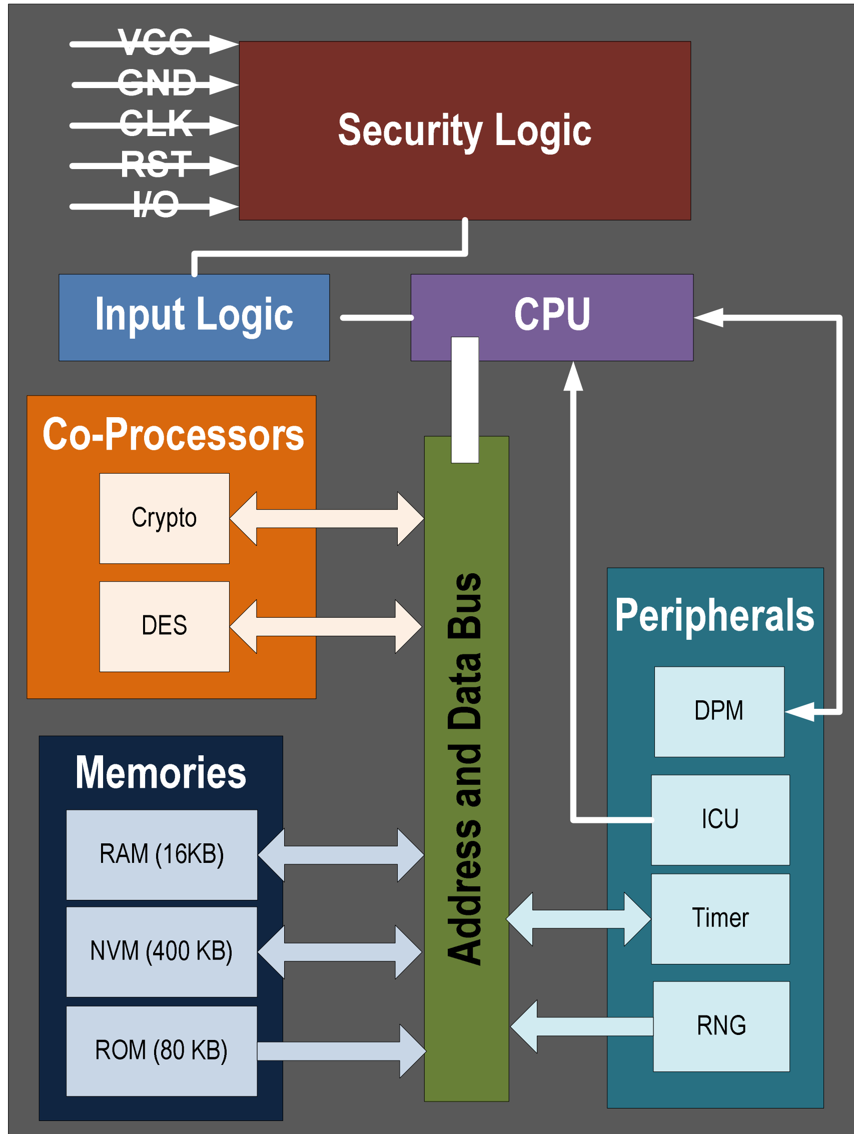


Figure 7-2 Block Diagram of the SLE88CFX2000P hardware [32]

The Diagram below shows the Microsoft Crypto Architecture for SmartCards and its built-in layered support in its Operating System and other identity products.

