

THE EXPERT'S VOICE® IN SECURE COMPUTING

A Practical Guide to TPM 2.0

Using the Trusted Platform
Module in the New Age of Security

*MASTER SECURITY CONCEPTS
USING TPM 2.0*

Will Arthur and David Challener
With Kenneth Goldman

apress
open

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Apress®

Contents at a Glance

About the Authors..... xxi

About the Technical Reviewers xxiii

Acknowledgmentsxxv

Introductionxxvii

■ Chapter 1: History of the TPM 1

■ Chapter 2: Basic Security Concepts 7

■ Chapter 3: Quick Tutorial on TPM 2.0 23

■ Chapter 4: Existing Applications That Use TPMs..... 39

■ Chapter 5: Navigating the Specification 51

■ Chapter 6: Execution Environment 71

■ Chapter 7: TPM Software Stack..... 77

■ Chapter 8: TPM Entities 97

■ Chapter 9: Hierarchies..... 105

■ Chapter 10: Keys 119

■ Chapter 11: NV Indexes 137

■ Chapter 12: Platform Configuration Registers..... 151

■ Chapter 13: Authorizations and Sessions..... 163

■ Chapter 14: Extended Authorization (EA) Policies..... 217

■ Chapter 15: Key Management 249

- **Chapter 16: Auditing TPM Commands..... 263**
- **Chapter 17: Decrypt/Encrypt Sessions..... 271**
- **Chapter 18: Context Management 289**
- **Chapter 19: Startup, Shutdown, and Provisioning..... 301**
- **Chapter 20: Debugging 311**
- **Chapter 21: Solving Bigger Problems with the TPM 2.0 323**
- **Chapter 22: Platform Security Technologies
That Use TPM 2.0..... 331**
- Index..... 349**

Introduction

“Seminal!”

“Riveting! I couldn’t put it down until the last page.”

“I’m exhausted from reading this book! It kept me up three nights in a row. Where’s my Ambien when I need it?”

“The suspense was killing me. I just *had* to read it straight through!”

Although these responses to our book would be gratifying, it’s doubtful that any book on digital security will ever garner this type of reaction. Digital security is the computer equivalent of disaster insurance. Few people care very much about it or give it much thought, and everyone hates paying for it ... until a catastrophe hits. Then we are either really glad we had it or really sad that we didn’t have enough of it or didn’t have it at all.

We may sound like Chicken Little crying the “the sky is falling, the sky is falling,” but mark our words: a digital security catastrophe is headed your way. We could quote a plethora of statistics about the rising occurrence of digital security threats, but you’ve probably heard them, and, quite frankly, you don’t care, or at least you don’t care enough. It’s questionable whether any preaching on our part will make you care enough until you’re personally impacted by such a calamity, but we’ll try anyway.

When your reputation is tarnished, your finances are impacted, your identity is stolen, your physical well-being is threatened, your company’s reputation and finances are harmed, and, quite possibly, your country is overthrown, then you’ll wake up to the need for cyber security. But it might be too late then. Like people living in a flood zone, the question isn’t whether the flood is coming, but rather when the disaster will hit and whether you’ll be prepared for it. The time to buy digital-security flood insurance is now! Don’t wait until the flood hits.

A Practical Guide to TPM 2.0 can be part of your digital-security insurance policy. The TPM was designed as one of the core building blocks for digital security solutions. The November 2013 “Report to the President: Immediate Opportunities for Strengthening the Nation’s Cybersecurity” recommends “the universal adoption of the Trusted Platform Module (TPM), an industry-standard microchip designed to provide basic security-related functions, primarily involving encryption keys, including for phones and tablets.” Computers and devices that incorporate a TPM are able to create cryptographic keys and encrypt them so they can be decrypted only by the TPM. A TPM provides this limited but fundamental set of capabilities that higher layers of cybersecurity can then leverage. Today, TPMs are present in many laptop and desktop personal computers. They’re used by enterprises for tasks like secure disk encryption, but they have yet to be incorporated to any significant extent in smartphones, game consoles, televisions, in-car computer systems, and other computerized devices and industrial control systems. This needs to happen for such devices to be trustworthy constituents of the increasingly interconnected device ecosystem.

Our passion in writing this book is to empower and excite a rising generation of IT managers, security architects, systems programmers, application developers, and average users to use the TPM as the bedrock of increasingly sophisticated security solutions that will stem the rising tide of threats that are being aimed at us, our employers, and our civil institutions. Furthermore, the TPM is just plain cool. How many engineers, as children, played with simple cryptography for fun? The ability to send an encrypted message to a friend appeals to the secretive part of our human nature—the same part that enjoyed playing spy games when we were young. And besides being fun, there’s something inherently, morally right about protecting people’s assets from being stolen.

The TPM 2.0 technology can accomplish this. We believe in this technology and hope to make believers of you, our readers, as well. Our hope is that you’ll get as excited about this technology as we are and “go out and do wonderful things” with it, to paraphrase Robert Noyce, one of Intel’s founders.

Why a Book?

Technical specifications are typically poor user manuals, and TPM 2.0 is no exception. One reader of the specification claimed it was “security through incomprehensibility.” Although the specification attempts to describe the functionality as clearly as possible, its prime objective is to describe how a TPM should work, not how it should be used. It’s written for implementers of TPMs, not for application writers using TPMs.

Also, for better or for worse, the detailed operations of the TPM commands are specified in C source code. The structures are defined with various keywords and decorations that permit the Word document to be parsed into a C header file. Microsoft agreed with TCG that the source code in the specification would have an open source license and could be used to implement a TPM. However, although C can describe actions very precisely, even the best code isn’t as readable as text. One of the major purposes of this book is to interpret the specification into language that is more understandable to average software developers, especially those who need to understand the low-level details of the specification.

Many readers don’t need to understand the detailed operation of the TPM and just want to know how to use the various functions. These readers expect TSS (the TCG software stack) middleware to handle the low-level details. They’re interested in how to use the new TPM features to accomplish innovative security functions. Thus, this book is just as concerned with describing how the TPM can be used as it is with explaining how it works. Throughout the book, as features are described, use cases for those features are interwoven. The use cases aren’t complete—they describe what the TPM 2.0 specification writers were thinking about when those features were designed, but the specification is so rich that it should be possible to implement many things beyond these use cases.

Audience

In writing this book, we’re trying to reach a broad audience of readers: low-level embedded system developers, driver developers, application developers, security architects, engineering managers, and even non-technical users of security applications. We hope to encourage the broadest possible adoption and use of TPMs.

Non-technical readers will want to focus on the introductory material, including the history of the TPM (Chapter 1), basic security concepts (Chapter 2), and existing applications that use TPMs (Chapter 4). Visionaries who know what they want to accomplish but aren't themselves programmers will also benefit from reading these chapters, because knowing the basic ways in which TPMs can be used may provide inspiration for new use cases.

Engineering managers, depending on their needs and technical expertise, can go as deep as they need to or want to. We hope that executives will read the book, see the possibilities provided by TPMs, and subsequently fund TPM-related projects. When they realize, for example, that it's possible for an IT organization to cryptographically identify all of its machines before allowing them onto a network, that true random number generators are available to help seed OSs' "get random number" functions, and that weaker passwords can be made stronger using the anti-dictionary-attack protections inherent in the TPM design, they may decide (and we hope they will) to make these features easily available to everyday people.

