56

Protecting Information With Cryptography

Chapter by **Peter Reiher (UCLA)**

56.1 Introduction

In previous chapters, we've discussed clarifying your security goals, determining your security policies, using authentication mechanisms to identify principals, and using access control mechanisms to enforce policies concerning which principals can access which computer resources in which ways. While we identified a number of shortcomings and problems inherent in all of these elements of securing your system, if we regard those topics as covered, what's left for the operating system to worry about, from a security perspective? Why isn't that everything?

There are a number of reasons why we need more. Of particular importance: not everything is controlled by the operating system. But perhaps you respond, you told me the operating system is all-powerful! Not really. It has substantial control over a limited domain – the hardware on which it runs, using the interfaces of which it is given control. It has no real control over what happens on other machines, nor what happens if one of its pieces of hardware is accessed via some mechanism outside the operating system's control.

But how can we expect the operating system to protect something when the system does not itself control access to that resource? The answer is to prepare the resource for trouble in advance. In essence, we assume that we are going to lose the data, or that an opponent will try to alter it improperly. And we take steps to ensure that such actions don't cause us problems. The key observation is that if an opponent cannot understand the data in the form it is obtained, our secrets are safe. Further, if the attacker cannot understand it, it probably can't be altered, at least not in a controllable way. If the attacker doesn't know what the data means, how can it be changed into something the attacker prefers?

The core technology we'll use is **cryptography**, a set of techniques to convert data from one form to another, in controlled ways with expected outcomes. We will convert the data from its ordinary form into another form using cryptography. If we do it right, the opponent will not be able to determine what the original data was by examining the protected form.

Of course, if we ever want to use it again ourselves, we must be able to reverse that transformation and return the data to its ordinary form. That must be hard for the opponent to do, as well. If we can get to that point, we can also provide some protection for the data from alteration, or, more precisely, prevent opponents from altering the data to suit their desires, and even know when opponents have tampered with our data. All through the joys of cryptography!

But using cryptography properly is not easy, and many uses of cryptography are computationally expensive. So we need to be selective about where and when we use cryptography, and careful in how we implement it and integrate it into our systems. Well chosen uses that are properly performed will tremendously increase security. Poorly chosen uses that are badly implemented won't help at all, and may even hurt.

THE CRUX OF THE PROBLEM:
HOW TO PROTECT INFORMATION OUTSIDE THE OS'S DOMAIN

How can we use cryptography to ensure that, even if others gain access to critical data outside the control of the operating system, they will be unable to either use or alter it? What cryptographic technologies are available to assist in this problem? How do we properly use those technologies? What are the limitations on what we can do with them?

56.2 Cryptography

Many books have been written about cryptography, but we're only going to spend a chapter on it. We'll still be able to say useful things about it because, fortunately, there are important and complex issues of cryptography that we can mostly ignore. That's because we aren't going to become cryptographers ourselves. We're merely going to be users of the technology, relying on experts in that esoteric field to provide us with tools that we can use without having full understanding of their workings[1]. That sounds kind of questionable, but you are already doing just that. Relatively few of us really understand the deep details of how our computer hardware works, yet we are able to make successful use of it, because we have good interfaces and know that smart people have taken great care in building the hardware for us. Similarly, cryptography provides us with strong interfaces, well-defined behaviors, and better than usual assurance that there is a lot of brain power behind the tools we use.

That said, cryptography is no magic wand, and there is a lot you need to understand merely to use it correctly. That, particularly in the context of operating system use, is what we're going to concentrate on here.

[1] If you'd like to learn more about the fascinating history of cryptography, check out Kahn [K96]. If more technical detail is your desire, Schneier [S96] is a good start.

The basic idea behind cryptography is to take a piece of data and use an algorithm (often called a **cipher**), usually augmented with a second piece of information (which is called a **key**), to convert the data into a different form. The new form should look nothing like the old one, but, typically, we want to be able to run another algorithm, again augmented with a second piece of information, to convert the data back to its original form.

Let's formalize that just a little bit. We start with data P (which we usually call the **plaintext**), a key K, and an encryption algorithm $E()$. We end up with C, the altered form of P, which we usually call the **ciphertext**:

$$C = E(P, K) \tag{56.1}$$

For example, we might take the plaintext "Transfer $100 to my savings account" and convert it into ciphertext "Sqzmredq #099 sn lx rzuhmfr zbbntms." This example actually uses a pretty poor encryption algorithm called a Caesar cipher. Spend a minute or two studying the plaintext and ciphertext and see if you can figure out what the encryption algorithm was in this case.

The reverse transformation takes C, which we just produced, a decryption algorithm $D()$, and the key K:

$$P = D(C, K) \tag{56.2}$$

So we can decrypt "Sqzmredq #099 sn lx rzuhmfr zbbntms" back into "Transfer $100 to my savings account." If you figured out how we encrypted the data in the first place, it should be easy to figure out how to decrypt it.

We use cryptography for a lot of things, but when discussing it generally, it's common to talk about messages being sent and received. In such discussions, the plaintext P is the message we want to send and the ciphertext C is the protected version of that message that we send out into the cold, cruel world.

For the encryption process to be useful, it must be deterministic, so the first transformation always converts a particular P using a particular K to a particular C, and the second transformation always converts a particular C using a particular K to the original P. In many cases, $E()$ and $D()$ are actually the same algorithm, but that is not required. Also, it should be very hard to figure out P from C without knowing K. Impossible would be nice, but we'll usually settle for computationally infeasible. If we have that property, we can show C to the most hostile, smartest opponent in the world and they still won't be able to learn what P is.