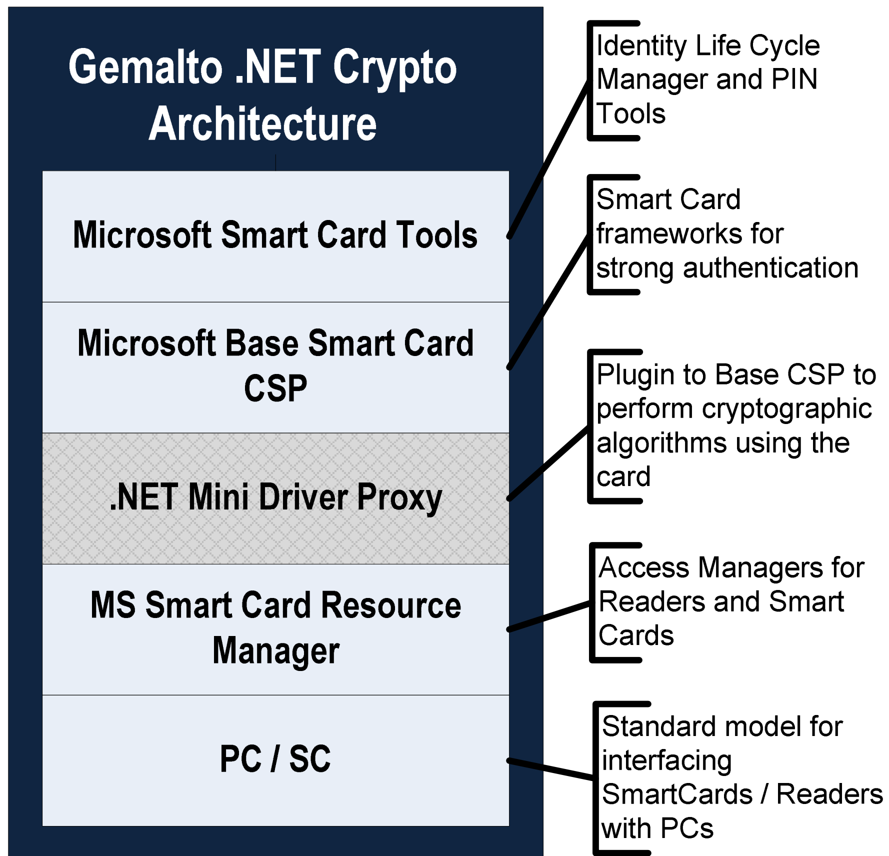


Figure 7-3 Microsoft Crypto Architecture for Smart Cards [8]

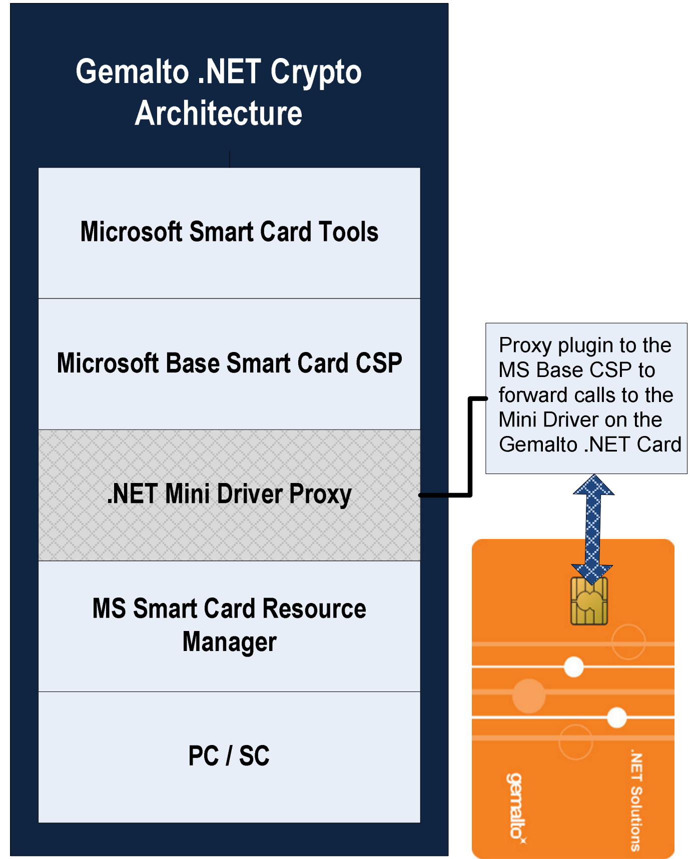


Figure 7-4 Crypto Architecture in .NET Card [8]

Appendix C – Differences between Microsoft .NET and Gemalto .NET framework [33]

This appendix lists some of the major differences between the standard .NET Compact Framework 2.0 and the framework that is implemented in the Gemalto .NET Smart Card. Since the smart card is a constrained environment, the platform framework has to be tailored to run with the processing and memory limitations in mind, yet maintaining compatibility and providing adequate security for the applications

We recommend any reader who wishes to start development on a .NET Smart Card to analyze these differences before making any efforts on developing the applications as these differences will help a developer to have an idea of the limitations and differences beforehand.