Security architects definitely need to understand the functions provided by TPM 2.0 and, depending on the applications being developed, dive deep into how the TPM works in order to understand the security guarantees provided. Linking disparate machines or different functions to provide trusted software and networks should be possible using TPM functionality as security architects get creative. Commercial availability of this capability is long overdue.

Application developers, both architects and implementers, are a significant focus of this book. These readers need to understand the TPM from a high-level viewpoint and will be especially interested in the use cases. TPM 2.0 is feature rich, and the use cases we describe will hopefully inspire creativity in developing and inventing security applications. Developers have to know the basics of symmetric and asymmetric keys and hashes in developing their applications—not the bit-by-bit computations, which are done in the TPM or support software—but rather the types of guarantees that can be obtained by using the TPM correctly.

We also want the book to be useful to embedded system developers, middle ware developers, and programmers integrating TCG technology into operating systems and boot code. The TPM now exposes more general-purpose cryptographic functions, which are useful when a crypto library isn't available due to either resource constraints or licensing issues. We hope that low-level developers will find that this book goes as deep as they need it to and that it serves as a critical tool in interpreting the specification. Toward this end, diagrams and working code examples are used to help clarify many concepts. We expect that embedded systems will increasingly use TPMs as the cost of the technology is reduced (making cryptographic computations cheap to integrate into embedded software) and as attacks on embedded software become more active.

Roadmap

If you're new to security or need a refresher, Chapter 2 gives an overview of the security concepts required to understand the book. This chapter provides high-level knowledge of cryptography: we explain symmetric and asymmetric keys, secure hash algorithms, and how a message authentication code (MAC) can be used as a symmetric key digital

signature. This chapter doesn't delve into the underlying math used to implement cryptographic algorithms; this isn't intended as a general-purpose security or cryptography textbook, because there is no need for most TPM 2.0 developers to possess that depth of knowledge.

Chapter 3 presents a high-level tutorial on TPM 2.0 and the design rationale behind it. It begins with applications and use cases enabled by TPM 1.2, all of which are also available in TPM 2.0, and then continues by describing the new capabilities that are available with the TPM 2.0 specification. This chapter should help you understand why people are excited about the technology and want to use it in their applications and environments.

Chapter 4 describes existing applications that use TPMs (currently, mostly 1.2). We assume that many of these applications will be ported to TPM 2.0. Some are open source, some are demonstration code written by academics to demonstrate what the TPM can do, some are applications that have been around a long time and that can be linked to use TPM features, and some are generally available applications written specifically to take advantage of the TPM's capabilities.

Chapter 5 provides a high-level orientation to the TPM 2.0 specification, offers pointers to critical parts of the specification, and explores some best practices for using the specification.

Chapter 6 describes the setup and use of the execution environments available for running TPM 2.0 code examples.

Chapter 7 discusses the trusted software stack (TSS). This is presented early in the book because succeeding code examples use various layers of the TSS.

Chapter 8 begins the deep dive into TPM 2.0 functionality with a description of TPM 2.0 entities: keys, data blobs, and NV indices.

Chapter 9 discusses hierarchies.

Chapter 10 covers keys.

Chapter 11 discusses NV indexes.

Chapter 12 explores PCRs and attestation.

Chapter 13 is one of the most in-depth chapters and is crucial if you're developing low-level code or architecting systems that make extensive use of sessions and authorizations.

Chapter 14 discusses enhanced authorization.

Chapter 15 explains key management.

Chapter 16 describes the TPM's auditing capabilities.

Chapter 17 examines decryption and encryption sessions and how to set them up.

Chapter 18 describes object, sequence, and session context management and the basic functionality of a resource manager.

Chapter 19 discusses TPM startup, initialization, and provisioning. In typical usage, these occur before keys and sessions are used, but knowledge of TPM entities and sessions is a prerequisite to understanding TPM initialization and provisioning. This is why we include this chapter after the previous three chapters.

Chapter 20 presents best practices for debugging TPM 2.0 applications.

Chapter 21 examines high-level applications that could use TPM 2.0 functionality.

Chapter 22 discusses platform-level security technologies that incorporate TPM 2.0 devices into their security solutions.

Assumptions

Although this is a technology book, we have tried to assume as little about our readers as possible. Code examples use C, and a working knowledge of C is useful. However, most of the concepts stand alone, and much of the book should be comprehensible to non-programmers. Security concepts are explained at a high level, and every attempt is made to make them understandable.

Some knowledge of the TPM 1.2 and 2.0 specifications is definitely beneficial but not required. We encourage you to download the TPM 2.0 specifications from www.trustedcomputinggroup.org so that you can refer to them as you read the book.



History of the TPM

A Trusted Platform Module, also known as a TPM, is a cryptographic coprocessor that is present on most commercial PCs and servers. In terms of being present in computers, TPMs are nearly ubiquitous, but until recently they've been mostly invisible to users due to lack of compelling applications that use them. That situation is rapidly changing. With the recent awarding of Federal Information Processing Standards (FIPS) certification to various TPM designs, and recommendations from the President's Council of Advisors that the United States government begin using TPMs to defend the nation's computers, the TPM has become a strategic asset for computer owners to defend their cryptographic assets. It is still true that very few people know enough about TPMs to use them in an advantageous manner, a situation that motivated the writing of this book. This chapter introduces you to TPMs, starting with TPM 1.1b, and describes the history of TPM 2.0's predecessors.

Why a TPM?

In the 1990s, it became increasingly obvious to people in the computer industry that the Internet was going to change the way personal computers were connected, and that commerce was going to move toward this environment. This immediately led to a realization that there was a need for increased security in personal computers. When PCs were first designed, little thought was given to their security, so the hardware did not support it. Additionally, software was designed without any thought to security—ease of use was the main driving force in software development.

The computer engineers who got together to develop the first TPMs—and who were part of what came to be known as the Trusted Computing Group (TCG)—were trying to reverse this trend and create a hardware anchor for PC system security on which secure systems could be built. Cost pressures dictated that any such solution had to be very cheap. The result was a hardware TPM chip intended to be physically attached to the motherboard of a PC. The TPM command set was architected to provide all functions necessary for its security use cases, detailed in Chapter 3; but anything not absolutely necessary was moved off chip to software, to keep the cost down. As a result, the specification was hard to read, because it wasn't obvious how commands were to be used when a lot of the logic was left to software. This started a trend in unreadability that has continued through all the updates to the specification.

Hardware security is not an easy topic to begin with, and it doesn't help to have a specification that is hard to understand. A good starting place to understand this technology is the history of its development.

History of Development of the TPM Specification from 1.1b to 1.2

The first widely deployed TPM was TPM 1.1b, which was released in 2003. Even at this early date, the basic functions of a TPM were available. These included key generation (limited to RSA keys), storage, secure authorization, and device-health attestation. Basic functionality to help guarantee privacy was available through the use of anonymous identity keys, based on certificates that could be provided with the TPM; owner authorization was required to create those identity keys. A new network entity called a *privacy certificate authority (CA)* was invented to provide a means to prove that a key generated in the TPM came from a real TPM without identifying which TPM it came from.

Areas of dynamic memory inside the TPM, called Platform Configuration Registers (PCRs), were reserved to maintain the integrity of a system's boot-sequence measurements. PCRs, together with identity keys, could be used to attest to the health of the system's boot sequence. This began the building of a secure architecture based on the TPM anchor, one of the key aims of the TCG.