Provided, of course, that ...

This is where cleanly theoretical papers and messy reality start to collide. We only get that pleasant assurance of secrecy if the opponent does not know both $D()$ and our key K. If they are known, the opponent will apply $D()$ and K to C and extract the same information P that we can.

It turns out that we usually can't keep $E()$ and $D()$ secret. Since we're not trying to be cryptographers, we won't get into the why of the matter, but it is extremely hard to design good ciphers. If the cipher has weaknesses, then an opponent can extract the plaintext P even without K. So we need to have a really good cipher, which is hard to come by. Most of us don't have a world-class cryptographer at our fingertips to design a new one, so we have to rely on one of a relatively small number of known strong ciphers. AES, a standard cipher that was carefully designed and thoroughly studied, is one good example that you should think about using.

It sounds like we've thrown away half our protection, since now the cryptography's benefit relies entirely on the secrecy of the key. Precisely. Let's say that again in all caps, since it's so important that you really need to remember it: **THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY**. It probably wouldn't hurt for you to re-read that statement a few dozen times, since the landscape is littered with insecure systems that did not take that lesson to heart.

The good news is that if you're using a strong cipher and are careful about maintaining key secrecy, your cryptography is strong. You don't need to worry about anything else. The bad news is that maintaining key secrecy in practical systems for real uses of cryptography isn't easy. We'll talk more about that later.

For the moment, revel in the protection we have achieved, and rejoice to learn that we've gotten more than secrecy from our proper use of cryptography! Consider the properties of the transformations we've performed. If our opponent gets access to our encrypted data, it can't be understood. But what if the opponent can alter it? What's being altered is the encrypted form, i.e., making some changes in C to convert it to, say, C'. What will happen when we try to decrypt C? Well, it won't decrypt to P. It will decrypt to something else, say P'. For a good cipher of the type you should be using, it will be difficult to determine what a piece of ciphertext C' will decrypt to, unless you know K. That means it will be hard to predict which ciphertext you need to have to decrypt to a particular plaintext. Which in turn means that the attacker will have no idea what the altered ciphertext C' will decrypt to.

Out of all possible bit patterns it could decrypt to, the chances are good that P' will turn out to be garbage, when considered in the context of what we expected to see: ASCII text, a proper PDF file, or whatever. If we're careful, we can detect that P' isn't what we started with, which would tell us that our opponent tampered with our encrypted data. If we want to be really sure, we can perform a hashing function on the plaintext and include the hash with the message or encrypted file. If the plaintext we get out doesn't produce the same hash, we will have a strong indication that something is amiss.

So we can use cryptography to help us protect the integrity of our data, as well.

TIP: DEVELOPING YOUR OWN CIPHERS: DON'T DO IT

Don't.

It's tempting to leave it at that, since it's really important that you follow this guidance. But you may not believe it, so we'll expand a little. The world's best cryptographers often produce flawed ciphers. Are you one of the world's best cryptographers? If you aren't, and the top experts often fail to build strong ciphers, what makes you think you'll do better, or even as well?

We know what you'll say next: but the cipher I wrote is so strong that I can't even break it myself. Well, pretty much anyone who puts their mind to it can create a cipher they can't break themselves. But remember those world-class cryptographers we talked about? How did they get to be world class? By careful study of the underpinnings of cryptography and by breaking other people's ciphers. They're very good at it, and if it's worth their trouble, they will break yours. They might ignore it if you just go around bragging about your wonderful cipher (since they hear that all the time), but if you actually use it for something important, you will unfortunately draw their attention. Following which your secrets will be revealed, following which you will look foolish for designing your own cipher instead of using something standard like AES, which is easier to do, anyway.

So, don't.

Wait, there's more! What if someone hands you a piece of data that has been encrypted with a key K that is known only to you and your buddy Remzi? You know you didn't create it, so if it decrypts properly using key K, you know that Remzi must have created it. After all, he's the only other person who knew key K, so only he could have performed the encryption. Voila, we have used cryptography for authentication! Unfortunately, cryptography will not clean your room, do your homework for you, or make thousands of julienne fries in seconds, but it's a mighty fine tool, anyway.

The form of cryptography we just described is often called **symmetric cryptography**, because the same key is used to encrypt and decrypt the data. For a long time, everyone believed that was the only form of cryptography possible. It turns out everyone was wrong.

56.3 Public Key Cryptography

When we discussed using cryptography for authentication, you might have noticed a little problem. In order to verify the authenticity of a piece of encrypted information, you need to know the key used to encrypt it. If we only care about using cryptography for authentication, that's inconvenient. It means that we need to communicate the key we're using for

that purpose to whoever might need to authenticate us. What if we're Microsoft, and we want to authenticate ourselves to every user who has purchased our software? We can't use just one key to do this, because we'd need to send that key to hundreds of millions of users and, once they had that key, they could pretend to be Microsoft by using it to encrypt information. Alternately, Microsoft could generate a different key for each of those hundreds of millions of users, but that would require secretly delivering a unique key to hundreds of millions of users, not to mention keeping track of all those keys. Bummer.