- A common language runtime (CLR) that contains the elements needed to manage applications loaded onto a Gemalto .NET Card (see .NET Smart Card Framework, Common Language Runtime (CLR) for details).
- A special upload file format optimized for Smartcard profile devices. This is an alternative that produces a much smaller (by a factor of 4) binary file than a full .NET assembly, better suited to the constraints of a smart card.
- The .NET Smart Card Framework has been adapted to accommodate the smart card memory model, in which an application is stored in persistent memory and activated when an external application talks to it.
- Floating point based types are not supported.
- Non-vector arrays (arrays with more than one dimension or with lower bounds other than zero) are not supported in .NET Smart Card Framework.
- Reflection is not supported in .NET Smart Card Framework.
- .NET Smart Card Framework supports server-side remoting only.
- The var args feature set (supports variable length argument lists and runtime typed pointers) is not supported in .NET Smart Card Framework. However, .NET Smart Card Framework supports runtime typed pointers.
- Assembly scope names are ignored in .NET Smart Card Framework, and types are identified by their name alone. Two types with the same name in different assemblies are considered to be identical. Only the method signature default calling convention is supported.
- There are no implicit types in the .NET Smart Card Framework CLR. All types are explicitly defined in the metadata loaded into the CLR. In the presence of multiple loaded assemblies, it is possible to have multiple definitions for types which might normally be implicit. However, the CLR treats these multiple definitions as if there was a single one; there is no way to distinguish if there is one definition or several.
- Asynchronous calls are not supported in .NET Smart Card Framework.
- Only BeforeFieldInit type-initializers are supported .NET Smart Card Framework; all other initializers are considered to be errors.
- Finalizers are not supported in .NET Smart Card Framework.

- New slot member overriding is not supported in .NET Smart Card Framework. The existing slot for of member overriding is supported.
- (Class Layout) Only auto layout of classes is supported in .NET Smart Card Framework. (The loader is free to lay out the class in any way it sees fit.)
- The zero init flag is not supported in .NET Smart Card Framework; local and memory pools are never initialized to zero.
- Locks and threads are not supported in .NET Smart Card Framework; therefore, any types associated with these constructs are not supported.
- The security descriptor method state is not supported in .NET Smart Card Framework.

Differences between Gemalto .NET App Domains & standard .NET Application Domains

- The ExecuteAssembly method of an AppDomain can only be executed from off the card, either through the Card Explorer or through the SmartCard.CardAccessor.CardAccessor API
- An instance of AppDomain cannot be created by an application on the card.
- If an application domain does not create a service, the application domain will be garbage collected. What this means is that if application alpha.exe does not create a service, the .exe file will remain on the card after execution, but there will be no running application domain. If alpha.exe DOES create a service, the alpha.exe application domain continues to run (even after the Main method exits) until the service is deleted.
- One application domain can communicate with another application domain indirectly by using Activator.GetObject to obtain a reference to a remoted object in the other application domain.
- An application domain can delete itself by using the static AppDomain.Unload method. An application might be interested in unloading itself if it were an application that were time or usage based (such as a coupon application), or if the application were to reach an unrecoverable situation due to a security breach.

REFERENCES

[1] D. Grawrock, *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*, 1st ed. Intel Press, 2006.

[2] C. J. Mitchell, *Trusted Computing*, C. J. Mitchell, Ed. London, UK: IEE, 2005.

[3] (2007) TCG TPM Specification Version 1.2 Revision 103. https://www.trustedcomputinggroup.org/specs/TPM/.

[4] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. V. Doorn, *A Practical Guide to Trusted Computing*. IBM Press, 2008.

[5] K. Markantonakis and K. M. Mayes, *Smart Cards, Security, Tokens and Applications*. London, UK: Springer, 2008.

[6] W. Rankl and W. Effing, *Smart Card Handbook*, 3rd ed. Wiley, 2004.

[7] I. O. f. Standardization. (1987) Identification cards - Integrated circuit(s) cards with contacts Part 1: Physical characteristics.

[8] Gemalto. (2007) .NET Solutions. [Online]. http://www.netsolutions.gemalto.com/

[9] S. Microsystems. Java Card Technology.

[10] Multos. Smart Card Application Development. http://www.multos.com/developer/.

[11] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, and W. Paul, "LestWe Remember: Cold Boot Attacks on Encryption Keys," in *Proc. 2008 USENIX Security Symposium*, 2008.

[12] V. Costan, L. F. G. Sarmenta, M. van Dijk, and S. Devadas, "The Trusted Execution Module Commodity General-Purpose Trusted Computing," in *Eighth Smart Card Research and Advanced Application Conference*, London, 2008.

[13] G. E. Suh, D. Clarke, B. Gassend, M. V. Dijk, and S. Devadas, "The AEGIS processor architecture for tamperevident and tamper resistant processing," Massachusetts Institute of Technology, 2003.

[14] G. Tal, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," in *Proceedings of the 19th Symposium on Operating System*, 2003.