One specific non-goal of the TCG was making the TPM design immune to physical attacks. Although such capabilities are possible, it was decided to leave physical protections to the manufacturers as an area where they could differentiate. Any software attacks are within scope for TPM-based security, however.

In a manner similar to smart-card chips attached to the motherboard, IBM PCs were the first to use TPMs (similar security coprocessors had been used in mainframe computers for decades). HP and Dell soon followed suit in their PCs, and by 2005 it was difficult to find a commercial PC that did not have a TPM.

One drawback of TPM 1.1b was incompatibilities at the hardware level. TPM vendors had slightly different interfaces, requiring different drivers, and package pinouts were not standardized. TPM 1.2 was developed from 2005–2009 and went through several releases. Its initial improvements over 1.1b included a standard software interface and a mostly standard package pinout.

The TCG realized that although TPM 1.1b protected keys against attackers who did not know a key's authorization password, there was no protection against an attacker trying one password after another in an attempt to guess a correct password. Attackers who do this usually try passwords from a dictionary of common passwords; this is known as a *dictionary attack*. The TPM 1.2 specification required that TPMs have protection against dictionary attacks.

Privacy groups complained about the lack of implementations of privacy CAs. This led to the inclusion in TPM 1.2 of new commands for a second method of anonymizing keys to help address this concerns—direct anonymous attestation (DAA)—and a method of delegating key authorization and administrative (owner-authorized) functions.

It turned out that shipping a machine with a certificate for its TPM's endorsement keys on the hard disk was in many cases impractical, because IT organizations often erased the hard disk when they received it and installed their own software load. When they did so, the certificate was deleted. In order to provide a solution, a small amount of nonvolatile RAM (usually about 2KB) was added in TPM 1.2; it had specialized access controls along with a small number of monotonic counters.

The 1.1b specification had a means of copying migratable keys from one TPM to another TPM in case a machine died or needed to be upgraded. This process required the approval of the key owner and the TPM owner. It was designed with the assumption that an IT administrator would be the TPM owner and the user would be the key owner. But in 1.2, the user needed to be able to use the TPM owner authorization to mitigate dictionary attacks and create NVRAM, which made this design impractical. Therefore, in 1.2, another technique was designed to let users create keys that could only be migrated by a designated third party. Such keys could be certified to this effect and hence were called Certified Migratable Keys (CMKs).

Signing keys are often used to sign contracts, and having a timestamp of when the signing takes place is useful. The TCG considered putting a clock into the TPM, but the problem was that the TPM loses power whenever the PC is turned off. Although putting a battery in the TPM is possible, it is unlikely that the increased function would be worth the higher cost. Therefore the TPM was given the ability to synchronize an internal timer with an external clock and then to sign with the value of the internal timer. As an example, this combination could be used to determine when a contract was signed. This functionality also lets the TPM be used to distinguish how much time elapsed between two signature operations performed by the TPM.

No changes were made to existing application programming interfaces in 1.2, which preserved binary compatibility of software written to the 1.1b specification. A side effect was that the TPM 1.2 specification became even more complex, because special cases had to be used to maintain compatibility.

Before TPMs became ubiquitous, security coprocessors such as smart cards were used by some applications to store keys to identify users and keys to encrypt data at rest. TPMs are well equipped to take over this task. But they can do much more: because the security coprocessor is integrated onto the system's motherboard in the form of a TPM, it has additional uses (such as device identification) that are detailed in Chapter 3.

TPM 1.2 was deployed on most x86-based client PCs from 2005 on, began to appear on servers around 2008, and eventually appeared on most servers. Just having hardware does nothing, however—software needs to use it. In order to make use of the TPM hardware, Microsoft supplied a Windows driver, and IBM open sourced a Linux driver. Software began to be deployed, as described in Chapter 4.

How TPM 2.0 Developed from TPM 1.2

In early 2000, when the TCG was faced with the choice of a hash algorithm, it had two choices: MD5, which was most widely deployed; and SHA-1, which was stronger and was deployed widely, although not as widely as MD5. SHA-1 was the strongest commercial algorithm at the time and could feasibly be used in a small, cheap package. This was a fortunate choice, because MD5 weaknesses became apparent shortly afterward.

Around 2005, cryptographers published the first significant attack on the SHA-1 digest algorithm. The TPM 1.2 architecture relied heavily on the SHA-1 algorithm and had hard-coded SHA-1 everywhere. Although an analysis of the attack showed that it did not apply to the ways SHA-1 was used in the TPM, a common axiom in cryptography is that cryptographic algorithms only become weaker over time, never stronger. The TCG immediately began work on a TPM 2.0 specification that would be agile with respect to digest algorithms. That is, the specification would not hard-code SHA-1 or any other algorithm, but rather would incorporate an algorithm identifier that would permit design of a TPM using any algorithm without changing the specification. With this change (and other changes allowing all cryptographic algorithms to be agile), the TCG hoped the new specification could be the last major TPM specification to be released.

The original mandate of the TPM 2.0 Work Group within the TCG was only that: digest agility. However, even a cursory look at the TPM 2.0 specification shows that it's far more than TPM 1.2 plus an algorithm identifier. How did that happen?

TPM 1.1b had carefully crafted structures so that serialized versions (the structures translated into byte streams) were compact enough to be encrypted with a 2,048-bit RSA key in a single encryption. That meant there were only 2,048 bits (256 bytes) to work with. No symmetric encryption algorithms were required in the design, which kept the cost down and avoided problems when exporting designs that could do bulk symmetric encryption. RSA was the only required asymmetric algorithm, and performance required that structures be encrypted in one operation.

TPM 2.0 had to provide for digests that were larger than SHA-1's 20 bytes, so it was clear the existing structures were too large. An RSA key could not directly encrypt the serialized structures in one operation. Using multiple operations to encrypt the structures in blocks was impractical, because RSA operations are slow. Increasing the size of the RSA key would mean using key sizes that were not widely used in the industry and would also increase the cost, change the key structures, and slow the chip. Instead, the TPM Work Group decided the specification had to use the common practice of encrypting a symmetric key with the asymmetric key and the data with the symmetric key. Symmetric key operations are well suited for encrypting large byte streams, because they are much faster than asymmetric operations. Symmetric key encryption thus removed the barrier on the size of structures. This freed the specification developers to use their creativity to architect several functions that were distinct from TPM 1.2.

It is said that the best designs come about when the architects make a design and then, having learned all the problems they will encounter, throw away the first design and start over with a second. TPM 2.0 gave the architects the opportunity to do this. However, they still had to make sure that the opportunities for software development that were enabled by the TPM 1.2 design were not lost with the new architecture.

History of TPM 2.0 Specification Development

The specification made slow but steady progress for several years, with features being debated, added, and deleted. David Grawrock of Intel was the chair of the specification committee; under his leadership, the group selected the major features of the design and settled on a basic feature set and high-order design. At this point, the committee decided to change all the structures to allow for algorithm independence in the specification—this is called *algorithm agility*. All the authentication techniques were unified with

a technique originally called *generalized authorization* and now called *enhanced authorization* (EA). This increased the flexibility of authorization while simultaneously reducing the cost of the implementation and reducing the cognitive difficulty of understanding the specification. All objects and entities use the same authentication techniques. Many discussions took place regarding the problems created by having algorithm flexibility while still allowing a user to determine precisely what algorithms were used, both by a given key and also to protect the key, so the overall security of any key held by the TPM could be determined.

When Grawrock left the chairmanship due to changing responsibilities at Intel, Microsoft contributed a full-time editor, David Wooten, and HP took over the chairmanship. It was decided at this point that the specification should be compilable, which drove Wooten to create an emulator while writing the specification. A compilable specification has the advantage of much-reduced ambiguity: if there is doubt about how the specification is supposed to work, it can be compiled into an authoritative emulator. Additionally, the generalized authentication structure was moved from Polish notation (such as used in a TI calculator) to Reverse Polish notation (such as used in an HP calculator), which made implementing the specification easier (but made understanding the specification harder). The committee decided to add multiple key hierarchies to accommodate different user roles.

Wooten worked tirelessly to develop an implementation of the specification and provided strong leadership that drove the specification to its current feature set. When HP's Graeme Proudler stepped down from the chairmanship, David Challener of Johns Hopkins Applied Physics Laboratory formed a joint chairmanship first with Julian Hammersly of AMD, and later with Ari Singer of DMI. Kenneth Goldman (of IBM) took over the editorship from David Wooten after the first release, reprising a role he held for many years with the TPM 1.2 specification.

As new members joined the group over the years and began trying to understand the specification, some of them, notably Will Arthur and Kenneth Goldman, dove deep into the specification line by line. They submitted many bug and readability fixes to the TPM Work Group, and most of those resulted in changes to the specification that enhanced its consistency and readability. Even with these changes, it still is not easy reading, which led to the original impetus for this book.

Summary

The TPM specification has been developed twice. The first time, it developed from 1.1b to 1.2, evolving to incorporate capabilities as they came to be known to the specification committee. This feature-creep form of evolution made the final specification very complicated. In the second generation, TPM 2.0, after the cryptographic weaknesses of SHA-1 caused the need for a change, the architecture was redesigned from scratch—resulting in a much more integrated and unified design. The next chapter introduces the cryptographic concepts that will be used throughout the rest of the book. A good high-level understanding of these is imperative for you to understand TPM 2.0.



Basic Security Concepts

This chapter provides an overview of security concepts and their application to the TPM. It is not important that you understand the underlying mathematics of those concepts, so they will not be described. Instead, the chapter discusses the behavior of the various cryptographic algorithms so you can understand how they are used in the TPM 2.0 specification.

Security experts can skip this chapter. If you have less or somewhat rusty knowledge, you are advised to skim it to refresh your memory or learn how basic cryptographic concepts are used in TPMs. If you have little or no TPM knowledge, this chapter is a must read.

All the cryptographic algorithms included in TPM 2.0 are based on public standards that have been extensively reviewed. Creating custom algorithms that are cryptographically strong has generally proven to be very difficult. Many years of cryptanalysis must be performed before an algorithm is considered strong by the community, and only algorithms that have met that criteria are approved for use in the TPM by the Trusted Computing Group (TCG). This chapter describes three types of these algorithms: hash algorithms used mostly for integrity, symmetric-encryption algorithms used mostly for confidentiality, and asymmetric-encryption algorithms used mostly for digital signatures and key management. Specifically, we explore secure hash algorithms the Advanced Encryption Standard (AES); and two asymmetric standards, RSA and elliptic curve cryptography (ECC). But before considering the actual algorithms, you need to know what they are used to defend against.

The chapter begins with a description of the two attack classes: brute force and cryptanalysis. It then defines some fundamental concepts: messages, secrecy, integrity, authentication, and authorization, along with two higher-layer concepts, anti-replay and nonrepudiation. It finishes with a listing of the cryptographic algorithm classes used in the TPM.

For the most part, these are general security principles. They are used throughout the TPM design, and a description of their specific application to TPM 2.0 will be given as they are used throughout the book. In the few cases where the TPM uses cryptography in a less common way, or where the specification introduces a new cryptographic term, we explain it here. (The *extend* operation is an example of both cases: it's a general security concept that has been applied in a new way in TPMs.)

Everything done in a TPM is related in some way to mitigating cryptographic attacks.

Cryptographic Attacks

Cryptography is all about preventing attackers from doing malicious things. Security systems that use cryptography are designed to prevent bad people from having their way with your data, impersonating you, or changing documents without being detected. These attackers may attempt to compromise the security of a cryptographic design in two basic ways: either by deeply understanding the mathematics of the algorithms and protocols and using that knowledge to look for and exploit a flaw in the design, or by using brute force.

If you use well-vetted algorithms and protocols, and use them the way they were designed to be used, your design will *probably* be immune to the first type of attacks. That is why you need to take care to use accepted algorithms and protocols in your design. In a brute-force attack, the attacker tries every possible key, input, or password, trying to guess the secrets used to protect the design.

Brute Force

Cryptographers don't like to claim that anything is impossible. Rather, they say something is "computationally infeasible," meaning it would take an impractical amount of time to attack the cryptography by trying every combination. For the Data Encryption Standard (DES), 2^{56} possible keys could be used. This number is large: 72,057,594,037,927,936. Although breaking the DES algorithm might seem to be computationally infeasible, in 1998, a machine was developed that did exactly that.¹

Passwords are also often attacked using brute force—first dictionary words are tried, then combinations of words and numbers, and even special characters. Attackers may even try Klingon words or words from the Harry Potter books or *The Lord of the Rings*! RainbowCrack (www.project-rainbowcrack.com/) is a well-known program used to crack even fairly long passwords using brute force.

A cryptographic algorithm is well designed if the strength of the algorithm is not dependent on keeping the algorithm secret. The algorithm is infeasible to break if this is true and if it is infeasible to guess the algorithm secret, commonly called a *key*. Defending against brute force attacks is about picking key sizes so large that it is infeasible to try them all, or reducing the number of attacks that can be performed in a period of time.

TCG assumes that all the algorithms it approves are well designed. This may not be true. Because the specification is not wedded to a particular algorithm, if one is found to be weak at some point in the future, then instead of the specification having to be rewritten, just that algorithm can be removed from the list of approved algorithms.

The strength of an algorithm that is well designed is measured by its immunity to mathematical attacks. The strength of an implementation depends on both the type of algorithm and the size of the key used.

¹http://en.wikipedia.org/wiki/Data_Encryption_Standard#Chronology.

Calculating the Strength of Algorithms by Type

Symmetric algorithms are used for traditional encryption, where the same key is used for encryption and decryption. For a well-designed symmetric algorithm, the strength of the cryptographic algorithm should depend exponentially on the size of the key used. Thus if a key is only 4 bits long, there are 2^4 or 16 possible keys. A brute-force attacker who tried all 16 keys would break the encryption; but on average the attacker would have to try only half of them, or 8 keys, before finding the correct key. Because the numbers get large quickly via exponentiation, the strength of algorithms is generally quoted in bits. A well-designed symmetric algorithm has a strength equal to the number of bits in its key. In a TPM, the symmetric algorithms usually have key sizes of 128, 192, or 256 bits.

A *secure hash algorithm* is sort of like an algorithm that encrypts but cannot decrypt. This may sound non-useful, but it has a number of very interesting uses. A secure hash algorithm always produces the same size output, independent of the input. A wonderful property of a secure hash algorithm is that given the input, you always get the same output; but given just the output, you can't calculate the input. The strength of such an algorithm can be calculated two ways:

- The number of tries that would have to be attempted to guarantee finding an input that produces a given hash output. For a well-designed hash algorithm, this is assumed to be the size of the output in bits.
- The number of tries that would have to be attempted to have a 50% chance of finding two inputs with exactly the same output. For a well-designed hash algorithm, this is half the size of the output in bits.²

Depending on how the secure hash is used, either can be correct; but because cryptographers tend to be P3 people (Paid Professional Paranoids), the latter is generally used for the strength of a well-designed secure hash algorithm.

Asymmetric algorithms are strange at first introduction: The encryption algorithm is different from the decryption algorithm, and the two use different keys, which together form a public and private key pair. There are two asymmetric algorithms you should be concerned about and that are described later in this chapter: RSA (named after its inventors, Rivest, Shamir, and Adleman) and Elliptic Curve Cryptography (ECC).

For asymmetric algorithms, it is difficult to calculate strength corresponding to a particular key size. For RSA, there are tables you can consult. Those tables say that 2048 bits in an RSA asymmetric key corresponds to 112 bits in a symmetric key; 3,076 bits corresponds to 128 bits; and 15,360 bits corresponds to 256 bits of key strength. For ECC, the strength is considered to be half the number of bits of its key's size. Therefore, a 256-bit ECC key is the same strength as a 128-bit symmetric key, and a 384-bit ECC key corresponds to a 192-bit symmetric key.

²This kind of attack is generally called a *birthday attack*, because of an old party trick. If there are 23 (which is close to the square root of 365) people in a room, the chances of 2 of them having the same birthday is 50%. If there are substantially more people in the room, the probability rises accordingly. If there are 40 people, the probability is almost 90%.

If brute-force attacks are infeasible due to a large key size, the attacker may seek to analyze the mathematics of the cryptographic algorithm or protocol with the hope of finding a shortcut.

Attacks on the Algorithm Itself

Cryptographic algorithm design is a bit of an art. The mathematics are based on how hard it is to solve a particular type of problem, and that difficulty in turn is based on current knowledge. It's very difficult to design an algorithm against which no attacks can ever be mounted.

An attack on the SHA-1 hash algorithm³ was one of the motivations for moving from TPM 1.2 to TPM 2.0. Under normal circumstances, a brute-force birthday attack on SHA-1 would take about 2^{80} calculations, with a cryptographic strength of 80. The attack, which was based on a weakness in the underlying mathematics, successfully reduced the number of calculations required for a successful attack to 2^{63} —a cryptographic strength of 63. TPM 1.2 used SHA-1 throughout the design. With 56-bit DES encryption defeated by brute-strength attacks in 1998, it was clear that 63 bits was not enough for the industry. For that reason, TPM 2.0 removed this dependency on the SHA-1 algorithm. To defend against such an attack ever happening again, the specification was made *algorithm agile*—algorithms can be added to or subtracted from the specification without requiring that the entire specification be rewritten.

To summarize, in order to be secure, cryptographic algorithms must not have the following vulnerabilities:

- *Weaknesses in algorithms:* You can avoid weak algorithms by using well-vetted, internationally accepted, widely reviewed standards.
- *Brute-force attacks:* By choosing large key sizes and by allowing the end user to pick the key size they wish to use, you can avoid this vulnerability. Today 128 bits is generally considered a safe value for symmetric algorithms, but some researchers and security agencies insist on 192 bits.

Now that you've seen the attacks you're defending against, we can discuss the basic cryptography constructs used in the TPM specification. Let's begin with some definitions.

Security Definitions

Several concepts are important for understanding the TPM architecture and cryptographic concepts. People often equate security solely with secrecy: the inability of an attacker to decode a secret message. Although secrecy is certainly important, there is much more to security. It's easiest to understand these concepts by considering an

³Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, "Finding Collisions in the Full SHA-1," *Advances in Cryptology-CRYPTO 2005*.

example. Because electronic business was a big motivator in the design of the TPM, the following example comes from e-business.

An electronic order is transmitted from a buyer to a seller. The seller and buyer may want to keep details (credit card numbers, for example) of the purchase secret. However, they may also want to ensure that the order really came from the buyer, not an attacker; that the order went only to the seller; that the order wasn't altered in transit (for example, by changing the amount charged); and that it was sent exactly once, not blocked or sent multiple times. Finally, the seller may wish to verify that the buyer is permitted by their company to buy the item and to spend the total amount of the purchase order. All these aspects are the problems that cryptography and security protocols attempt to solve.

Based on this example, we can describe several commonly used security terms and concepts, and then explain how they can be used to provide the various aspect of security.

- *Message*: An array of bytes sent between two parties.
- *Secrecy*: A means of preventing an unauthorized observer of a message from determining its contents.
- *Shared secret*: A value that is known to two parties. The secret can be as simple as a password, or it can be an encryption key both parties know.
- *Integrity*: An indication that a message has not been altered during storage or transmission.
- *Authentication*: A means of indicating that a message can be tied to the creator, so the recipient can verify that only the creator could have sent the message.
- *Authorization*: Proof that the user is permitted to perform an operation.
- *Anti-replay*: A means of preventing an attacker from reusing a valid message.
- *Nonrepudiation*: A means of preventing the sender of a message from claiming that they did not send the message.

Let's consider how each of these security concepts fits into the electronic purchase order example. The message is the number of items ordered and any confidential customer information, such as a credit card number. Integrity ensures that the order has not been altered in transit—for instance, from 3 items to 300 items. Authentication proves that the order came from the buyer. Authorization checks that the buyer is permitted to purchase the items on behalf of their company. Anti-replay prevents the attacker from sending the buyer's message again to purchase three items multiple times. And nonrepudiation means the buyer can't claim they never ordered the items.

To provide these security guarantees, designers of a security system have a toolbox of cryptographic functions that have been developed, analyzed, and standardized. Some items are fundamental mathematical building blocks, such as the SHA-256 secure hash algorithm or the RSA asymmetric-key encryption calculation. Other items, such as digital signatures, build on these fundamentals by using the RSA algorithm. These cryptographic functions are described next.

Cryptographic Families

Trust us, there's no math in this section. We won't be describing prime number algorithms and elliptic curves. But it's important to understand some cryptographic operations and how they relate to the basic security principles you've already seen.

A secure hash algorithm is used to provide integrity. It can be combined with a shared-secret signing key in an HMAC algorithm to ensure authentication. HMAC is in turn the basis for a cryptographic ticket and key-derivation functions. A shared secret provides secrecy when used in symmetric-key encryption. A nonce provides anti-replay protection. An asymmetric key used as a signing key offers nonrepudiation. The TPM also uses an asymmetric key for secrecy in some protocols. All these concepts are described in the following sections.

Secure Hash (or Digest)

Most computer science students are familiar with hashes; simple hashes are used to speed searches. A more advanced form of a hash is a checksum that is used to detect random errors in data. But cryptographers are concerned with malicious attackers trying to break a system, so they need a secure cryptographic hash with very specific properties.

A cryptographic hash, like its much simpler cousins, takes a message of any length and compresses it to a hash of fixed length. For example, a SHA-256 hash is 256 bits or 32 bytes. For security purposes, the important properties of a secure hash are as follows:

- It's infeasible, given a message, to construct another message with the same hash.
- It's infeasible to construct two messages with the same hash.
- It's infeasible to derive the message given its hash.

As an example, you can observe that even a very small change in a message causes a large change in the digest produced by the hash. For example, using SHA-1, the message "Hello" hashes to:

```
fedd18797811a4af659678ea5db618f8dc91480b
```

The message "hello" with the first character changed to lowercase hashes to:

```
aa5916ae7fd159a18b1b72ea905c757207e26689
```

The TPM 2.0 specification allows for a number of different types of hash algorithms—SHA-1, SHA-256, and SHA-384 are just some of them. Typically, TPMs implement only a few of the allowed hashes. One problem that vexed the developers for a long time was how to integrate multiple hash algorithms (which are used to maintain integrity) if one of those hash algorithms was later broken. This is harder than it sounds, because usually hash algorithms themselves are used to provide integrity to reports, and if the hash algorithm can't be trusted, how can you trust a report of which hash algorithm is being used? The design that was chosen managed to avoid this problem: tags are used throughout the design in data elements that identify the hash algorithms used.

In the TPM, a secure hash is a building block for other operations, such as hash-extend operations, HMACs, tickets, asymmetric-key digital signatures, and key-derivation functions, all described next.

Hash Extend

The term *extend* is not a common cryptographic term, but this operation is used throughout the TPM. An extend operation consists of the following:

1. Start with an existing value, A.
2. Concatenate another value, B (the value to be extended) to it, creating the message, A || B.
3. Hash the resulting message to create a new hash, hash(A || B).
4. This new hash replaces the original value, A.

The whole process can be summarized as: $A \leftarrow \text{hash}(\text{original } A \parallel B)$.

As an example, using SHA-1, extending the digest of the message 'abc' to an initial value of all zero yields this result:

```
ccd5bd41458de644ac34a2478b58ff819bef5acf
```

The extend value that results from a series of extend operations effectively records a history of the messages extended into it. Because of the properties of a secure hash, it is infeasible to back up to a previous value. Thus, once a message is extended, there is no “undo” to go backward—to reverse the calculation and erase past history. However, the actual size of the value, such as 32 bytes for the SHA-256 hash algorithm, never changes, no matter how many messages are extended. The fixed length of an extend value coupled with its ability to record a virtually unlimited history of extend operations is critical in the memory-constrained TPM.

Extend is used to update platform configuration register (PCR) values. A PCR is a TPM register holding a hash value. The values extended into the PCR can represent the platform state. Suppose the PCR indicates an untrusted state, but an attacker wants to change the PCR to a trusted value. To do this, the attacker would have to construct another message, starting with the current PCR value, whose resulting hash was a trusted value. The properties of a secure hash dictate that this is infeasible.

Extend is also used in TPM audit logs. The audit logs record TPM commands and responses. Because of the extend properties, an item cannot be removed from the log once it has been added, and the size of the log in the TPM remains constant no matter how many commands are audited.

In addition, extend is used in creating policies that represent how a TPM can be authenticated. This is described in the chapter on extended authorization.

Hashes are also used in a simpler form of authorization, which was also used in the TPM 1.1 and TPM 1.2, called an HMAC.

HMAC: Message Authentication Code

An HMAC is a keyed hash. That is, it performs a secure-hash operation but mixes in a shared secret key, the HMAC key. Because of the properties of a secure hash, given a message, only a party with knowledge of the HMAC key can calculate the result. Applying a key to a message in this way is known as “HMACing” the message.

TPM 1.2 used HMACs throughout to prove to the TPM that the user knew a TPM entity’s authorization data, which was used as the HMAC key. TPM 2.0 can also authorize entities this way. The HMAC key is a shared secret between the TPM and the caller. As an example, a TPM object, such as a signing key, may have an associated authorization value that is known to both the TPM and an authorized user of the key. The user constructs a TPM command to use the object and calculates an HMAC over the command message using an HMAC key that is derived in part from the object’s authorization value. The TPM receives the command message and performs the same HMAC operation using the same HMAC key. If the results are the same, the TPM knows the command has not been altered (integrity) and that the caller knew the object’s HMAC key, which in turn means the caller knew the authorization value. Thus the caller was authorized.

The TPM also uses HMAC for integrity of structures that may at times be stored externally—in other words, proof that an attacker has not altered a value. Here, the HMAC key is a secret known only to the TPM, not shared with any party outside the TPM. When the TPM outputs a value to be stored for later use, it includes an HMAC using its secret. When the value is later input, the TPM verifies the HMAC to detect any external alteration of the value. Essentially, the HMAC key is a shared secret between the TPM and itself across a time interval: from the time the value is stored externally to the time the value is input to the TPM. One example of this use of HMACs is an authentication ticket. The TPM uses these tickets to prove to itself that it did an operation by producing an HMAC of a digest of the result of the operation, using an internal secret that only it knows.

HMACs can also be used to derive keys, using something called a key derivation function.

KDF: Key Derivation Function

A manufacturer might want to ship a TPM with certificates for multiple key sizes for multiple algorithms. But a TPM has a limited amount of space to store these keys. Further, many TPM protocols require more than one secret. For example, one secret may be required to symmetrically encrypt a message, and a second may be used to HMAC the result. In order to reduce the cost of manufacturing a TPM, the TPM has the ability to create multiple keys from a single secret. This secret is called a *seed*, and the algorithm used to derive multiple secrets from this seed is called a *key derivation function* (KDF). The TPM can use KDFs to derive both symmetric and asymmetric keys.

Because, as explained earlier, encryption doesn’t provide integrity, the usual pattern is to encrypt with a symmetric key and then HMAC with an HMAC key. The question that arises is, “Does this mean there must be two shared secrets?” Not usually. The design pattern is to share one secret, a seed. From this one seed, a KDF is used to derive a symmetric encryption key and an HMAC key.

The TPM uses an HMAC as a KDF based on an algorithm specified by NIST in Special Publication 800-108. In a typical case, it HMACs some varying data using the seed

as the HMAC key to derive the keys. The varying data always includes a string of bytes describing the application of the keys. This ensures that, when the same seed is used for different applications, different keys are used. This satisfies a basic crypto rule: never use the same key for two different applications. The other data also includes values unique to the operation. If you need two keys with the same description, you must use different unique values to guarantee that unique keys are created.

Authentication or Authorization Ticket

A *ticket* is a data structure that contains an HMAC that was calculated over some data. The ticket allows the TPM to validate at a later time that some operation was performed by the TPM. The HMAC asserts that some operation has been previously performed correctly and need not be performed again. In practice, the data is often not the actual message, which may be too large to fit in the ticket, but a digest of the message. The TPM uses tickets when it splits cryptographic operations into multiple operations with respect to time. Here, the HMAC key used to generate the HMAC is not a shared secret, but a secret known only to the TPM.

For example, the TPM uses a ticket when it calculates a hash over a message that it will later sign. It produces a ticket that says, “I calculated this hash and I assert that it is a hash that I will sign.” It signs the ticket by producing an HMAC using a secret only the TPM knows. Later, the HMAC is presented along with the ticket. The ticket is verified using the same secret HMAC key. Because only the TPM knows the HMAC key, it knows the ticket was produced by the TPM (authenticity) and that it has not been altered (integrity).

The TPM can do something similar when storing data outside the TPM. It can encrypt that data with a secret only it knows and then decrypt it again when loading it back inside the TPM. In this case, a symmetric key is used for encryption.

Symmetric-Encryption Key

A *symmetric-encryption key* is a key used with a symmetric-encryption algorithm. Symmetric algorithms use the same key for both encryption and decryption. (An HMAC key is also a symmetric key, but it’s used for signing, not encryption.)

A typical symmetric-key algorithm is the Advanced Encryption Standard (AES). Other algorithms are supported by the specification, including Camellia and SM4; but because they all work pretty much the same, all of this book’s examples use AES. The TPM uses symmetric-key encryption in three different ways:

- *Keeping TPM data secret from all observers:* The symmetric key isn’t shared outside the TPM. It’s generated by and known only to the TPM. For example, when a key is cached (offloaded) from the TPM in order to free memory for other TPM operations, the TPM encrypts the key using symmetric encryption. This symmetric key is known only to the TPM.
- *Encrypting communications to and from the TPM:* Here, the symmetric key is generated based on a secret agreed on by the sender and the TPM. Then parameters are passed to the TPM encrypted, and the results are returned encrypted from the TPM to the user.

- *Using the TPM as a cryptographic coprocessor:* Because the TPM knows how to encrypt things using symmetric keys, you can use the TPM to do that. You can load a key into the TPM and then ask the TPM to encrypt data with that key. TPMs usually aren't very fast at doing this, so this is typically only done for a small amount of data, but it can prevent an application programmer from having to use a cryptographic library for some programs. When specified as optional by the platform-specific TPM specifications, it's likely that TPM vendors and/or platform manufacturers will exclude symmetric encryption and decryption commands, because a hardware device that can do bulk symmetric-key operations can be subject to export (or perhaps import) restrictions or licensing.

Symmetric-key encryption is a little more complicated than just picking an algorithm and a key. You also need to select a mode of encryption. Different modes are used in different protocols.

Symmetric-Key Modes

Typical symmetric-key encryption algorithms like AES work on blocks of data. Two problems must be solved when using block-mode encryption:

- If blocks are simply encrypted with the key, the same block will always produce the same result. This is called *electronic codebook (ECB)* mode. If a bitmap picture is encrypted using ECB, all that happens is that the colors are changed.⁴ Obviously this isn't useful if the data being encrypted is large.

To counter this, the TPM supports several other modes: cipher-block chaining (CBC), cipher-feedback (CFB), output-feedback (OFB), and counter (CTR). All these modes have the property that if the same block is encrypted more than once in the same message, the result is different each time.

- Some modes, like CBC, require that the output be an exact multiple of the block size of the underlying algorithm. If the input isn't a multiple of the block size (which is usually 128 bits or 16 bytes), it is padded to make this true. When this input is encrypted, the output is larger than the initial data by the size of the padding. For applications where the output can be a different size than the input (such as offloading a key), this isn't a problem; but it's inappropriate when the input and output must be the same size (such as when you're encrypting a TPM command).

⁴http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

In this second case, you can use CFB or CTR mode. In CFB mode, a symmetric key encrypts an initialization vector, with the result being used as the initialization vector for the next block. In CTR mode, the symmetric key is used to encrypt incrementing counter values. In both modes, the resulting byte stream is XORed with the input to produce the output. As many bytes of the stream as necessary are used, and extra bytes are discarded, so the output is the same size as the input.

A property of CFB and CTR modes (actually a property of XOR) is that flipping a bit in the encrypted stream flips exactly the same bit in the decrypted stream. The attacker may not know the message but can certainly alter it. An attacker can flip a bit in a message encrypted using CBC mode as well, but more bits will change in the decrypted data.

This leads to an important (and often missed) point. Encryption provides secrecy, but it does not provide integrity or authenticity. To ensure those latter properties, the TPM uses an HMAC on the encrypted data. It does not depend on the decrypted data “looking funny” to detect alteration. Indeed, by calculating the HMAC of the encrypted message first, the TPM will not even attempt to decrypt it unless it is first determined that the message’s integrity is intact and that it is authentic.

Additionally, encryption does not provide evidence that the message was produced recently. That is done with a nonce.

Nonce

A *nonce* is a number that is used only once in a cryptographic operation. It provides protection against a replay attack. In order to guarantee that a message hasn’t been replayed, the recipient generates the nonce and sends it to the sender. The sender includes that nonce in the message. Because the sender presumably has no way of knowing what nonce the recipient will choose, they can’t replay a message that was prepared earlier. But of course, you must take care that a previously prepared message can’t just be minimally modified. When sending commands to a TPM and receiving the results back, nonces provide proof to the user that the results of the command were sent by the TPM.

In a typical TPM use, the nonce is included in the calculation of the HMAC of a command message. After an operation using the message is complete, the TPM changes the nonce. If the caller attempts to replay the message which had an HMAC that used the previous nonce, the TPM will attempt to verify the HMAC of the replayed message using the new nonce, and this verification will fail.

For many applications, a nonce can simply be a number that increments at each use and is large enough to never wrap around. However, this would require the TPM to keep track of the last-used value. Instead, the TPM takes advantage of its random-number generator. It uses random numbers as its nonces, and it uses large enough values (for example, 20 bytes) that the odds of a repeat are nil.

Asymmetric Keys

Asymmetric keys are used by asymmetric algorithms for digital identities and key management. They are actually a key pair: a private key known only to one party and a public key known to everyone. Asymmetric keys make use of mathematical *one-way* functions. These functions have the property that calculating the public key from the private key is relatively easy computationally, but calculating the private key from the public key is computationally infeasible.

You all have window seats in The Restaurant at the End of the Universe

—Mullen 2011

If the owner of the private key uses it to encrypt some data (and the key is large enough), everyone can use the public key to decrypt the data, but everyone will know that only the holder of the private key could have encrypted that data. This is called *signing* the data. There are some complications to using this securely—the data that is signed should be in a particular format called a *signing scheme*, but the TPM ensures sure that the correct format is used.

If someone wants to share data (usually a symmetric key) with the owner of a private key in a secure way, they can provide the data to the owner by encrypting it with the owner's public key. Then they can be certain that only the owner will be able to recover the shared data by using the owner's private key. This is done in different ways depending on the type of asymmetric algorithm, but we skip these deep mathematical details here.

RSA Asymmetric-Key Algorithm

RSA is a well-known asymmetric-key algorithm. It uses the factoring of large numbers into large primes as its one-way function. RSA has an interesting property: If the private key is first applied to a message and then the public key is applied to the result, the original message is obtained. Alternatively, if the public key is applied to a message and then the private key is applied to the result, again the original message is obtained.

Thus, RSA can be used for both encryption and digital signatures. In encryption and decryption, the public key is used to encrypt data, and the private key is used to decrypt data. For digital signatures, the private key is used to digitally sign, and the public key is used to verify signatures.

RSA for Key Encryption

To encrypt using the asymmetric keys, you apply the recipient's public key, which is known to you because it's public, to the message. The holder of the private key can apply their key to recover the message. It is secret from everyone else because the private key is, well, private.

In practice, messages are not typically encrypted directly with an asymmetric key. The data size is limited, based on the size of the key. Breaking up the message into smaller pieces is possible but impractical because asymmetric-key operations are typically very slow.

The usual pattern is to encrypt a symmetric key with the asymmetric public key, send the encrypted symmetric key to the recipient, and then encrypt the message with that symmetric key. The recipient decrypts the symmetric key using their private key and then decrypts the message with the much faster symmetric-key algorithm.

RSA for Digital Signatures

A digital signature is similar to an HMAC but has additional properties. To create an asymmetric-key digital signature, the signer applies their private key to a message. The verifier applies the public key to the signature to recover and verify the message.

A digital signature permits the recipient to know that a message has integrity and is authentic, qualities that an HMAC also possesses. An asymmetric-key digital signature goes further:

- Because the verification key is public, multiple parties can verify the signature. With an HMAC, the verification key is a shared secret, and only a holder of the shared secret can verify the message.
- Because a private key is used to generate the signature, only the sender (the holder of the private key) could have generated the signature, and the recipient can prove it to a third party. With an HMAC, a shared secret is used, and both the sender and the recipient know the shared secret. The recipient can verify that the signature was generated by the sender, but the recipient can't prove this to a third party, because, for all the third party knows, the recipient could also have generated the signature.

As with asymmetric-key encryption, the digital signature isn't typically applied directly to the message, because the message would be limited based on the key size.

The usual pattern is to digest the message and apply the private key to the smaller digest. The verifier applies the public key to the signature to recover the signed digest and compares that digest to one calculated from the message. This works because it is infeasible for an attacker to construct a second message with the same digest.

RSA is not the only asymmetric-key algorithm. Elliptic curve cryptography (ECC) is gaining popularity and is included in the latest specification.

ECC Asymmetric-Key Algorithm

ECC is another type of asymmetric mathematics that is used for cryptography. Unlike RSA, which uses an easily understood mathematical operation—factoring a product of two large primes—ECC uses more difficult mathematical concepts based on elliptic curves over a finite field. We will not describe the mathematics but instead describe how it is used. Just like every other asymmetric algorithm, ECC has a private and public key pair. The public key can be used to verify something signed with the private key, and the private key can be used to decrypt data that was encrypted using the public key.

For equivalent strength, ECC keys are much smaller than RSA keys. The strength of an ECC key is half the key size, so a 256-bit ECC key has 128 bits of strength. A similarly strong RSA key is 3,076 bits long. Smaller key sizes use fewer resources and perform faster. For encryption, a procedure known as *Elliptic Curve Diffie-Hellman (ECDH)* is used with ECC. For signing, *Elliptic Curve Digital Signature Algorithm (ECDSA)* is used.

ECDH Asymmetric-Key Algorithm to Use Elliptic Curves to Pass Keys

When using ECC to encrypt/decrypt asymmetrically, you use the ECDH algorithm. The main difference between ECC and RSA for encryption/decryption is that the process of using an ECDH key takes two steps, whereas RSA takes only one. When encrypting a symmetric key with a TPM-based RSA key, you use the TPM RSA's public key to encrypt it. When encrypting a symmetric key with a TPM-based ECDH key, two steps are required: Generate (in software) another ECDH key; and then use the private key of the newly generated ECDH key and the public portion of the TPM ECDH key to generate a new ephemeral random number, which is input to a KDF to generate a symmetric key. To put this more succinctly, with RSA you can supply the symmetric key to be encrypted, but with ECDH the process generates the symmetric key.

To recover the symmetric key, the public portion of the software-generated ECDH key is given to the TPM. It uses it together with the private portion of its own ECDH key to regenerate the ephemeral random number, which it inputs into a KDF internally to regenerate the symmetric key.

ECDSA Asymmetric-Key Algorithm to Use Elliptic Curves for Signatures

ECDSA is used as an algorithm with ECC to produce signatures. Just as with RSA, in ECDSA the private key is used to sign and the public key is used to verify the signature. The main difference (other than the mathematical steps used) is that when using an ECC key, because it's much smaller than an RSA key, you have to ensure that the hash of the message you're signing isn't too big. The ECDSA signature signs only n bits of the hash, where n is the size of the key. (This is also true of RSA; but RSA keys sizes are typically $\geq 1,024$ bits and hash sizes top out at 512 bits, so this is never a problem.)

Whereas with RSA you can typically sign a message with any hash algorithm, with ECC you typically use a hash algorithm that matches the size of the key: SHA-256 for ECC-256 and SHA-384 for ECC-384. If you used SHA-512 (which produces 512-bit hashes) with an ECC-384 key, ECDSA would sign only the first 384 bits of the hash. You can sign smaller hashes without any problem, of course, so an ECC 384-bit key could be used to sign SHA-384, SHA-256, or SHA-1 (160 bits) hashes.

One problem with all signing protocols is that the recipient of the signature needs to be assured that the public key they use to verify the signature really belongs to the owner of the private key who signed it. This is handled with public key certificates.

Public Key Certification

Certification is part of an asymmetric-key protocol and solves the following problem: How do you trust the public key? After all, it accomplishes nothing to verify a digital signature with a public key if you don't know whose public key you're using. Secrecy won't be preserved if you encrypt a message with the attacker's public key. Establishing trust in a TPM public key includes knowing that the key really came from whom it was supposed to come from—in this case, a TPM.

The solution is to create a digital certificate. A certificate includes the public part of the key being certified plus attributes of that key. The certificate is then signed by a certificate authority (CA) key. It's possible that the CA public key is in turn certified by another CA key, forming a hierarchy (a certificate *chain*). At some point, this certificate chain terminates at a root certificate. The root public key must be conveyed to a verifier out of band and is declared trusted without cryptographic proof.

The X.509 standard⁵ describes a widely used certificate format. The TPM, as a limited-resource device, neither creates nor consumes X.509 certificates. The TCG Infrastructure work group does specify some X.509 certificate formats, and the TPM typically stores them. This storage is simply for provisioning convenience, pairing a certificate with its key, not to achieve any security goal.

In the TPM space, there are several certification processes:

- The TPM vendor and platform manufacturer may provision the TPM with TPM vendor and platform endorsement keys (EKs) and corresponding certificates before shipment to the end user. The TPM vendor certificate asserts, "This endorsement key is resident on an authentic TPM manufactured by me." The platform manufacturer certificate asserts, "This key is resident on a TPM that is part of my platform, and this platform supports certain TPM features." These certificates typically use X.509 format.
- If the TPM keys (and their corresponding certificates) just described exist as signing keys, they can be used to certify other keys as being resident on the TPM and having certain properties. The TPM 2.0 specification provides commands to create certificates. These TPM-generated certificates do not use X.509, which is too complex for the limited on-chip resources of the TPM.

Essentially, digital certificates rest on the integrity of the CA. The CA is considered to be a neutral party that can be trusted by two parties: the parties that create the certificates and those that use them. A CA's functioning is similar to an escrow agent that mediates fund transfers in a real-estate transaction. If the CA is worthy of trust, all is good. If not, all bets are off.

When a TPM manufacturer produces a certificate for a TPM, the manufacturer faces a quandary. What algorithm should be used for the key? The manufacturer doesn't know if the end user will want RSA-2048, ECC2-56, ECC-384, or some other algorithm. And it also needs to know what hash algorithm and symmetric algorithm should be used in the creation of the key. To solve this problem, the TPM is designed to allow the creation of many keys derived from a single large random number, using a key-derivation function, as described earlier. You see in the chapter on hierarchies how this is used to provide many certificates for multiple algorithms without using up space in the TPM.

⁵www.ietf.org/rfc/rfc2459.txt.

Summary

By examining a sample use case, you've seen all the major security operations and concepts that are used in the rest of the book to explain the creation and use of the TPM. This isn't surprising, because the TPM was designed with use cases in mind, and one of the major ones was e-commerce. By starting with the attacks cryptographic operations need to defend against, you saw why cryptographic algorithms are chosen from well-vetted internationally recognized algorithms and how key strengths are chosen. You reviewed the concepts of confidentiality, integrity, electronic identity, and nonrepudiation and how they relate to the standard classes of algorithms: symmetric, asymmetric, hash, and HMAC. Finally, you learned about some specific new features in the TPM specification that use those algorithms: extend, tickets, and certificates. You're ready to consider all the use cases the TPM was design to solve.