Fortunately, our good friends, the cryptographic wizards, came up with a solution. What if we use two different keys for cryptography, one to encrypt and one to decrypt? Our encryption operation becomes

$$C = E(P, K_{encrypt}) \tag{56.3}$$

And our decryption operation becomes

$$P = D(C, K_{decrypt}) \tag{56.4}$$

Life has just become a lot easier for Microsoft. They can tell everyone their decryption key $K_{decrypt}$, but keep their encryption key $K_{encrypt}$ secret. They can now authenticate their data by encrypting it with their secret key, while their hundreds of millions of users can check the authenticity using the key Microsoft made public. For example, Microsoft could encrypt an update to their operating system with $K_{encrypt}$ and send it out to all their users. Each user could decrypt it with $K_{decrypt}$. If it decrypted into a properly formatted software update, the user could be sure it was created by Microsoft. Since no one else knows that private key, no one else could have created the update.

Sounds like magic, but it isn't. It's actually mathematics coming to our rescue, as it so frequently does. We won't get into the details here, but you have to admit it's pretty neat. This form of cryptography is called **public key cryptography**, since one of the two keys can be widely known to the entire public, while still achieving desirable results. The key everyone knows is called the **public key**, and the key that only the owner knows is called the **private key**. Public key cryptography (often abbreviated as **PK**) has a complicated invention history, which, while interesting, is not really germane to our discussion. Check out a paper by a pioneer in the field, Whitfield Diffie, for details [D88].

Public key cryptography avoids one hard issue that faced earlier forms of cryptography: securely distributing a secret key. Here, the private key is created by one party and kept secret by him. It's never distributed to anyone else. The public key must be distributed, but generally we don't care if some third party learns this key, since they can't use it to sign messages. Distributing a public key is an easier problem than distributing a secret key, though, alas, it's harder than it sounds. We'll get to that.

Public key cryptography is actually even neater, since it works the other way around. You can use the decryption key $K_{decrypt}$ to encrypt, in which case you need the encryption key $K_{encrypt}$ to decrypt. We still

expect the encryption key to be kept secret and the decryption key to be publicly known, so doing things in this order no longer allows authentication. Anyone could encrypt with $K_{decrypt}$, after all. But only the owner of the key can decrypt such messages using $K_{encrypt}$. So that allows anyone to send an encrypted message to someone who has a private key, provided you know their public key. Thus, PK allows authentication if you encrypt with the private key and secret communication if you encrypt with the public key.

What if you want both, as you very well might? You'll need two different key pairs to do that. Let's say Alice wants to use PK to communicate secretly with her pal Bob, and also wants to be sure Bob can authenticate her messages. Let's also say Alice and Bob each have their own PK pair. Each of them knows his or her own private key and the other party's public key. If Alice encrypts her message with her own private key, she'll authenticate the message, since Bob can use her public key to decrypt and will know that only Alice could have created that message. But everyone knows Alice's public key, so there would be no secrecy achieved. However, if Alice takes the authenticated message and encrypts it a second time, this time with Bob's public key, she will achieve secrecy as well. Only Bob knows the matching private key, so only Bob can read the message. Of course, Bob will need to decrypt twice, once with his private key and then a second time with Alice's public key.

Sounds expensive. It's actually worse than you think, since it turns out that public key cryptography has a shortcoming: it's much more computationally expensive than traditional cryptography that relies on a single shared key. Public key cryptography can take hundreds of times longer to perform than standard symmetric cryptography. As a result, we really can't afford to use public key cryptography for everything. We need to pick and choose our spots, using it to achieve the things it's good at.

There's another important issue. We rather blithely said that Alice knows Bob's public key and Bob knows Alice's. How did we achieve this blissful state of affairs? Originally, only Alice knew her public key and only Bob knew his public key. We're going to need to do something to get that knowledge out to the rest of the world if we want to benefit from the magic of public key cryptography. And we'd better be careful about it, since Bob is going to assume that messages encrypted with the public key he thinks belongs to Alice *were* actually created by Alice. What if some evil genius, called, perhaps, Eve, manages to convince Bob that Eve's public key actually belongs to Alice? If that happens, messages created by Eve would be misidentified by Bob as originating from Alice, subverting our entire goal of authenticating the messages. We'd better make sure Eve can't fool Bob about which public key belongs to Alice.

This leads down a long and shadowy road to the arcane realm of key distribution infrastructures. You will be happier if you don't try to travel that road yourself, since even the most well prepared pioneers who have hazarded it often come to grief. We'll discuss how, in practice, we distribute public keys in a chapter on distributed system security. For the

moment, bear in mind that the beautiful magic of public key cryptography rests on the grubby and uncertain foundation of key distribution.

One more thing about PK cryptography: **THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY**. (Bet you've heard that before.) In this case, the private key. But the secrecy of that private key is every bit as important to the overall benefit of public key cryptography as the secrecy of the single shared key in the case of symmetric cryptography. Never divulge private keys. Never share private keys. Take great care in your use of private keys and in how you store them. If you lose a private key, everything you used it for is at risk, and whoever gets hold of it can pose as you and read your secret messages. That wouldn't be very good, would it?

56.4 Cryptographic Hashes

As we discussed earlier, we can protect data integrity by using cryptography, since alterations to encrypted data will not decrypt properly. We can reduce the costs of that integrity check by hashing the data and encrypting just the hash, instead of encrypting the entire thing. However, if we want to be really careful, we can't use just any hash function, since hash functions, by their very nature, have **hash collisions**, where two different bit patterns hash to the same thing. If an attacker can change the bit pattern we intended to send to some other bit pattern that hashes to the same thing, we would lose our integrity property.

So to be particularly careful, we can use a **cryptographic hash** to ensure integrity. Cryptographic hashes are a special category of hash functions with several important properties:

- It is computationally infeasible to find two inputs that will produce the same hash value.
- Any change to an input will result in an unpredictable change to the resulting hash value.
- It is computationally infeasible to infer any properties of the input based only on the hash value.

Based on these properties, if we only care about data integrity, rather than secrecy, we can take the cryptographic hash of a piece of data, encrypt only that hash, and send both the encrypted hash and the unencrypted data to our partner. If an opponent fiddles with the data in transit, when we decrypt the hash and repeat the hashing operation on the data, we'll see a mismatch and detect the tampering[2].

[2]Why do we need to encrypt the cryptographic hash? Well, anyone, including our opponent, can run a cryptographic hashing algorithm on anything, including an altered version of the message. If we don't encrypt the hash, the attacker will change the message, compute a new hash, replace both the original message and the original hash with these versions, and send the result. If the hash we sent is encrypted, though, the attacker can't know what the encrypted version of the altered hash should be.

To formalize it a bit, to perform a cryptographic hash we take a plaintext P and a hashing algorithm $H()$. Note that there is not necessarily any key involved. Here's what happens:

$$S = H(P) \tag{56.5}$$

Since cryptographic hashes are a subclass of hashes in general, we normally expect S to be shorter than P, perhaps a lot shorter. That implies there will be collisions, situations in which two different plaintexts P and P' both hash to S. However, the properties of cryptographic hashes outlined above will make it difficult for an adversary to make use of collisions. Even if you know both S and P, it should be hard to find any other plaintext P' that hashes to S[3]. It won't be hard to figure out what S' should be for an altered value of plaintext P', since you can simply apply the cryptographic hashing algorithm directly to P'. But even a slightly altered version of P, such as a P' differing only in one bit, should produce a hash S' that differs from S in completely unpredictable ways.

Cryptographic hashes can be used for other purposes than ensuring integrity of encrypted data, as well. They are the class of hashes of choice for storing salted hashed passwords, for example, as discussed in the chapter on authentication. They can be used to determine if a stored file has been altered, a function provided by well-known security software like Tripwire. They can also be used to force a process to perform a certain amount of work before submitting a request, an approach called "proof of work." The submitter is required to submit a request that hashes to a certain value using some specified cryptographic hash, which, because of the properties of such hashes, requires them to try a lot of request formats before finding one that hashes to the required value. Since each hash operation takes some time, submitting a proper request will require a predictable amount of work. This use of hashes, in varying forms, occurs in several applications, including spam prevention and blockchains.

Like other cryptographic algorithms, you're well advised to use standard algorithms for cryptographic hashing. For example, the SHA-3 algorithm is commonly regarded as a good choice. However, there is a history of cryptographic hashing algorithms becoming obsolete, so if you are designing a system that uses one, it's wise to first check to see what current recommendations are for choices of such an algorithm.

56.5 Cracking Cryptography

Chances are that you've heard about people cracking cryptography. It's a popular theme in film and television. How worried should you be about that?

[3]Every so often, a well known cryptographic hashing function is "broken" in the sense that someone figures out how to create a P' that uses the function to produce the same hash as P. That happened to a hashing function known as SHA-1 in 2017, rendering that function unsafe and unusable for integrity purposes [G17].

Well, if you didn't take our earlier advice and went ahead and built your own cipher, you should be very worried. Worried enough that you should stop reading this, rip out your own cipher from your system, and replace it with a well-known respected standard. Go ahead, we'll still be here when you get back.

What if you did use one of those standards? In that case, you're probably OK. If you use a modern standard, with a few unimportant exceptions, there are no known ways to read data encrypted with these algorithms without obtaining the key. Which isn't to say your system is secure, but probably no one will break into it by cracking the cryptographic algorithm.

How will they do it, then? Probably by exploiting software flaws in your system having nothing to do with the cryptography, but there's some chance they will crack it by obtaining your keys or exploiting some other flaw in your management of cryptography. How? Software flaws in how you create and use your keys are a common problem. In distributed environments, flaws in the methods used to share keys are also a common weakness that can be exploited. Peter Gutmann produced a nice survey of the sorts of problems improper management of cryptography frequently causes [G02]. Examples include distributing secret keys in software shared by many people, incorrectly transmitting plaintext versions of keys across a network, and choosing keys from a seriously reduced set of possible choices, rather than the larger theoretically possible set. More recently, the Heartbleed attack demonstrated a way to obtain keys being used in OpenSSL sessions from the memory of a remote computer, which allowed an attacker to decrypt the entire session, despite no flaws in either the cipher itself or its implementation, nor in its key selection procedures. This flaw allowed attackers to read the traffic of something between $1/4$ and $1/2$ of all sites using HTTPS, the cryptographically protected version of HTTP [D+14].

One way attackers deal with cryptography is by guessing the key. Doing so doesn't actually crack the cryptography at all. Cryptographic algorithms are designed to prevent people who don't know the key from obtaining the secrets. If you know the key, it's not supposed to make decryption hard.

So an attacker could try simply guessing each possible key and trying it. That's called a brute force attack, and it's why you should use long keys. For example, AES keys are at least 128 bits. Assuming you generate your AES key at random, an attacker will need to make 2^{127} guesses at your key, on average, before he gets it right. That's a lot of guesses and will take a lot of time. Of course, if a software flaw causes your system to select one out of thirty two possible AES keys, instead of one out of 2^{128}, a brute force attack may become trivial. Key selection is a big deal for cryptography.

For example, the original 802.11 wireless networking standard included no cryptographic protection of data being streamed through the air. The

TIP: SELECTING KEYS

One important aspect of key secrecy is selecting a good one to begin with. For public key cryptography, you need to run an algorithm to select one of the few possible pairs of keys you will use. But for symmetric cryptography, you are free to select any of the possible keys. How should you choose?

Randomly. If you use any deterministic method to select your key, your opponent's problem of finding out your key has just been converted into a problem of figuring out your method. Worse, since you'll probably generate many keys over the course of time, once he knows your method, he'll get all of them. If you use random chance to generate keys, though, figuring out one of them won't help your opponent figure out any of your other keys. This highly desirable property in a cryptographic system is called **perfect forward secrecy**.

Unfortunately, true randomness is hard to come by. The best source for operating system purposes is to examine hardware processes that are believed to be random in nature, like low order bits of the times required for pieces of hardware to perform operations, and convert the results into random numbers. That's called **gathering entropy**. In Linux, this is done for you automatically, and you can use the gathered entropy by reading `/dev/random`. Windows has a similar entropy-gathering feature. Use these to generate your keys. They're not perfect, but they're good enough for many purposes.

first attempt to add such protection was called WEP (Wired Equivalent Protocol, a rather optimistic name). WEP was constrained by the need to fit into the existing standard, but the method it used to generate and distribute symmetric keys was seriously flawed. Merely by listening in on wireless traffic on an 802.11 network, an attacker could determine the key being used in as little as a minute. There are widely available tools that allow anyone to do so[4].

As another example, an early implementation of the Netscape web browser generated cryptographic keys using some easily guess-able values as seeds to a random number generator, such as the time of day and the ID of the process requesting the key. Researchers discovered they could guess the keys produced in around 30 seconds [GW96].

You might have heard that PK systems use much longer keys, 2K or 4K bits. Sounds much safer, no? Shouldn't that at least make them stronger against brute force attacks? However, you can't select keys for this type of

[4]WEP got replaced by WPA. Unfortunately, WPA proved to have its own weaknesses, so it was replaced by WPA2. Unfortunately, WPA2 proved to have its own weaknesses, so it is being replaced by WPA3, as of 2018. The sad fate of providing cryptography for wireless networks should serve as a lesson to any of you tempted to underestimate the difficulties in getting this stuff right.

cryptosystem at random. Only a relatively few pairs of public and private keys are possible. That's because the public and private keys must be related to each other for the system to work. The relationship is usually mathematical, and usually intended to be mathematically hard to derive, so knowing the public key should not make it easy to learn the private key. However, with the public key in hand, one can use the mathematical properties of the system to derive the private key eventually. That's why PK systems use such big keys – to make sure "eventually" is a very long time.

But that only matters if you keep the private key secret. By now, we hope this sounds obvious, but many makers of embedded devices use PK to provide encryption for those devices, and include a private key in the device's software. All too often, the same private key is used for all devices of a particular model. Such shared private keys invariably become, well, public. In September 2016, one study found 4.5 million embedded devices relying on these private keys that were no longer so private [V16]. Anyone could pose as any of these devices for any purpose, and could read any information sent to them using PK. In essence, the cryptography performed by these devices was little more than window dressing and did not increase the security of the devices by any appreciable amount.

To summarize, cracking cryptography is usually about learning the key. Or, as you might have guessed: **THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY.**

56.6 Cryptography And Operating Systems

Cryptography is fascinating, but lots of things are fascinating[5], while having no bearing on operating systems. Why did we bother spending half a chapter on cryptography? Because we can use it to protect operating systems.

But not just anywhere and for all purposes. We've pounded into your head that key secrecy is vital for effective use of cryptography. That should make it clear that any time the key can't be kept secret, you can't effectively use cryptography. Casting your mind back to the first chapter on security, remember that the operating system has control of and access to all resources on a computer. Which implies that if you have encrypted information on the computer, and you have the necessary key to decrypt it on the same computer, the operating system on that machine can decrypt the data, whether that was the effect you wanted or not[6].

[5] For example, the late piano Sonatas of Beethoven. One movement of his last Sonata, Opus 111, even sounds like jazz, while being written in the 1820s!

[6] But remember our discussion of security enclaves in an earlier chapter, hardware that does not allow the operating system full access to information that the enclave protects. Think for a moment what the implications of that are for cryptography on a computer using such an enclave, and what new possibilities it offers.

Either you trust your operating system or you don't. If you don't, life is going to be unpleasant anyway, but one implication is that the untrusted operating system, having access at one time to your secret key, can copy it and re-use it whenever it wants to. If, on the other hand, you trust your operating system, you don't need to hide your data from it, so cryptography isn't necessary in this case. This observation has relevance to any situation in which you provide your data to something you don't trust. For instance, if you don't trust your cloud computing facility with your data, you won't improve the situation by giving them your data in plaintext and asking them to encrypt it. They've seen the plaintext and can keep a copy of the key.

If you're sure your operating system is trustworthy right now, but are concerned it might not be later, you can encrypt something now and make sure the key is not stored on the machine. Of course, if you're wrong about the current security of the operating system, or if you ever decrypt the data on the machine after the OS goes rogue, your cryptography will not protect you, since that ever-so-vital secrecy of the key will be compromised.

One can argue that not all compromises of an operating system are permanent. Many are, but some only give an attacker temporary access to system resources, or perhaps access to only a few particular resources. In such cases, if the encrypted data is not stored in plaintext and the decryption key is not available at the time or in the place the attacker can access, encrypting that data may still provide benefit. The tricky issue here is that you can't know ahead of time whether successful attacks on your system will only occur at particular times, for particular durations, or on particular elements of the system. So if you take this approach, you want to minimize all your exposure: decrypt infrequently, dispose of plaintext data quickly and carefully, and don't keep a plaintext version of the key in the system except when performing the cryptographic operations. Such minimization can be difficult to achieve.

If cryptography won't protect us completely against a dishonest operating system, what OS uses for cryptography are there? We saw a specialized example in the chapter on authentication. Some cryptographic operations are one-way: they can encrypt, but never decrypt. We can use these to securely store passwords in encrypted form, even if the OS is compromised, since the encrypted passwords can't be decrypted[7].

What else? In a distributed environment, if we encrypt data on one machine and then send it across the network, all the intermediate components won't be part of our machine, and thus won't have access to the key. The data will be protected in transit. Of course, our partner on the

[7] But if the legitimate user ever provides the correct password to a compromised OS, all bets are off, alas. The compromised OS will copy the password provided by the user and hand it off to whatever villain is working behind the scenes, before it runs the password through the one-way cryptographic hashing algorithm.

final destination machine will need the key if he or she is to use the data. As we promised before, we'll get to that issue in another chapter.

Anything else? Well, what if someone can get access to some of our hardware without going through our operating system? If the data stored on that hardware is encrypted, and the key isn't on that hardware itself, the cryptography will protect the data. This form of encryption is sometimes called **at-rest data encryption**, to distinguish it from encrypting data we're sending between machines. It's useful and important, so let's examine it in more detail.

56.7 At-Rest Data Encryption

As we saw in the chapters on persistence, data can be stored on a disk drive, flash drive, or other medium. If it's sensitive data, we might want some of our desirable security properties, such as secrecy or integrity, to be applied to it. One technique to achieve these goals for this data is to store it in encrypted form, rather than in plaintext. Of course, encrypted data cannot be used in most computations, so if the machine where it is stored needs to perform a general computation on the data, it must first be decrypted[8]. If the purpose is merely to preserve a safe copy of the data, rather than to use it, decryption may not be necessary, but that is not the common case.

The data can be encrypted in different ways, using different ciphers (DES, AES, Blowfish), at different granularities (records, data blocks, individual files, entire file systems), by different system components (applications, libraries, file systems, device drivers). One common general use of at-rest data encryption is called **full disk encryption**. This usually means that the entire contents (or almost the entire contents) of the storage device are encrypted. Despite the name, full-disk encryption can actually be used on many kinds of persistent storage media, not just hard disk drives. Full disk encryption is usually provided either in hardware (built into the storage device) or by system software (a device driver or some element of a file system). In either case, the operating system plays a role in the protection provided. Windows BitLocker and Apple's FileVault are examples of software-based full disk encryption.

Generally, at boot time either the decryption key or information usable to obtain that key (such as a passphrase – like a password, but possibly multiple words) is requested from the user. If the right information is provided, the key or keys necessary to perform the decryption become available (either to the hardware or the operating system). As data is placed on the device, it is encrypted. As data moves off the device, it is

[8] There's one possible exception worth mentioning. Those cryptographic wizards have created a form of cryptography called **homomorphic cryptography**, which allows you to perform operations on the encrypted form of the data without decrypting it. For example, you could add one to an encrypted integer without decrypting it first. When you decrypted the result, sure enough, one would have been added to the original number. Homomorphic ciphers have been developed, but high computational and storage costs render them impractical for most purposes, as of the writing of this chapter. Perhaps that will change, with time.

decrypted. The data remains decrypted as long as it is stored anywhere in the machine's memory, including in shared buffers or user address space. When new data is to be sent to the device, it is first encrypted. The data is never placed on the storage device in decrypted form. After the initial request to obtain the decryption key is performed, encryption and decryption are totally transparent to users and applications. They never see the data in encrypted form and are not asked for the key again, until the machine reboots.

Cryptography is a computationally expensive operation, particularly if performed in software. There will be overhead associated with performing software-based full disk encryption. Reports of the amount of overhead vary, but a few percent extra latency for disk-heavy operations is common. For operations making less use of the disk, the overhead may be imperceptible. For hardware-based full disk encryption, the rated speed of the disk drive will be achieved, which may or may not be slower than a similar model not using full disk encryption.

What does this form of encryption protect against?

- It offers no extra protection against users trying to access data they should not be allowed to see. Either the standard access control mechanisms that the operating system provides work (and such users can't get to the data because they lack access permissions) or they don't (in which case such users will be given equal use of the decryption key as anyone else).
- It does not protect against flaws in applications that divulge data. Such flaws will permit attackers to pose as the user, so if the user can access the unencrypted data, so can the attacker. For example, it offers little protection against buffer overflows or SQL injections.
- It does not protect against dishonest privileged users on the system, such as a system administrator. Administrator's privileges may allow the admin to pose as the user who owns the data or to install system components that provide access to the user's data; thus, the admin could access decrypted copies of the data on request.
- It does not protect against security flaws in the OS itself. Once the key is provided, it is available (directly in memory, or indirectly by asking the hardware to use it) to the operating system, whether that OS is trustworthy and secure or compromised and insecure.

So what benefit does this form of encryption provide? Consider this situation. If a hardware device storing data is physically moved from one machine to another, the OS on the other machine is not obligated to honor the access control information stored on the device. In fact, it need not even use the same file system to access that device. For example, it can treat the device as merely a source of raw data blocks, rather than an organized file system. So any access control information associated with files on the device might be ignored by the new operating system.

However, if the data on the device is encrypted via full disk encryption, the new machine will usually be unable to obtain the encryption

key. It can access the raw blocks, but they are encrypted and cannot be decrypted without the key. This benefit would be useful if the hardware in question was stolen and moved to another machine, for example. This situation is a very real possibility for mobile devices, which are frequently lost or stolen. Disk drives are sometimes resold, and data belonging to the former owner (including quite sensitive data) has been found on them by the re-purchaser. These are important cases where full disk encryption provides real benefits.

For other forms of encryption of data at rest, the system must still address the issues of how much is encrypted, how to obtain the key, and when to encrypt and decrypt the data, with different types of protection resulting depending on how these questions are addressed. Generally, such situations require that some software ensures that the unencrypted form of the data is no longer stored anywhere, including caches, and that the cryptographic key is not available to those who might try to illicitly access the data. There are relatively few circumstances where such protection is of value, but there are a few common examples:

- Archiving data that might need to be copied and must be preserved, but need not be used. In this case, the data can be encrypted at the time of its creation, and perhaps never decrypted, or only decrypted under special circumstances under the control of the data's owner. If the machine was uncompromised when the data was first encrypted and the key is not permanently stored on the system, the encrypted data is fairly safe. Note, however, that if the key is lost, you will never be able to decrypt the archived data.
- Storing sensitive data in a cloud computing facility, a variant of the previous example. If one does not completely trust the cloud computing provider (or one is uncertain of how careful that provider is – remember, when you trust another computing element, you're trusting not only its honesty, but also its carefulness and correctness), encrypting the data before sending it to the cloud facility is wise. Many cloud backup products include this capability. In this case, the cryptography and key use occur before moving the data to the untrusted system, or after it is recovered from that system.
- User-level encryption performed through an application. For example, a user might choose to encrypt an email message, with any stored version of it being in encrypted form. In this case, the cryptography will be performed by the application, and the user will do something to make a cryptographic key available to the application. Ideally, that application will ensure that the unencrypted form of the data and the key used to encrypt it are no longer readily available after encryption is completed. Remember, however, that while the key exists, the operating system can obtain access to it without your application knowing.

One important special case for encrypting selected data at rest is a **password vault** (also known as a **key ring**), which we discussed in the

authentication chapter. Typical users interact with many remote sites that require them to provide passwords (authentication based on "what you know", remember?) The best security is achieved if one uses a different password for each site, but doing so places a burden on the human user, who generally has a hard time remembering many passwords. A solution is to encrypt all the different passwords and store them on the machine, indexed by the site they are used for. When one of the passwords is required, it is decrypted and provided to the site that requires it.

For password vaults and all such special cases, the system must have some way of obtaining the required key whenever data needs to be encrypted or decrypted. If an attacker can obtain the key, the cryptography becomes useless, so safe storage of the key becomes critical. Typically, if the key is stored in unencrypted form anywhere on the computer in question, the encrypted data is at risk, so well designed encryption systems tend not to do so. For example, in the case of password vaults, the key used to decrypt the passwords is not stored in the machine's stable storage. It is obtained by asking the user for it when required, or asking for a passphrase used to derive the key. The key is then used to decrypt the needed password. Maximum security would suggest destroying the key as soon as this decryption was performed (remember the principle of least privilege?), but doing so would imply that the user would have to re-enter the key each time a password was needed (remember the principle of acceptability?). A compromise between usability and security is reached, in most cases, by remembering the key after first entry for a significant period of time, but only keeping it in RAM. When the user logs out, or the system shuts down, or the application that handles the password vault (such as a web browser) exits, the key is "forgotten." This approach is reminiscent of single sign-on systems, where a user is asked for a password when the system is first accessed, but is not required to re-authenticate again until logging out. It has the same disadvantages as those systems, such as permitting an unattended terminal to be used by unauthorized parties to use someone else's access permissions. Both have the tremendous advantage that they don't annoy their users so much that they are abandoned in favor of systems offering no security whatsoever.

56.8 Cryptographic Capabilities

Remember from our chapter on access control that capabilities had the problem that we could not leave them in users' hands, since then users could forge them and grant themselves access to anything they wanted. Cryptography can be used to create unforgeable capabilities. A trusted entity could use cryptography to create a sufficiently long and securely encrypted data structure that indicated that the possessor was allowed to have access to a particular resource. This data structure could then be given to a user, who would present it to the owner of the matching resource to obtain access. The system that actually controlled the resource must be able to check the validity of the data structure before granting access, but would not need to maintain an access control list.

Such cryptographic capabilities could be created either with symmetric or public key cryptography. With symmetric cryptography, both the creator of the capability and the system checking it would need to share the same key. This option is most feasible when both of those entities are the same system, since otherwise it requires moving keys around between the machines that need to use the keys, possibly at high speed and scale, depending on the use scenario. One might wonder why the single machine would bother creating a cryptographic capability to allow access, rather than simply remembering that the user had passed an access check, but there are several possible reasons. For example, if the machine controlling the resource worked with vast numbers of users, keeping track of the access status for each of them would be costly and complex, particularly in a distributed environment where the system needed to worry about failures and delays. Or if the system wished to give transferable rights to the access, as it might if the principal might move from machine to machine, it would be more feasible to allow the capability to move with the principal and be used from any location. Symmetric cryptographic capabilities also make sense when all of the machines creating and checking them are inherently trusted and key distribution is not problematic.

If public key cryptography is used to create the capabilities, then the creator and the resource controller need not be co-located and the trust relationships need not be as strong. The creator of the capability needs one key (typically the secret key) and the controller of the resource needs the other. If the content of the capability is not itself secret, then a true public key can be used, with no concern over who knows it. If secrecy (or at least some degree of obscurity) is required, what would otherwise be a public key can be distributed only to the limited set of entities that would need to check the capabilities[9]. A resource manager could create a set of credentials (indicating which principal was allowed to use what resources, in what ways, for what period of time) and then encrypt them with a private key. Any one else can validate those credentials by decrypting them with the manager's public key. As long as only the resource manager knows the private key, no one can forge capabilities.

As suggested above, such cryptographic capabilities can hold a good deal of information, including expiration times, identity of the party who was given the capability, and much else. Since strong cryptography will ensure integrity of all such information, the capability can be relied upon. This feature allows the creator of the capability to prevent arbitrary copying and sharing of the capability, at least to a certain extent. For example, a cryptographic capability used in a network context can be tied to a particular IP address, and would only be regarded as valid if the message carrying it came from that address.

[9] Remember, however, that if you are embedding a key in a piece of widely distributed software, you can count on that key becoming public knowledge. So even if you believe the matching key is secret, not public, it is unwise to rely too heavily on that belief.

Many different encryption schemes can be used. The important point is that the encrypted capabilities must be long enough that it is computationally infeasible to find a valid capability by brute force enumeration or random guessing (e.g., the number of invalid bit patterns is 10^{15} times larger than the number of valid bit patterns).

We'll say a bit more about cryptographic capabilities in the chapter on distributed system security.

56.9 Summary

Cryptography can offer certain forms of protection for data even when that data is no longer in a system's custody. These forms of protection include secrecy, integrity, and authentication. Cryptography achieves such protection by converting the data's original bit pattern into a different bit pattern, using an algorithm called a cipher. In most cases, the transformation can be reversed to obtain the original bit pattern. Symmetric ciphers use a single secret key shared by all parties with rights to access the data. Asymmetric ciphers use one key to encrypt the data and a second key to decrypt the data, with one of the keys kept secret and the other commonly made public. Cryptographic hashes, on the other hand, do not allow reversal of the cryptography and do not require the use of keys.

Strong ciphers make it computationally infeasible to obtain the original bit pattern without access to the required key. For symmetric and asymmetric ciphers, this implies that only holders of the proper key can obtain the cipher's benefits. Since cryptographic hashes have no key, this implies that no one should be able to obtain the original bit pattern from the hash.

For operating systems, the obvious situations in which cryptography can be helpful are when data is sent to another machine, or when hardware used to store the data might be accessed without the intervention of the operating system. In the latter case, data can be encrypted on the device (using either hardware or software), and decrypted as it is delivered to the operating system.

Ciphers are generally not secret, but rather are widely known and studied standards. A cipher's ability to protect data thus relies entirely on key secrecy. If attackers can learn, deduce, or guess the key, all protection is lost. Thus, extreme care in key selection and maintaining key secrecy is required if one relies on cryptography for protection. A good principle is to store keys in as few places as possible, for as short a duration as possible, available to as few parties as possible.

References

[D88] "The First Ten Years of Public Key Cryptography" by Whitfield Diffie. Communications of the ACM, Vol. 76, No. 5, May 1988. *A description of the complex history of where public key cryptography came from.*

[D+14] "The Matter of Heartbleed" by Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. Proceedings of the 2014 Conference on Internet Measurement Conference. *A good description of the Heartbleed vulnerability in OpenSSL and its impact on the Internet as a whole. Worth reading for the latter, especially, as it points out how one small bug in one critical piece of system software can have a tremendous impact.*

[G02] "Lessons Learned in Implementing and Deploying Crypto Software" by Peter Gutmann. Usenix Security Symposium, 2002. *A good analysis of the many ways in which poor use of a perfectly good cipher can totally compromise your software, backed up by actual cases of the problems occurring in the real world.*

[G17] "SHA-1 Shattered" by Google. `https://shattered.io`, 2017. *A web site describing details of how Google demonstrated the insecurity of the SHA-1 cryptographic hashing function. The web site provides general details, but also includes a link to a technical paper describing exactly how it was done.*

[GW96] "Randomness and the Netscape Browser" by Ian Goldberg and David Wagner. Dr. Dobbs Journal, January 1996. *Another example of being able to deduce keys that were not properly created and handled, in this case by guessing the inputs to the random number generator used to create the keys. Aren't attackers clever? Don't you wish they weren't?*

[K96] "The Codebreakers" by David Kahn. Scribner Publishing, 1996. *A long, but readable, history of cryptography, its uses, and how it is attacked.*

[S96] "Applied Cryptography" by Bruce Schneier. Jon Wiley and Sons, Inc., 1996. *A detailed description of how to use cryptography in many different circumstances, including example source code.*

[V16] "House of Keys: 9 Months later... 40% Worse" by Stefan Viehbock. Available on: `blog.sec-consult.com/2016/09/house-of-keys-9-months-later-40-worse.html`. *A web page describing the unfortunate ubiquity of the same private key being used in many different embedded devices.*