[15] M. Nakamura, et al., "Thin Clean Client for an Instant Trusted Environment," in *The Second Workshop on Advances in Trusted Computing (WATC)*, Tokyo, 2006.

[16] V. Costan, L. F. G. Sarmenta, M. van Dijk, and S. Devadas, "The Trusted Execution Module Commodity General-Purpose Trusted Computing," in *Eighth Smart Card Research and Advanced Application Conference*, 2008.

[17] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys'08)*, Glasgow, 2008.

[18] D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga, "Attacking a Trusted Computing Platform. Improving the Security of the TCG Specification," Universit`a degli Studi di Milano, Milan, Technical Report, 2005.

[19] S. D. Network. Remote Method Invocation Home. [Online]. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[20] I. Rammer and M. Szpuszta, *Advanced .NET Remoting*, 2nd ed. Apress, 2005.

[21] A. X.94. (2002) Retail Financial Services Symmetric Key Management - Part 1: Using Symmetric Techniques.

[22] P. George, "User Authentication with Smart Cards in Trusted Computing, SAM '04," in *Security and Management*, H. R. Arabnia, S. Aissi, and Y. Mun, Eds. Las Vegas, USA: CSREA Press, 2004, pp. 25-31.

[23] A. Shamir and N. V. Someren, "Playing Hide and Seek with Stored Keys," in *Proceedings of the Third International Conference on Financial Cryptography*, vol. 1648, London, 1999, pp. 118-124.

[24] D. Brumley and D. Song, "Automatically partitioning programs for privilege separation," in *Proceedings of the 13th conference on USENIX Security Symposium*, vol. 13, San Diego, 2004, pp. 5-5.

[25] M. Bellare and P. Rogaway, "Optimal Asymmetric Encryption - How to Encrypt with RSA," in *Advances in Cryptology - Eurocrypt '94 Proceedings, Lecture Notes in Computer Science*, vol. Vol. 950, 1994.

[26] L. F. Sarmenta, M. Van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas, "Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS (Extended Version)," MIT-CSAIL-TR-2006-064, 2006.

[27] J.-L. Giraud and L. Rousseau, "Trust Relations in a Digital Signature System Based on a Smart Card," in *Proceedings of 23rd National Information Systems Security Conference*, Baltimore, 2000.

[28] V. Haldar, D. Chandra, and M. Franz, "Semantic Remote Attestation - A Virtual Machine directed approach to Trusted Computing," in *USENIX Virtual Machine Research and Technology Symposium*, 2004, pp. 29-41.

[29] AT&T. (2008) iPhone 3G, AT&T and Apple. [Online]. http://www.wireless.att.com/cell-phone-service/services/index.jsp

[30] V. Costan, L. F. G. Sarmenta, M. van Dijk, and S. Devadas, "The Trusted Execution Module Commodity General-Purpose Trusted Computing," in *Eighth Smart Card Research and Advanced Application Conference*, 2008.

[31] *AMD64 virtualization: Secure virtual machine architecture reference manual*. Advanced Micro Devices, May 2005.

[32] Infineon. SLE 88 family: High End Security Controller. [Online]. http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab693a350166

[33] Gemalto. (2007) .NET SmartCard API Documentation. SDK.

[34] M. E. Russinovich and D. A. Solomon, *Windows Internals*, 4th ed. USA: Microsoft Press, 2005.

[35] (2006, Jun.) Common Language Infrastructure. ECMA Standard 335.

[36] "Flexible OS Support and Applications for Trusted Computing," in *9th Workshop on Hot Topics in Operating Systems* , 2003, pp. 145-150.

[37] S. Balfe and K. G. Paterson, "Augmenting Internet-based Card Not Present Transactions with Trusted Computing," in *Proceedings of the Twelfth International Conference of Financial Cryptography and Data Security*, Cozumel, Mexico, 2008.

[38] E. Gallery and C. J. Mitchell, "Trusted Mobile Platforms," in *Foundations of Security Analysis and Design IV*, Berlin, 2007, pp. 282-323.

[39] "A trusted process to digitally sign a document," in *Proceedings of the 2001 workshop on New security paradigms*, Cloudcroft, New Mexico, 2001, pp. 79-86.

[40] B. Chen and R. Morris, "Certifying program execution with secure processors," in *9th Workshop on Hot Topics in Operating Systems*, 2003.

[41] R. Laboratories. (2002, Jun.) PKCS #1 v2.1: RSA Cryptography Standard. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf.